UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
A Compute Capable SSD Architecture for Next-Generation Non -volatile Memories /

Permalink
https://escholarship.org/uc/item/6c08r3qq

Author
De, Arup

Publication Date
2014

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Compute Capable SSD Architecture for Next-Generation Non-volatile Memories

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

in

Computer Science (Computer Engineering)

by

Arup De

Committee in charge:

Rajesh Gupta, Co-Chair
Steven Swanson, Co-Chair
James Buckwalter
Maya Gokhale
Ryan Kastner
Michael Taylor

2014

Copyright

Arup De, 2014

All rights reserved.

The dissertation of Arup De is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2014

DEDICATION

To my family and friends for their unwavering support and encouragement over the years.

EPIGRAPH

Arise! Awake! and stop not until the goal is reached.
—Swami Vivekananda (Katha Upanishad)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am thankful to many people for their support and help to reach this point and prepare this dissertation.

I would like to thank my advisers Prof. Steven Swanson and Prof. Rajesh Gupta, and my supervisor Dr. Maya Gokhale for their constant support and guidance. They teach me various aspects of research work and give me a wonderful opportunity to build a smart storage system for next-generation non-volatile memories. This work provides me a deeper insight into the field of storage systems and significantly improves my system design skills starting from hardware design to application software. I would like to thank my other committee members for their comments and suggestions throughout this process.

I would like to thank the Non-Volatile Systems Lab (NVSL) and the Microelectronic Embedded Systems Lab (MESL) members. They help me in my research work. I spend an excellent time here.

I would like to thank the Persistent Memory Architecture (PerMA) group at LLNL. I work there as a Lawrence scholar (LSP) that gives me a nice opportunity to see one of the best US national laboratories. I meet with wonderful people and get to know about several other research work in my field.

I wish to thank those at UC Santa Barbara, UC San Diego, LLNL, IIT Kanpur, NVIDIA, Infineon and Skyworks. I have had some excellent friends, colleagues, mentors and managers along the path of this journey.

I would like to thank my parents, brothers, sisters and other family members. Their love and sacrifices are invaluable and I believe this is their success too.

Thank you all.

Chapters 1, 2, 3, 4, 5 and 7 contain material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine*. The dissertation author was the first investigator and author of this paper. The material in Chapters 1, 2, 3, 4, 5 and 7 is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

Chapters 1, 2, 3, 4, 5, 6 and 7 contain material from "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation author was the third investigator and author of this paper.

Chapters 2 and 6 contain material from "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories", by Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson, wwhich appears in *MICRO-43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* The dissertation author was the second investigator and author of this paper. The material in Chapters 2 and 6 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.

ACADEMIC DISCLAIMER

The author wrote this dissertation in support of requirements for the degree Doctor of Philosophy in Computer Science at UC San Diego. The research is funded in part by the LLNL Graduate Scholars Program, and is not a deliverable for any United States government agency. The views and opinions expressed are those of the author, and do not state or reflect those of the United States government or Lawrence Livermore National Security, LLC.

LLNL DISCLAIMER

Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

VITA

2003	B. Tech. in Electrical Engineering Indian Institute of Technology Kanpur, India
2003-2004	Design Engineer Skyworks Solutions, Inc. New Delhi, India
2004-2007	Senior Design Engineer Infineon Technoloigies Bangalore, India
2008-2008	Teaching assistant University of California, Santa Barbara
2008	Internship NVIDIA Santa Clara, California
2009	M. S. in Computer Engineering University of California, Santa Barbara
2009-2014	Research assistant University of California, San Diego
2011	Internship Lawrence Livermore National Laboratory Livermore, California
2012-2014	Lawrence scholar Lawrence Livermore National Laboratory Livermore, California
2014	Ph. D. in Computer Science (Computer Engineering) University of California, San Diego

PUBLICATIONS

Arup De, Maya Gokhale, Rajesh Gupta, Steven Swanson, “Minerva: Accelerating Data Analysis in Next-Generation SSDs”, *Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2013.

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

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

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

ABSTRACT OF THE DISSERTATION

A Compute Capable SSD Architecture for Next-Generation Non-volatile Memories

by

Arup De

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2014

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

Existing storage technologies (e.g., disks and flash) are failing to cope with the processor and main memory speed and are limiting the overall performance of many large scale I/O or data-intensive applications. Emerging fast byte-addressable non-volatile memory (NVM) technologies, such as phase-change memory (PCM), spin-transfer torque memory (STTM) and memristor are very promising and are approaching DRAM-like performance with lower power consumption and higher density as process technology scales. These new memories are narrowing down the performance gap between the storage and the main memory and are putting forward challenging problems on existing SSD architecture, I/O interface (e.g, SATA, PCIe) and software. This dissertation addresses those challenges and presents a novel SSD architecture called XSSD. XSSD offloads computation in storage to exploit fast NVMs and reduce the redundant data traffic across the I/O bus. XSSD offers a flexible RPC-based programming framework that developers can use for application development on SSD without dealing with the complication of the underlying architecture and communication management. We have built a prototype of XSSD on the BEE3 FPGA prototyping system. We implement various data-intensive applications and achieve speedup and energy efficiency of 1.5-8.9× and 1.7-10.27× respectively.

This dissertation also compares XSSD with previous work on intelligent storage and intelligent memory. The existing ecosystem and these new enabling technologies make this system more viable than earlier ones.

Chapter 1

Introduction

Existing memory hierarchy comprises of fast volatile SRAM-based caches, DRAM-based main memory and slow non-volatile disk or flash-based storage. The storage technologies (e.g., disk and flash) are orders of magnitude slower than the processor and volatile memories (SRAM and DRAM). Hence, those block-based storage technologies primarily use for persistence. The processor usually loads data from the storage to the main memory in form of byte stream, creates appropriate data structures and performs various computations on that data and finally, stores data to the storage. The existing software and hardware mainly optimize for block-based slow storage systems and use various techniques such as caching and prefetching to minimize the impact of those slow storage technologies on application's performance.

Recently, we have seen a growing interest for large scale I/O or data-intensive applications such as social networks, financial modeling, scientific simulations, data mining and enterprise applications. Those applications have large datasets (in order of petabytes or larger) that can not fit into the main memory (in order of gigabytes). Thus, we need to keep those data on storage and access via read and write I/O interfaces. Those large scale applications are either streaming over large data structures or randomly access different locations on storage. Thus the existing memory hierarchy is not very effective for those applications and the overall performance is limited by the slow storage technologies.

Emerging non-volatile memory (NVM) technologies such as phase-change

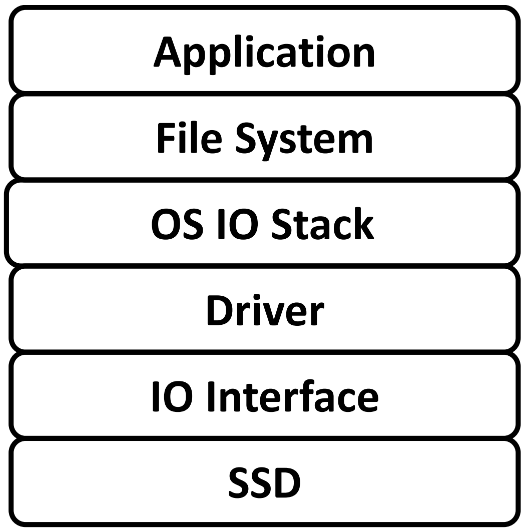

Figure 1.1: **The conventional system storage hierarchy** to access data from the storage by traversing the layers of software and the I/O bus.

memory (PCM) [Bre08], spin-transfer torque memory (STTM) [DSPE08] and memristor [Wil08] are very promising and are projected to have DRAM-like performance with lower power consumption and higher density as technology scales. They appear as a boon to those applications and achieve great attention in academia and industry as a viable replacement of existing storage technologies. However, the simple replacement of slow storage technologies with NVMs can not exploit the full potential of NVMs.

Figure 1.1 shows the conventional system storage hierarchy. It comprises of various layers: application software, file systems, operating system, driver, I/O interface (PCIe) and storage (e.g., disks and flash). The replacement of existing storage technologies with NVMs can significantly impact different layers. It can shift bottleneck from the storage to the software stack, and raise challenging problems on SSD architecture, I/O interface and software. Thus we need a thorough analysis and a broad perspective to find an efficient solution to best utilize NVMs.

This dissertation addresses various challenges with NVMs in the storage hierarchy and presents a novel SSD architecture called XSSD that colocates computation with data and supports various I/O or data-intensive computations on

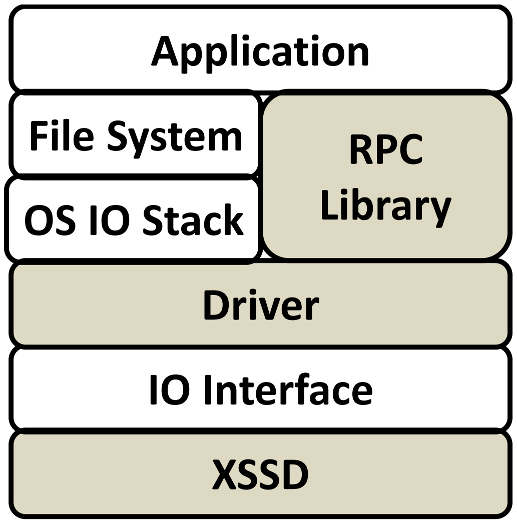

Figure 1.2: **The system storage hierarchy for XSSD** XSSD (gray boxes) extends the hardware and software to support application-specific processing close to the storage.

storage. Figure 1.2 shows the system hierarchy for XSSD. We make three major contributions: First, we extend the existing SSD architecture to offload I/O and data intensive application code to the storage array. By moving computation as close as possible to data, we can exploit the low latency and high bandwidth of these new storage technologies and reduce the data traffic between the storage and the host, resulting in a dramatic performance increase and significantly lower power consumption. Second, we enhance the existing system software stack with an RPC library to send the computational request from the host application to XSSD using simple RPC calls. We provide a runtime library that facilitates a smooth migration from the conventional I/O based storage to XSSD. Third, we develop a range of data or I/O intensive applications on XSSD using the RPC interface and compare them with the conventional I/O based implementation.

Chapter 2 discusses emerging NVM technologies. There are several NVM technologies (e.g., phase change memory (PCM), spin-transfer torque memory (STTM), memristor, magnetoresistive random access memory (MRAM), resistive random-access memory (RRAM), ferroelectric random access memory (FRAM), nano random access memory (NRAM) etc.) and they are at different levels of maturity starting from the initial research and prototyping to manufacturing [KK09]. We discuss three most promising NVM technologies such as PCM, STTM and memristor. Then we study the latest work on NVM-based storage systems and discuss the advantages and disadvantages of NVMs on the memory bus and I/O bus. Based on our real measurement, we study the impact of NVMs on existing storage hierarchies. We present motivation for XSSD and discuss salient features of XSSD to best utilize NVMs.

Chapter 3 discusses XSSD system overview and various design enhancement to exploit the full potential of NVMs and reduce the redundant data movement between the storage and the host. We enhance the host software interface and the request scheduler to easily dispatch various computational requests to XSSD. XSSD's SPU has a heterogeneous processing architecture that executes control-intensive operations on a single-issue MIPS processor and data-intensive operations on hardware accelerators. The MIPS and hardware accelerator significantly improve performance and energy efficiency. XSSD facilitates a flexible RPC based framework that enables applications to offload any application-specific functions down to the storage and provide a simple programming environment that developers can use to process data stored in the SSD without dealing with the complication of the underlying architecture and communication management. We build a prototype of XSSD on the BEE3 FPGA prototyping system [bee] for our evaluation. We implement a power model for XSSD based system to measure the power dissipation to run various workloads.

Chapter 4 presents the implementation of different I/O and data-intensive applications on XSSD. We have chosen stream and random access applications for our evaluation. XSSD uniquely supports large scale applications with poor temporal and spatial locality. Example includes key-value store and tree-traversal which often have very little data reuse and frequent random access to different data locations. As a result, they make poor use of existing memory hierarchies and perform poorly on the conventional system due to the I/O limitation. We compare the performance and the energy efficiency of those applications on XSSD with the highly optimized I/O based implementation. XSSD improves performance and energy efficiency by up to 8.91× and 10.27× for stream applications and up to 4.39× and 5.07× for applications with random accesses to storage.

Chapter 5 discusses some of the most prominent previous work on intelligent storage (e.g., Active Disks [RGF98, AUS98], Intelligent Disks [KPH98], Smart SSD [KsKMP13] and BlueDBM [JLFA14]) and intelligent memory (e.g., IRAM [PAC+97], Active Pages [OCS98] and FlexRAM [KHH+99]). Then we compare XSSD with previous work.

Chapter 6 presents XSSD infrastructure. XSSD infrastructure has several components such as PCIe, request scheduler, ring network, MIPS processor and NVM controller. We mainly focus on three main components: MIPS processor, ring network and NVM controller. In MIPS processor, we present the pipeline architecture and memory-mapped device interface. We discuss various utility applications such as loader, debugger and profiler to improve productivity. Then we discuss design details of ring network that ensures reliable communication among different components. Finally, we present NVM controller that can emulate different NVM technologies and support start-gap wear leveling technique [QKF+09].

Finally, in Chapter 7 we conclude and present some future work.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from “Willow: A User-Programmable SSD”, by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation author was the third investigator and author of this paper.

Chapter 2

Emerging Storage Technologies and Motivation

In this chapter, we discuss emerging non-volatile memory (NVM) technologies and the recent work on NVM based storage systems. We study the advantages and disadvantages of NVMs on parallel memory bus and serial I/O bus (PCIe). Then we identify problems with the existing software stack and I/O interface (e.g., PCIe, SATA) for NVMs and present motivation for XSSD.

2.1 Emerging non-volatile memory technologies

In the last five decades, the disk has been the basic unit of persistent storage for small and large scale computing systems. Disks are highly reliable but generally suffer from poor latency and bandwidth, especially relative to processor and main memory advances. Recently, flash based PCIe-attached SSDs have become popular (e.g., FusionIO [FI09] and Virident [Vir10] as representative of the high end SSDs) and are significantly faster (100×) than disk but still have slow erase, high write latency and wear-out issues. In addition, flash does not support in-place update and requires extra erase cycle to update data at same location. To overcome that drawback, SSD vendors provide a flash translation layer (FTL) between the software and flash memory which creates an illusion of in-place update and maintains the logical to physical page mapping.

Table 2.1: Memory Technology Summary [ITR09, LIMB09, PCM, GIS10, memb]

Technology	Density	Endurance	Latency		Energy	
			Read	Write	Read	Write
Flash	4 F^2	10^5	25 us	200 us	250 pJ/bit	250 pJ/bit
PCM	4 F^2	10^8	67.5 ns	215 ns	3.4 pJ/bit	17.84 pJ/bit
STTM	10 F^2	10^{15}	29.5 ns	95 ns	0.33 pJ/bit	10.0 pJ/bit
Memristor	6 F^2	10^{10}	100 ns	100 ns	2 pJ/bit	2 pJ/bit
DRAM	4 F^2	10^{18}	25 ns	25 ns	2.4 pJ/bit	2.4 pJ/bit

Emerging byte-addressable NVM technologies such as phase change memory (PCM), spin-transfer torque memory (STTM) and memristor are narrowing the performance gap between the storage and the main memory and have potential, with density improvements, to replace existing storage technologies in future. NVMs are fundamentally different than existing slow block-based storage technologies and offer orders of magnitude performance improvement and energy saving as compared to disks and flash. NVMs support in-place update and do not need the FTL between the software and memory device unlike flash.

There are several NVM technologies under research in academia and industry. Here, we present three well-known NVM technologies: phase change memory, spin-transfer torque memory and memristor. Table 2.1 briefly summarizes density, latency and energy dissipation of different memory technologies.

Phase change memory Phase change memory (PCM) is the most promising of the upcoming NVM technologies [Bre08]. It exploits the property of chalcogenide glass to switch between two states, amorphous (high-resistance) and crystalline (low-resistance), with application of current pulses. The crystalline state achieves by heating above crystallization temperature using a moderate, long current pulse, and logically stores “1”. The amorphous state achieves by high, short current pulse and logically stores “0”. A small read current (less than 100uA) used to sense data stored in a cell by measuring its resistance thus PCM consumes very small power for read. PCM approaches DRAM-like performance with lower power consumption and higher density as process technology scales. Despite this promise, PCM suffers from long write latency, high energy writes and limited write endurance (on the order of 10^8). Recent studies [LIMB09, QSR09, QKF+09, DAR09] proposed several hardware and software enhancements such as various wear-leveling methods, new row buffer design, selective writes, (DRAM/PCM) hybrid memory architecture, hot page swapping and write buffers to overcome PCM technology limitations and make them a viable option for charge-based memory replacement such as DRAM and flash.

Spin-transfer torque memory Spin-transfer torque memory (STTM) stores bits as the orientation of a magnetic layer in a magnetic tunnel junction. The junction's resistance is low (the anti-parallel) or high (the parallel state) based on the orientation [DSPE08]. STTM uses a spin-polarized current instead of electric fields of previous MRAM technologies to set the orientation. It offers low latency and high bandwidth read and writes accesses, and as technology scales, it will achieve SRAM's performance with zero standby power. It can significantly reduce the power consumption of a wide range of applications starting from mobile devices to data centers. Recent research [GIS10] also proposed a novel STTM-based resistive computation to replace the conventional CMOS technology based logic implementation in near future.

Memristor Memristor [Chu71] represents fourth passive element after the resistor, inductor and capacitor. It relies on a hysteresis effect between the current and voltage. Presently, it is a most promising Resistive RAM (RRAM) technology. The memristor device has a platinum crossbar with titanium dioxide switches. The titanium dioxide has a bipolar resistive switching that produces hysteresis loop behavior. In year 2008, HP Labs fabricated first memristor [Wil08]. The memristor is projected to offer better scalability, power, performance and endurance with scaling and is compatible with CMOS technology. The memristor is not only use as a non-volatile memory device but also use to perform arithmetic operations on stored data that can reignite the processing in memory (PIM) research.

2.2 NVMs on Memory Bus vs. I/O Bus

Recently, there is a huge interest in academia and industry for NVM-based storage systems. There are several research works on NVM-based storage systems where either NVMs are attached to the memory bus such as NV-Heaps [CCA$^+$11], BPFS [CNF$^+$09] and CDDS [VTRC11] or attached to the I/O bus (e.g. PCIe) such as Moneta [CDC$^+$10]. Those systems made various software and hardware enhancements for better utilization of NVMs.

NVMs on memory bus provide fast access to NVMs using load and store operations. NV-heaps [CCA$^+$11] project proposes a persistent object system for NVMs which facilitates a simple programming interface and protects against system and application failures by providing a model for persistence (to catch all sorts of programming errors) and supports transactional semantics. They implement various data structures such as search trees, hash tables, sparse graphs, and arrays using NV-heaps. The BPFS [CNF$^+$09] project proposes a file system and the underlying hardware enhancement for better utilization of NVMs. They optimize short-circuit shadow paging for fast and consistent updates. The hardware ensures ordering and reliability of NVM accesses. CDDS (Consistent and Durable Data Structures) [VTRC11] safely exploits fast NVMs and provides atomic update using versioning. The versioning scheme also facilitates rollback for failure recovery. CDDS implements B-Tree data structure and optimizes lookup, insert, and delete operations. However, NVMs on memory bus may suffer from scalability issues due to parallel interface and limited package pin count of the processor [ITR09].

XSSD connects with the host using PCIe which facilitates safety and scalability to sustain fast growth of data as compared to the parallel memory bus interface. The recent work on PCIe-attached NVM-based SSD architecture called Moneta-D [CMD$^+$12] focuses on optimizing read and write I/O performance for NVMs. Moneta-D provides a channel based I/O interface for each application and bypasses some portion of file system and OS overhead for different I/O calls. However, the PCIe interface and driver still limit the overall performance of Moneta-D. It inspires XSSD architecture that offloads application-specific functions down to the storage to eliminate the PCIe and driver overhead, and better utilize NVMs.

Table 2.2: Storage Systems

Name	Technology	Capacity	Description
RAID-Disk	Disk	4 TB	RAID-0 of 4x 1 TB 7200 rpm hard drives
Fusion-IO	Flash	80 GB	Fusion-IO 80 GB PCIe-attached flash-based SSD
Moneta-D	PCM	64 GB	Moneta-D 64 GB PCIe-attached PCM-based SSD

2.3 Why XSSD for NVMs?

NVMs are shifting the bottleneck from the storage to I/O interface (e.g., SATA and PCIe) and system software stack. We measure the time spent on various components of I/O for different storage technologies to get the essence of XSSD for NVMs. Table 2.2 briefly describes different storage systems. Figure 2.1 shows the I/O latency for 4 KB random read accesses on different storage systems. The total I/O latency of RAID-disk is 7.8 ms and it spends more than 99% of total I/O latency on disk due to the disk seek latency and rotational delay. Fusion-IO shows 114× improvement as compared to the disk and takes 68 μs for accessing 4 KB pages. However, the raw storage access overhead is still more than 90% of total I/O latency for flash-based SSD. Moneta-D shows the huge improvement as compared to the disk and flash and achieves I/O latency of 9 μs for 4 KB random reads. The raw NVM access overhead for Moneta-D is less than 20% of total I/O latency and the remaining overhead comes from the PCIe and software drivers. This indicates a huge scope of improvement in performance and energy efficiency with XSSD by offloading computations in storage. We also measure the read bandwidth of different storage technologies. The RAID-disk achieves 500 KB/s for 4 KB random page accesses and 125 MB/s for 4 KB sequential page accesses. The disk seek latency (∼10 ms) limits the random access performance. Fusion-IO achieves 250 MB/s for 4 KB random page accesses and outperforms disk by 250× due to relatively fast random read accesses on flash. PCM-based Moneta-D achieves 1.6 GB/s for 4 KB random page accesses which is more than 6.4× improvement as compared to Fusion-IO. However, the aggregate internal bandwidth of Moneta-

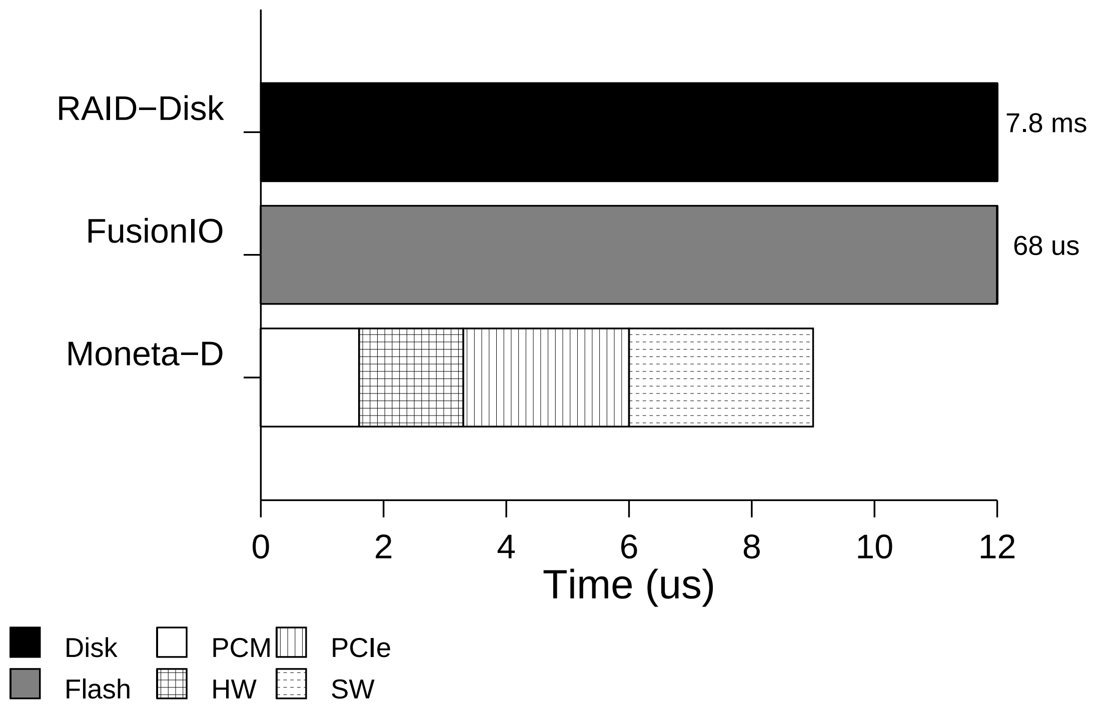

Figure 2.1: **The I/O latency breakdown for different storage technologies** the PCM access time is less than 20% of total I/O latency for 4 KB page access.

D is 23.8 GB/s and the PCIe bandwidth and software driver limit the overall Moneta-D's performance. XSSD exposes the large internal bandwidth to the end application by providing low latency and high bandwidth accesses to NVMs and eliminating PCIe and driver overhead. As the performance of NVMs increase with technology scaling, we get more advantage in terms of performance and energy efficiency by pushing the computation close to data.

XSSD also supports more flexible access patterns to leverage byte-addressable fast NVMs and provides a significantly higher level of programmability compared to the conventional storage stack. It offers a flexible RPC-based programming framework that developers can use for application development on SSD without dealing with the complication of the underlying architecture and communication management. XSSD is not limited to streaming applications and supports a wide range of I/O intensive applications with small random accesses to the storage such as key-value store and B^+tree traversal.

2.4 Motivation

XSSD extends the conventional SSD architecture to incorporate the computational capabilities in the SSD to achieve high performance and energy efficiency than the conventional SSD based system. XSSD considered all three P's of system design: performance, power and programmability. Several fundamental trends inspire XSSD architecture.

Inadequate existing SSD architecture The existing SSD architecture optimizes for block accessible flash where the overhead of I/O interface and software is negligible (less than 10%) as compared to the raw storage access overhead. NVMs are more than 1000× faster than flash and facilitate in-place update unlike flash. NVMs are radically changing the I/O latency breakdown among the software stack, PCIe and NVM. The overhead of PCIe and software stack is more than 80% of total I/O latency and the remaining overhead comes from NVM [CMD$^+$12]. So, the simple replacement of flash with NVMs actually destroy the huge gain from fast NVMs and it instigates XSSD architecture to offload application-specific processing down to the storage to exploit the low latency and high bandwidth of NVMs and reduce the redundant data movement between the storage and the host.

Huge performance and energy gain XSSD is based on NVMs which have low latency and low power dissipation as compared to existing storage technologies such as disk and flash. XSSD significantly improves performance and energy efficiency by reducing data traffic between the host and the storage, and performing efficient application-specific execution in the SSD. XSSD has a heterogeneous processing architecture that executes control-intensive operations on single-issue MIPS processor and data-intensive operations on hardware accelerator (holds fixed function hardwares).The MIPS and hardware accelerator significantly improve performance and energy efficiency. There are several research work on heterogeneous computing such as EXOCHI [WCC$^+$07] and C-cores [VSG$^+$10] have demonstrated more than 10× performance improvement and 16× energy saving as compared to general-purpose processing. We implemented various data-intensive applications on XSSD and achieved speedup and energy efficiency of 1.5-8.9× and 1.7-10.27× respectively.

Growing demand for large scale applications In the age of Big data, large scale applications such as scientific data analysis, social networks and enterprise applications need to process large amount of data in order of petabytes. For example, semantic graphs representing social networks of interest to the Department of Homeland Security will have 10^{15} entities [KBC+05]. The execution time of such large data analysis algorithms is dominated by storage performance. They are continuously spanning on three dimensions: volume (petabytes per day), velocity (real-time analytics) and variety (structured and unstructured data web logs, audio and video images). These applications often have low temporal and spatial locality. As a result, they make poor use of existing memory hierarchies and perform badly on the conventional system due to large I/O overhead. XSSD exposes huge bandwidth and low latency of NVMs to those applications and improves performance by eliminating I/O and efficient data processing in the SSD. XSSD allows the processing of the system to scale with increasing storage demand for various applications.

Ecosystem Existing ecosystem favors XSSD architecture. There are three main reasons. First, the EDA tools and FPGA/ASIC technologies are more mature than before to build XSSD system. These design methodologies can reduce the design cycle time of the hardware development. Second, the software tools, compiler infrastructure and programming languages are more user-friendly to utilize the heterogeneous processing architecture of XSSD. We have seen OpenCL becomes the de facto standard for heterogeneous computing and significantly reduces the cost of adopting a new processor architecture. Third, XSSD keeps same PCIe interface to connect with the host analogous to the existing SSD that simplifies migration from the existing SSD to XSSD. Also, it can marginally increase cost to incorporate additional computing resources.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine*. The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories", by Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson, wwhich appears in *MICRO-43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. The dissertation author was the second 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 "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14)*. The dissertation author was the third investigator and author of this paper.

Chapter 3

XSSD System Overview

As discussed in the previous chapter, emerging byte-addressable non-volatile memory (NVM) technologies are very promising and offer orders of magnitude faster accesses than existing storage technologies such as disks and flash. This huge improvement shrinks the performance gap between the storage and the main memory, and requires rethink in the storage architecture, I/O interface and system software stack to exploit fast NVMs and maximize application gain.

XSSD offloads various application-specific computations down to the storage to exploit the low latency and high bandwidth of emerging NVMs and significantly reduce data movement across the I/O bus. Figure 3.1 shows the XSSD architecture. We enhance the existing SSD with a new host software interface, request scheduler and storage processing unit to perform computation in storage. XSSD has four main components: request scheduler, storage processing unit (SPU), ring network and memory controller. We extend the host interface for XSSD to dispatch function calls from the host using Remote Procedure Call (RPC) over the PCIe. The driver offers a private communication channel interface per thread to issue an RPC call to the XSSD. The request scheduler receives it and dispatches it to the corresponding storage processing unit over the ring network. The storage processing unit (SPU) receives the RPC request and sends back acknowledgement. Then it executes that function with the given arguments and sends back a response to the host via the request scheduler using the RPC interface. The RPC hides complex message passing among different components and provides a sin-

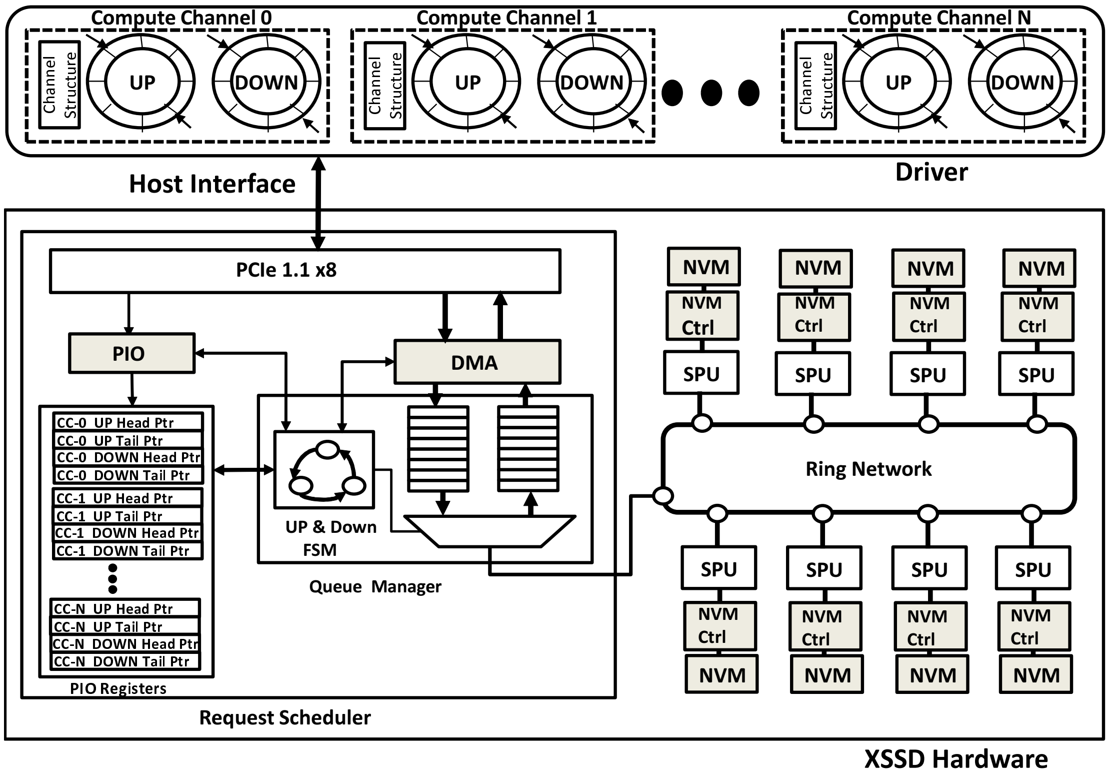

Figure 3.1: **XSSD system architecture** extends the software driver and SSD hardware with an augmented request scheduler and SPUs.

gle programming interface for the application developer to dispatch computations from the host to SPUs. Each NVM controller connects with dual-rank DIMMs using the DDR2 interface. The ring network provides a packet based reliable communication among the request scheduler and storage processing units, and has peak bandwidth and round trip latency of 3.72 GB/s and 88 ns respectively. It dispatches an RPC request from the request scheduler to the SPU based on SPU ID and packet size information and similarly, sends back response from the SPU to the request schedule. It supports hardware flow control to efficiently utilize the ring network.

We describe each component in more detail below.

3.1 Host Software Interface

Applications use a compute channel to offload computations to the storage. The channel provides a low-level communication mechanism between the SPU and the application thread. The channel has a set of control registers to manage communication, access permission and a pair of circular queues (downstream and upstream) for data communication between the request scheduler and the host. The kernel driver allocates a channel up on application request. The user space library *mmaps* it to application space and grants access to the control registers and circular queues. The application can write to the downstream queue and read from the upstream queue. The driver and hardware maintains the head and tail pointers for both the downstream and upstream queues. The host issues a Programmed IO (PIO) write to update the head pointer of the downstream queue to send data to the XSSD and issue a PIO write to update the tail pointer of the upstream queue to receive data from the XSSD. The tail pointer of the downstream queue and the head pointer of the upstream queue are managed by the request scheduler of XSSD. We use a DMA interface to send commands with various sizes along with data and the PIO interface updates different control registers, head and tail pointers of circular queues and notifies data availability to the request scheduler. When the host wants to send data to the XSSD, it opens a channel and writes the data into the downstream queue, and issues a PIO write to update the head pointer of the downstream queue. XSSD uses this information to fetch data from the host. During receive, the host polls on the upstream queue and reads all data and then issues a PIO write to update the tail pointer of the upstream queue. We provide an RPC library on top of the driver to call application-specific functions and exploit the compute resources inside XSSD. In Section 3.5 we describe our RPC library and low level message protocols in detail.

3.2 Request Scheduler

The request scheduler receives data from the host over the PCIe and sends to different SPUs over the ring network. The request scheduler has two main com-

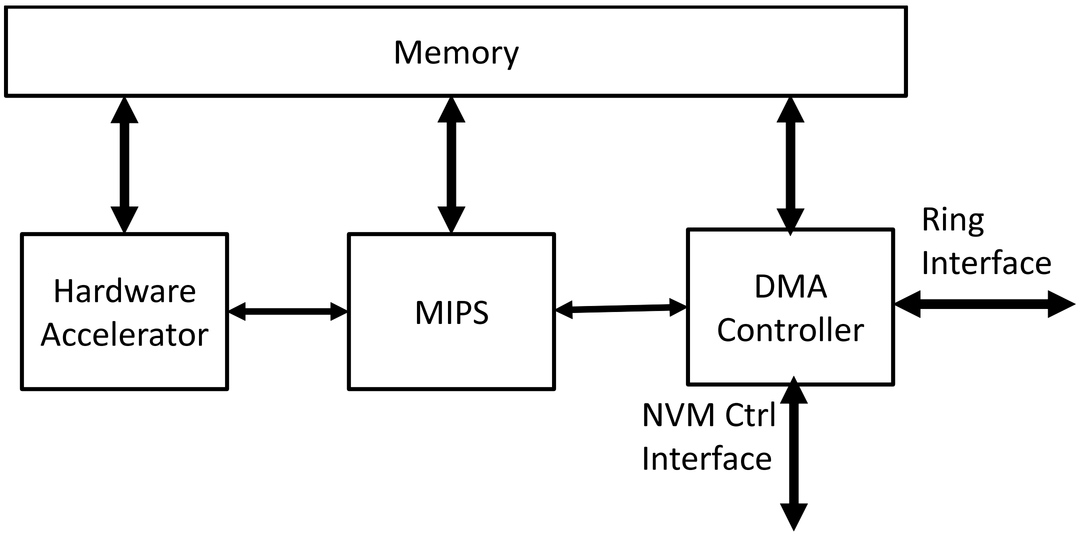

Figure 3.2: **Storage processing unit** executes application-specific functions in the SSD using MIPS. The DMA controller loads data from the NVM controller and the ring network and the hardware accelerator improves performance and energy efficiency.

ponents: PCIe interface (PIO and DMA) and queue manager. The PIO interface is responsible for read and write to various control registers and the DMA interface is responsible for issuing and handling DMA requests to send and receive data from the host. The queue manager is responsible for reading requests from the host, sending them to SPUs via ring network and writing the response back to the host. It has two finite state machines : *down_fsm* and *up_fsm*. When the host sends data to XSSD, it opens a compute channel, writes to the downstream queue and issues a PIO write to update the head pointer of the downstream queue. The *down_fsm* uses this pointer to find how many new entries are available in the downstream queue and issues the appropriate DMA requests to read data from the host. Then it checks the message header for the destination SPU and dispatches over the ring network. The *up_fsm* is similar to *down_fsm* for sending data from the XSSD to the host.

3.3 Storage Processing Unit

The storage processing unit (SPU) receives an RPC from the host via the request scheduler, executes application-specific functions and sends back a response. It comprises of four main components: MIPS, DMA Controller, hardware accelerator and local memory. We use a single issue, 5-stage pipelined 32-bit MIPS processor to run application-specific functions in the SSD. The MIPS simplifies design, reduces gate count and improves energy efficiency. The DMA controller is a memory-mapped hardware device that enables fast movement of data from the NVM controller and ring network to the local memory and vice-versa. The MIPS configures various control registers of DMA controller to set source address, destination address and data size, initiates DMA operation and waits for completion. This movement of data by the DMA controller significantly reduces the load on the MIPS. The hardware accelerator holds fixed function hardware to improve performance and energy efficiency. The MIPS sends input arguments and receives response from the hardware accelerator using a memory-mapped interface. We developed several application-specific hardware accelerators manually in Verilog. They offer the highest performance at the corresponding cost of hardware design expertise.

As an example, Figure 3.3 shows a simple streaming kernel, fgrep-8, which receives input arguments from the MIPS using a memory mapped request FIFO. The request extractor receives them and extracts the reference string and data location information. The data may reside in a local NVM, in which case it is loaded by the local NVM controller, or may reside in a remote NVM. In the latter case, the data must be read by the remote NVM controller and transferred via the ring network to a memory block local to the SPU. The fgrep-8 FSM compares the reference pattern with the data in the local memory block. On matches, the kernel increments the match word counter and when all the requested data has been searched, it sends back the results to the MIPS via the response generator by writing to the response FIFO.

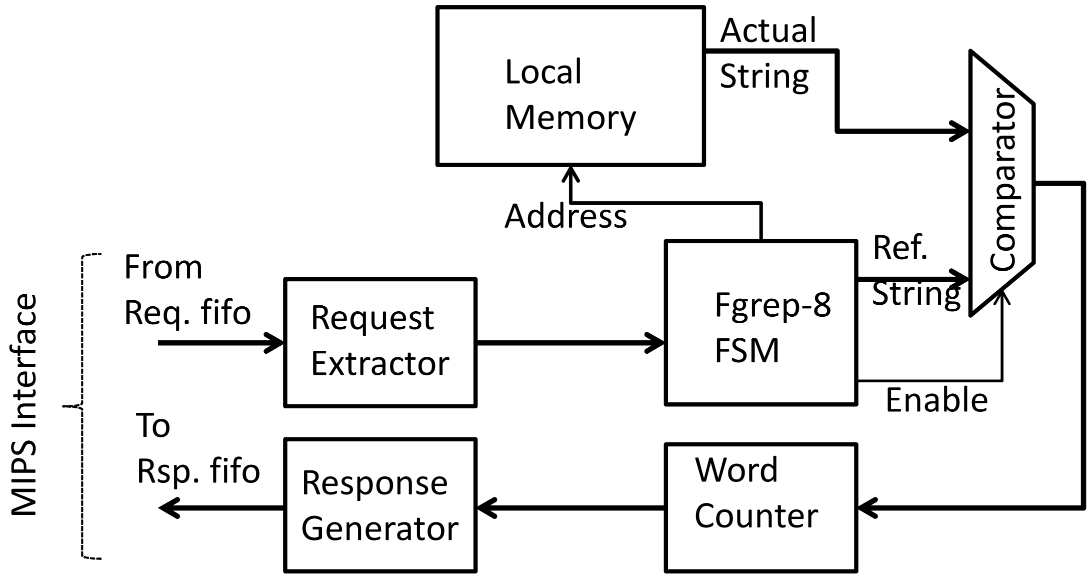

Figure 3.3: **Fgrep-8 kernel** receives compute requests from the MIPS, finds matching strings, updates the word match counter, and sends a response to the MIPS.

3.4 NVM Controller

XSSD has eight NVM controllers. Each controller connects with dual-rank DIMM using DDR2 interface with total capacity 8 GB. Each controller runs at 250 MHz, receives 16 bytes of data each cycle and has peak bandwidth of 3.72 GB/s. Each DIMM has 16 memory chips per rank, each chip has 8 banks and has a row buffer of 8 KB. NVM controller ensures persistence and writes back data from volatile row buffer to non-volatile NVM memory array in case of power failures to prevent data loss. Since phase change memory cells have limited life time (10^6 writes), the NVM controller performs the start-gap wear-leveling and address randomization scheme [QKF$^+$09] to evenly distribute write memory accesses across different physical memory pages with very little (less than 1%) overhead.

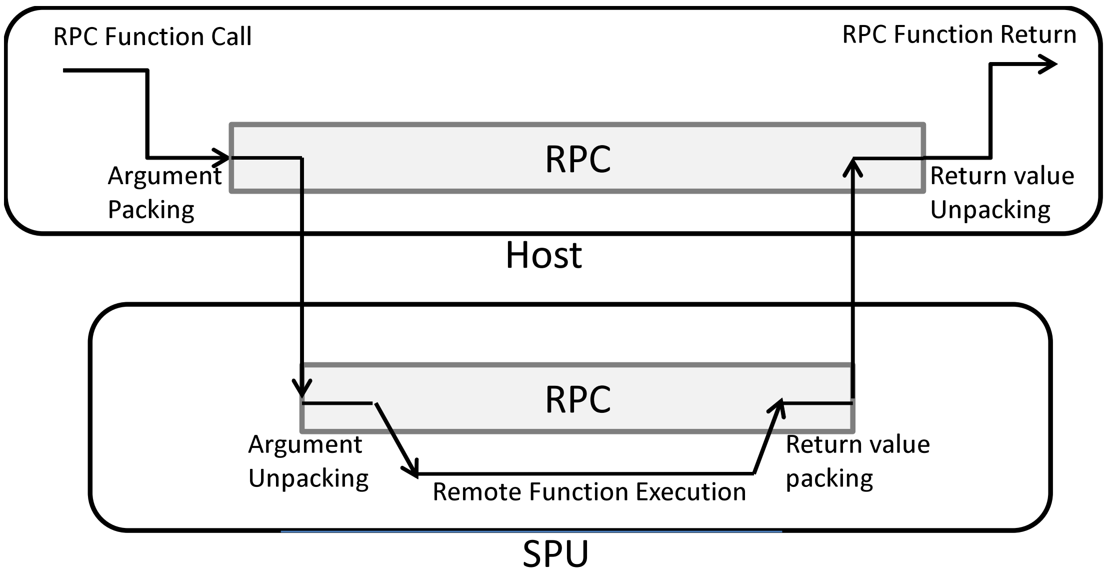

Figure 3.4: **XSSD RPC mechanism** between the host and the SPU.

3.5 Programming Model

XSSD facilitates an RPC-based framework to exploit the processing capability in storage. RPC is simple to use and is often deployed in distributed services. It is based on procedure call in a client/server setting and provides simple programmability to implement various application-specific functions in storage. Figure 3.4 shows the RPC mechanism between the host and the SPU. The host application calls an RPC function with a given arguments. The RPC library creates a packet and dispatches it to the SPU. The destination SPU unpacks it, calls the RPC function and returns back result to the host. Figure 3.5 shows the RPC packet structure for communication. The packet has following fields.

- **Source address** - identifies the sources address (host or SPU ID information)
- **Destination Address** - identifies the destination address (host or SPU ID information)
- **Length** - holds the length of the packet

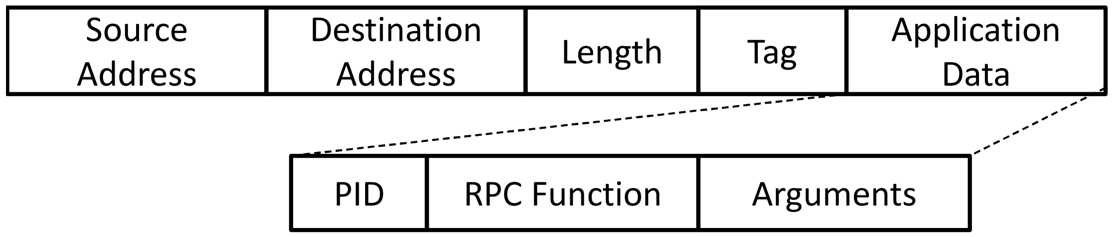

Figure 3.5: **XSSD RPC packet structure** to communicate with the SPU.

- **Tag** - keeps sequence number to maintain ordering
- **Application data** - holds application related information such as process ID (PID), RPC function and associated arguments.

The RPC library provides a set of APIs for simple communication between the host and SPUs. Table 3.1 briefly describes those RPC APIs that are responsible for creating RPC requests and responses, performing DMA operations, and sending and receiving RPC messages.

Figure 3.6 shows the typical execution flow of a data-intensive application using the XSSD architecture. If needed, the host CPU loads file data onto the XSSD. Files are striped across multiple NVM controllers for parallelism. Next, the host CPU allocates a compute channel per thread and distributes computations across multiple SPUs through RPC calls. The SPU receives a RPC request, executes it and sends back a response. The host receives RPC responses, extracts and merges results from different SPUs and produces the final output. Finally, the host CPU deallocates compute channels.

3.6 Example

As an example, the Fgrep-8 benchmark scans through a data file, searches for a specified 8-byte character pattern and counts the number of occurrences. We implement two versions of this benchmark. The first, `Fgrep-IO` uses the conventional I/O interface to read data from the SSD and perform computation on the host. Figure 3.7 presents a code snippet illustrating a parallel fgrep-IO function

Table 3.1: RPC APIs

Create RPC request and response
`BuildMsgHeader(CPUID_t dst_id, uint32_t RPC_Handler);` *Create an RPC request header with the destination ID and RPC_handler information.*
`AppendMsgBody(Port_t src, uint32_t size, Address_t src_addr );` *Append a message data from the given source (NV_MEMORY_PORT or LOCAL_MEMORY_PORT) and address.*
`CreateResponse(Msg_t* header, MsgResp_t* response, RPC_Status status);` *Extract the source and destination information from the request header and then create RPC ack/nack response message.*
DMA operation
`DMATransfer(uint32_t size, Port_t src, Address_t src_addr, Port_t dst, Address_t dst_addr);` *Supports various combinations of DMA operations: 1. NVM controller and network, 2. NVM controller and local memory, and 3. local memory and network.*
Send and receive of RPC messages
`AppendMsgBodyAndFlush(Port_t src, uint32_t size, Address_t src_addr);` *Append a message data and then send the RPC request.*
`SendResponse(response);` *Send an RPC ack/nack response.*
`ReceiveBytes(void* buf, unsigned int size);` *Receive "size" bytes from the network buffer to local memory buffer.*

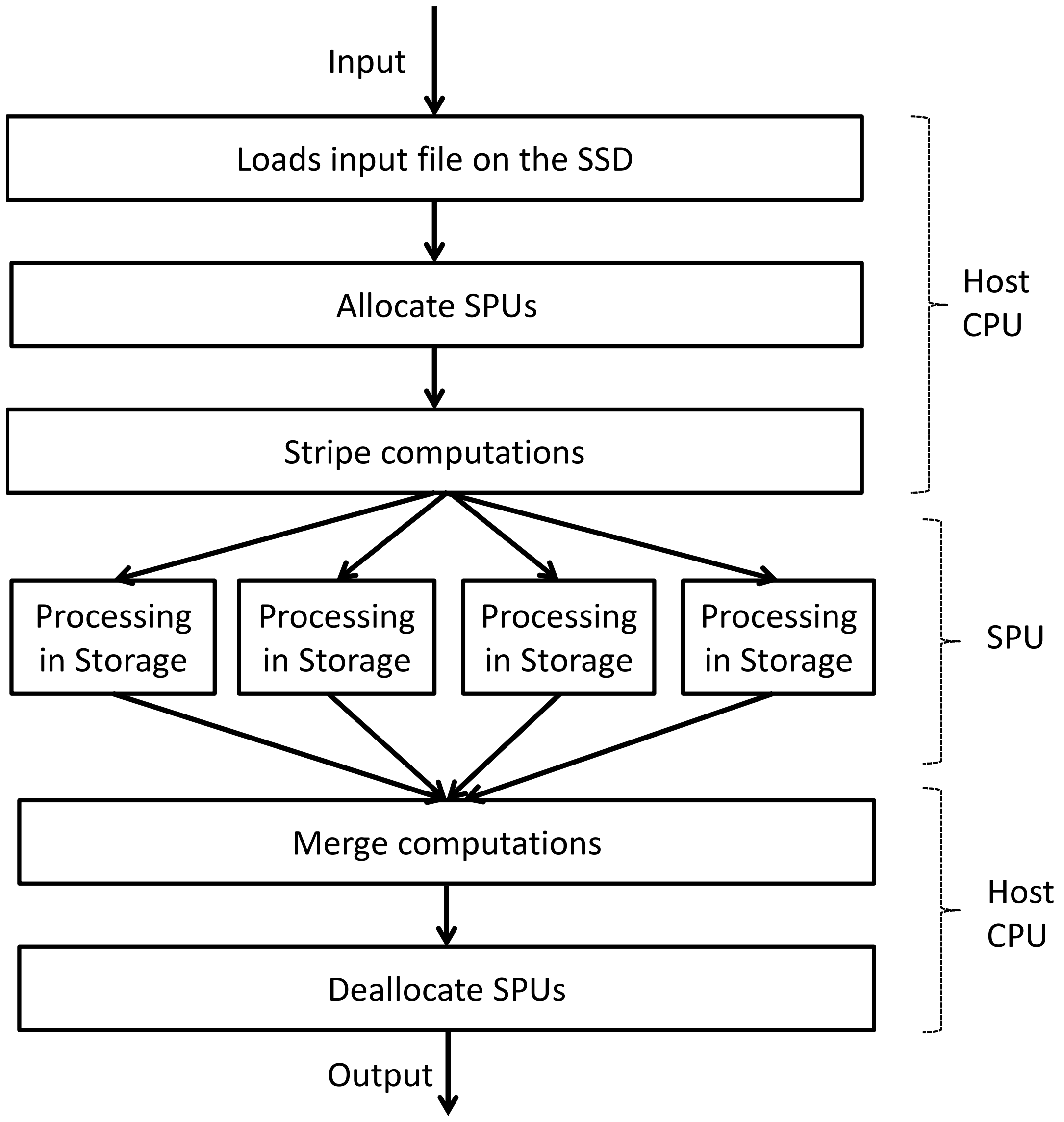

Figure 3.6: **The execution flow on XSSD based system** is initiated by the host, runs asynchronously on the SPU, and is terminated by the host.

```
Algorithm 3.6.1: FGREP-IO($int\ fd,\ int64\_t\ offset,\ size\_t\ size,\ int64\_t\ key$)

comment: Allocate local buffer
$buf \leftarrow (int64\_t*)malloc(buf\_size);$
$int64\_t\ match\_count = 0;$
for $i \leftarrow 0$ to $size$
  do
    comment: Read file with given offset
    $pread(fd,\ buf,\ buf\_size,\ offset + i);$
    comment: Search for key and increment
    for $j \leftarrow 0$ to $buf\_size$
      do
        if $buf[j] = key$
          then $match\_count \leftarrow match\_count + 1;$
    $i \leftarrow i + buf\_size;$
return $(match\_count);$
```

Figure 3.7: **Fgrep-IO code** scans through a data file and counts the number of occurrences of a given 8-byte character pattern.

using pthreads. The data file is split across multiple threads and each thread reads the file at its assigned offset. The thread reads "size" bytes of data from the SSD into a local buffer. Then it searches for a specified character pattern and increments *match_count* when a match found. Finally, the main program combines all the *match_count*s to get the total number of matches.

The second, `Fgrep-RPC` implements an RPC function FGREP-8() to count the number of matches of a specified 8-byte character pattern using the SPU and sends back the number of matches instead of fetching the data file from the storage. The host application initiates computation using an RPC call to the SPU with input arguments of the RPC function such as file extents and key. Our run-time library function extracts information on the data file extents from the file system. Upon receiving RPC, the SPU calls the RPC function FGREP-8() as shown in Figure 3.8 that retrieves command comprises of file extents and key. Then it reads file data based on the extents information using DMA, and counts the number of matches. Finally, it sends back the number of matches to the host using RPC.

```
Algorithm 3.6.2: FGREP-8($Msg\_t * header$)

comment: Retrieves file extents and key
$GrepCmd\_t\ cmd$;
RECEIVEBYTES($cmd, sizeof(GrepCmd\_t)$);
comment: Sends ack/nack response
$MsgResp\_t\ response$;
CREATERESPONSE($header, \&response, RPC\_SUCCESS$);
SENDRESPONSE($response$);
comment: Compute number of matches
$int64\_t\ spu\_match\_count = 0$;
for $i \leftarrow 0$ to $cmd.extents\_length$
  do
    comment: Use DMA to read data from NVM controller
    DMATRANSFER($buf\_size$,
    $NV\_MEMORY\_PORT$,
    $cmd.extents\_addr + i$,
    $LOCAL\_MEMORY\_PORT$,
    $buf$);
    comment: Search for key and increment
    for $j \leftarrow 0$ to $buf\_size$
      do
        if $buf[j] = cmd.key$
          then $spu\_match\_count \leftarrow spu\_match\_count + 1$;
    $i \leftarrow i + buf\_size$;
comment: Sends number of matches to the host
$CPUID\_t\ dst = cmd.src$;
BUILDMSGHEADER($dst, GREP\_COMPLETE\_HANDLER$);
APPENDMSGBODYANDFLUSH($LOCAL\_MEMORY\_PORT$,
$8, \&spu\_match\_count$);
return ;
```

Figure 3.8: **Fgrep-8 code** RPC-based implementation on SPU to count the number of occurrences of a given 8-byte key.

3.7 Prototype

We implemented XSSD on the BEE3 FPGA prototyping system jointly developed by Microsoft Research, UC Berkeley, and BEEcube Inc.. The BEE3 system holds 64 GB of 667 MHz DDR2 DRAM under the control of four Xilinx Virtex-5 LX155T FPGAs, and it provides a PCIe-1.1 x8 (2 GB/s full duplex) link to the host system. We used the high speed DDR2 ring network to connect multiple FPGAs with roundtrip latency of 88 ns and bandwidth of 3.7 GB/s. The system clock runs at 250 MHz. We implemented the hardware components such as request scheduler, SPUs and NVM controllers on the FPGAs. Since PCM is the most promising among different NVM technologies and receives great attention in research community as a viable future replacement of existing charge based memory technologies, we use PCM for our evaluation. We emulate PCM using the DRAM of the BEE3 system with access latency (read: 48 ns and write: 150 ns) as described in [LIMB09]. We modify the memory controller to add latency between the read address strobe and column address strobe commands during reads and extend the pre-charge latency after a write by inserting delay. We cannot stop DRAM refresh to preserve data which is not required for these NVM technologies. The memory controller delivers data at 3.7 GB/s. The MIPS in SPU runs at only 125 MHz, and has 32 KB instruction memory and 32 KB data memory. To overcome the slow processing of MIPS and effectively utilizing FPGA resources, we implement various application-specific hardware accelerators to improve performance.

3.8 Power Model

We implement power model of XSSD to evaluate energy consumption of XSSD based system. Table 3.2 briefly summarizes the power model of XSSD based system. The SPU power evaluation uses Synopsys CAD tools and 45 nm TSMC standard cell. The host Intel Xeon E5-2690, DDR3 DIMM and the remaining components power come from datasheets and published papers [Int08, LAS$^+$09, CDC$^+$10, LIMB09, MAC$^+$11, Int]. We measure the utilization factor of each components in XSSD using hardware counters and the host

Table 3.2: **XSSD power model** that includes the host from [Int], DIMM power from [MAC+11], PCM from [LIMB09], the SPU components from Synopsys CAD tools, and other components come from datasheets [Int08] and papers [LAS+09, CDC+10]

Component	Idle	Active
Host Intel Xeon E5-2690 [Int]	60 W	135 W
4 GB DDR3 DIMM [MAC+11]	1 W	5 W
PCIe [Int08]	0.12 W	0.4 W
Scheduler [Int08]	0.3 W	1.3 W
Network [LAS+09]	0.03 W	0.06 W
MIPS	0.078 W	0.37 W
DMA Controller	0.01 W	0.09 W
Grep Hardware Accelerator	0.013 W	0.12 W
Saxpy Hardware Accelerator	0.06 W	0.3 W
FFT Hardware Accelerator	0.029 W	0.274 W
PCM controller [LAS+09]	0.24 W	0.34 W
PCM write [LIMB09]		16.82 pJ/bit
PCM read [LIMB09]		2.47 pJ/bit
PCM background [LIMB09]	264 μW/die	20 μW/bit

CPU utilization using "sar" command. We model each component's power as $P = IdlePower \times (1 - UtilizationFactor) + ActivePower \times (UtilizationFactor)$.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 compo-

nents of this work owned by others than IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation author was the third investigator and author of this paper.

Chapter 4

Applications

In this chapter, we present a set of applications from various application domains such as scientific computing, artificial intelligence, image processing, enterprise computing and financial computing. Table 4.1 presents brief description about different applications for evaluation. These applications are data or I/O intensive. They are either stream over large data structures (e.g., fgrep-8, saxpy and 2D-FFT) or randomly accesses different data locations (e.g., key-value store and B^+ tree). The stream applications have poor temporal locality and the I/O bandwidth mainly limits the performance of those applications. The random access applications have poor temporal and spatial locality. As a result, they make poor use of existing memory hierarchies and perform poorly on the conventional system. XSSD exposes huge bandwidth and low latency of NVMs to those applications and improves the performance by eliminating I/O and performing efficient data processing in the SSD. We briefly discuss about the implementation of each

Table 4.1: Applications

Name	Description
Fgrep-8	Scans a file for a given fixed pattern matching.
Saxpy	Computes floating point scalar multiplication and vector addition.
2D-FFT	Calculates 2D-FFT.
Key-value Store	Key-value store operations lookup, insert and delete.
B^+ Tree	B^+ Tree traversal.

Table 4.2: The latency and bandwidth of access 8 KB data

Description	Latency (us)	Bandwidth (GB/s)
Host I/O	11.2	1.67
SPU DMA	3.4	2.7

applications on XSSD.

We implement two versions of each applications. The first, *[application name]-IO* uses the highly optimized storage I/O interface [CMD+12]. The second, *[application name]-RPC* uses the RPC interface of XSSD. We also implement the hardware version of fgrep-8 and key-value store. We compare the performance and energy efficiency of each implementation. We use the power model to find the average power of each component as discussed in Section 3.8. Then we calculate the system energy consumption for an application by multiplying the execution time of the application with the total average power.

Before we measure the performance of each application, we measure the access latency and bandwidth of 8 KB data access from the host over the PCIe and inside XSSD. Table 4.2 summarizes the latency and bandwidth of the host I/O access and SPU DMA access. XSSD has aggregate internal bandwidth of 21.6 GB/s that outperforms the host I/O bandwidth by 12.93×.

4.1 Fgrep-8

Fgrep-8 is a file readonly application. It scans through a data file, searches for a specified 8-byte character pattern and counts the number of occurrences. It has sequential accesses and the I/O interface bandwidth limits the overall performance of fgrep-8. We implement three versions: `Fgrep-IO`, `Fgrep-RPC` and `Fgrep-HW`.

4.1.1 Fgrep-IO

`Fgrep-IO` reads the data file using I/O, searches for the pattern and counts the number of matches using the host. We implement a parallel fgrep-IO function using pthreads that splits the data file across multiple threads with assigned offset and size. Each thread reads data from the SSD using I/O read interface and searches a given character pattern on the host. If matches, it increments the number of matches. Finally, the main program combines the number of matches of each thread to get the total number of matches.

4.1.2 Fgrep-RPC

`Fgrep-RPC` implements an RPC function FGREP-8() to count number of matches of a specified 8-byte character pattern using SPU. The host application initiates computation using RPC to the SPU with input arguments of the RPC function such as file extents and 8-byte character pattern. Our run-time library function extracts information on the data file extents from the file system. Upon receiving, the SPU calls the RPC function with given arguments. Then it loads file data from the NVM controller to local buffer using DMA based on the extents information, and counts the number of matches. Since the MIPS processor runs very slowly (at 125 MHz) and is unable to exploit the full NVM bandwidth (3 GB/s). We implemented a hardware accelerator for key comparison that can process data at 4 GB/s. As an optimization, we implement *double buffering* to simultaneously load data from the NVM controller to one local buffer and perform computation on another buffer.

4.1.3 Fgrep-HW

We also implement a hardware version of fgrep-8. It has an FSM that receives the key and file extents information from the host. Then it issues DMA requests to load data from the NVM controller to the local memory buffer based on file extents information. The FSM configures the source address, destination address and data size of a DMA operation, and waits for the DMA completion. When DMA completes, it sends the key and local buffer information to fgrep-8 hardware kernel as discussed in Section 3.3. The fgrep-8 hardware kernel compares the key with the local buffer data, increments match count when it matches and finally, sends back match count to the host.

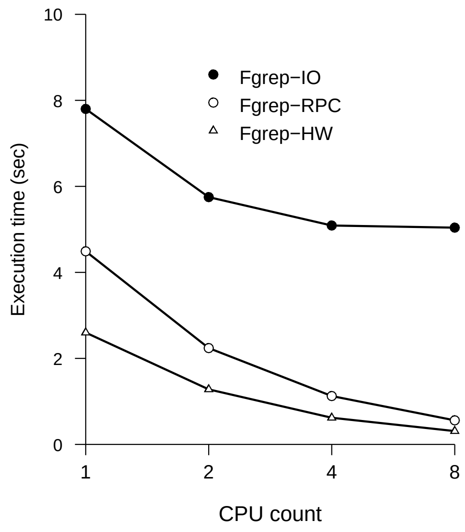

Figure 4.1: **The execution time comparison** with increasing CPU count for Fgrep-IO, Fgrep-RPC and Fgrep-HW (File size 8 GB).

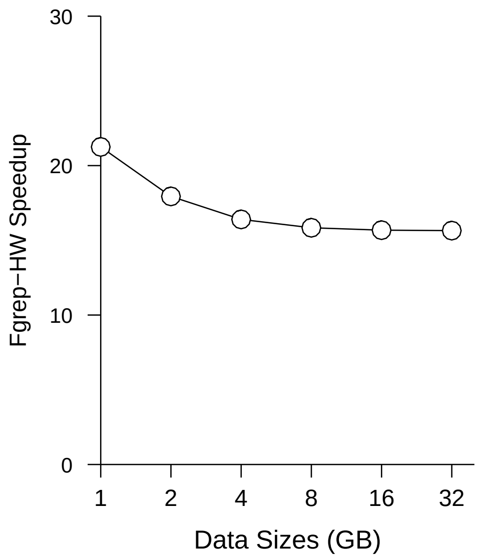

Figure 4.2: **The performance improvement of fgrep-HW as compared to fgrep-IO** as we increase the size of data file from 1 GB to 32 GB.

4.1.4 Evaluation

Fgrep-8 has sequential readonly accesses and is mainly I/O bandwidth limited which indicates a large scope of improvement with XSSD's RPC version. `Fgrep-IO` achieves maximum performance with 16 threads configuration and the total execution time is 20.06 sec to scan a 32 GB data file. This is almost 4× speedup over the single-threaded performance due to better utilization of the I/O stack and multi-core system. However, the performance of `Fgrep-IO` is mainly limited by the PCIe interface and the software driver overhead (about 86% of total execution time). `Fgrep-IO` achieves read I/O bandwidth of 1.6 GB/s that is 6.7% of XSSD's internal bandwidth. `Fgrep-RPC` exploits huge internal bandwidth and parallelism across multiple NVM controllers by running the RPC function FGREP-8() parallel on different SPUs to significantly improve performance. It eliminates storage I/O overhead by offloading computations to the SSD. `Fgrep-RPC` achieves aggregate read bandwidth of 21.6 GB/s and takes total execution time 2.25 sec that outperforms the I/O version by 8.91×. `Fgrep-IO` consumes 6454.8 J energy whereas `Fgrep-RPC` consumes 628.23 J by processing using MIPS and hardware accelerator and achieves 10.27× energy efficiency as compared to the I/O version. `Fgrep-HW` completely eliminates RPC software overhead and uses stream fgrep-8 hardware kernel to process data and achieves 1.7× improvement as compared to RPC version.

Figure 4.1 shows the performance with increasing processing power for `Fgrep-IO`, `Fgrep-RPC` and `Fgrep-HW`. `Fgrep-IO` performance saturates when number cores reaches 4. `Fgrep-RPC` shows better performance improvement as compared to `Fgrep-IO` with increasing processing power of SPUs and its speedup ranges from 1.73× to 8.91×. `Fgrep-HW` shows larger improvement for small number of cores due to lower software overhead and achives 44-73% improvement as compared to RPC version. In addition, we experiment with different sizes of dataset. Figure 4.2 shows the performance of the Fgrep-8 where we vary the data set size from 1 GB to 32 GB. `Fgrep-HW`'s speedup ranges from 15.7× to 21.25×. It achieves maximum speedup for 1 GB data since the I/O version can not exploit the full PCIe bandwidth for small datasets whereas `Fgrep-HW` exploits the internal bandwidth of SSD. Eventually, we achieve sustainable speedup of 15.7× for larger datasets. `Fgrep-RPC` facilitates software programmability but the performance goes down by 43.2% as compared to `Fgrep-HW`.

4.2 Saxpy

The saxpy is a file read-write benchmark. It is a combination of a scalar multiplication and addition of vectors. ($A = \alpha B + C$ where α is a scalar, and A, B and C are vectors). Saxpy is a very common operation in the Basic Linear Algebra Subprograms (BLAS) package. For our experiment, we used large vectors with 2 billion entries (16 GB each) that can not fit into main memory. Those vectors reside in storage and have sequential read and write accesses to fetch and update data to those vectors. We similarly implement two versions: `Saxpy-IO` and `Saxpy-RPC`.

4.2.1 Saxpy-IO

`Saxpy-IO` reads vector B and C using the conventional read I/O interface, performs computations on the host and writes back results to the vector A using the conventional write I/O interface. We implemented a parallel Saxpy-IO function using pthreads. We split vectors accross multiple threads with assigned offset and size. Each thread allocates three local buffers buf_A, buf_B and buf_C, and reads the vector B and C from the SSD to the local buffers buf_B and buf_C respectively. Then it executes saxpy operations on each item of those local buffers, produces output to the local buffer buf_A and finally, writes back data from the local buffer buf_A to specified location of vector A and exits. The main program waits to finish all threads.

4.2.2 Saxpy-RPC

The RPC version `Saxpy-RPC` implements SAXPY() on the SPU that reads specified portions of vectors from the NVM controller using DMA, performs saxpy computations, writes back to the NVM controller and finally, sends back completion notification to the host. Here we implement multiply and add unit in hardware to reduce the computation time and improve the energy efficiency.

4.2.3 Evaluation

`Saxpy-IO` operates on vectors with 2 billion entries (16 GB each) and the overall execution time is 25.18 sec. It spends 76% time on I/O and exploits the full-duplex PCIe bandwidth to read and write vectors from the storage. `Saxpy-RPC` bypasses the PCIe and software driver overheads and operates on data resides in local NVM controller. It takes only 6.29 sec and outperforms `Saxpy-IO` by 4.0×. `Saxpy-IO` and `Saxpy-RPC` consume 8104.36 J and 1784.58 J respectively, and `Saxpy-RPC` achieves 4.54× energy efficiency by running saxpy operations on SPU.

4.3 2D-FFT

FFT has been widely used in digital signal processing, wireless communication systems, image processing and various scientific applications. Here we focus on the evaluation of 2D-FFT of large matrix (NxN) that does not fit into main memory thus keeps in the storage. We implement an external 2D-FFT algorithm [BC95]. It has three steps.

1. It computes 1D-FFT on N rows.

2. Then it performs matrix transpose operation on (NxN) matrix.

3. Finally, it performs 1D-FFT on N rows of transposed matrix.

Basically, this algorithm performs 2N 1D-FFT of size N and transpose of matrix (NxN). Here we implement two versions: `2D-FFT-IO` and `2D-FFT-RPC`.

4.3.1 2D-FFT-IO

The I/O version `2D-FFT-IO` implements multi-threaded FFT computations. Initially, it distributes N/T rows to each thread (where T is the total number of threads). Then each thread issues I/Os to read a row from the storage, performs 1D FFT computation on that row and finally writes back the output to the storage. It continues those operations for all N/T rows. When all threads terminate then it starts matrix transpose operation. We use an efficient multi-threaded out-of-core block-based transpose algorithm [KBCcL03] for transpose operation. Then we again perform 1D FFT computation on each rows of transposed matrix.

4.3.2 2D-FFT-RPC

The RPC version `2D-FFT-RPC` refactors 2D-FFT computation into two parts: 1D-FFT computation and matrix transpose computation. We perform the 1D-FFT computation on the SPU and the matrix transpose computation on the host. In 1D-FFT(), we split N rows across multiple SPUs. SPU loads data from the NVM controller using DMA, performs 1D-FFT computations, stores back results to the NVM controller without the host intervention. When it finishes computation then notifies to the host using RPC. When the host receives notification from all SPUs, it performs Transpose() on the host as it requires exchange of data among different NVM controllers and synchronization. Similarly, we again perform 1D-FFT on transposed matrix using SPUs.

4.3.3 Evaluation

We implement both versions of 2D-FFT. Our FFT implementation has a large sequential access to the storage to read different rows for FFT computations. The I/O bandwidth limits the overall performance `2D-FFT-IO`. We perform 2D-FFT on large matrix N=32768 that occupies 32 GB of storage. `2D-FFT-IO` takes total 58.61 sec to perform the 2D-FFT computation in which 50.20% time spend on the 1D-FFT computation of 2N rows and it consumes 17,084 J energy. `2D-FFT-RPC` offloads the 1D-FFT computation down to the storage and exploits the huge bandwidth and parallelism across multiple NVM controllers. It reduces the overall computation time to 37.5 sec and achieves 1.7× energy savings as compared to `2D-FFT-IO`.

4.4 Key-value store

The key-value store (KVS) is becoming a fundamental building block for many large-scale enterprise applications. Examples include Amazon's Dynamo [DHJ+07], LinkedIn's Voldemort [vol], Facebook's Memcached [mema], in-line data deduplication [ZLP08] and online multiplayer games [Xbo]. Those applications have opted to use the key-value store rather than the conventional relational database because of its simplicity and better scalability. Persistent key-value stores such as BerkeleyDB [ora], Cassandra [LM10, cas], and MongoDB [PHM10] often use in-storage data structures (e.g., BTrees or hash tables). They have random accesses over large datasets and are mainly limited by the poor performance of existing storage technologies such as disk and flash.

A persistent key-value store keeps key-value data in storage and uses an in-storage hash table data structure. We use chaining for collision avoidance. The key-value store provides three operations: PUT() to insert or update a key-value pair, GET() to retrieve the value corresponding to a key, and DELETE() to remove a key-value pair. We similarly implement 2 versions: `KVS-IO` and `KVS-RPC`.

4.4.1 KVS-IO

The I/O version `KVS-IO` implements three operations: GET(), PUT() and DELETE(). During GET(), it applies hash function to a given key to get the bucket index of the hash table. Then it uses the head pointer of that bucket to traverse the in-storage linked list of key-value pairs associated with that bucket. If it finds the key-value pair with matching key then it returns that key-value pair otherwise returns NULL. During PUT(), it applies hash function to get the bucket index and then inserts given key-value pair to the corresponding in-storage linked list of that bucket. During DELETE(), similarly it applies hash function to get

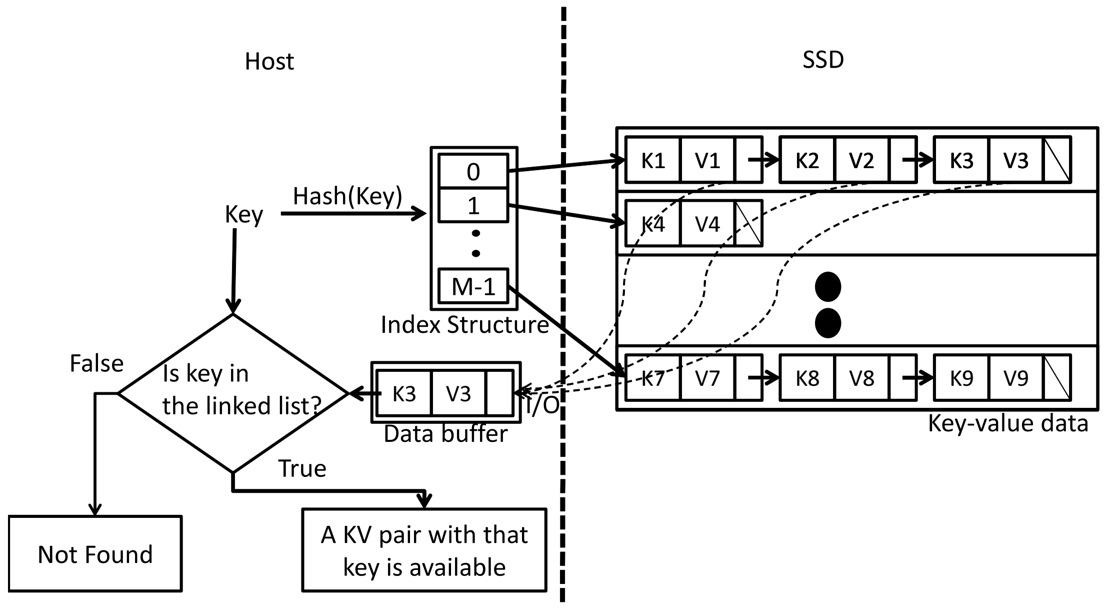

Figure 4.3: **KVS-IO GET operation** applies hash and selects bucket index. Then traveses in-storage linked list associated with that bucket using host. It issues multiple I/O requests to fetch data from the storage to the host.

the bucket index and then removes key-value pair to the corresponding in-storage linked list with matching key.

As an example, Figure 4.3 shows the I/O implementation of GET() and we are looking for key-value pair associated with a given key "K3". It has following steps:

1. The host gets the bucket index by applying hash function on a given key "K3". Here we get bucket index "0".
2. It gets the *head* pointer of the linked list from the index structure.
3. It performs the I/O read to fetch the first element (K1, V1) of the linked list, loads it to the local data buffer and compares it with a given key "K3".
4. The key does not match ($K3 \neq K1$).
5. It checks the *next* pointer to get the location of the second element of that linked list.

6. It again performs the I/O read to fetch the next element (K2, V2) of the linked list and similarly compares it with a given key "K3".
7. Again, the key does not match ($K3 \neq K2$).
8. Similarly, it uses the *next* pointer, issues the I/O read to fetch the next element (K3, V3) of the linked list and compares it with a given key "K3".
9. Now, it matches with a given key "K3" and returns key-value pair (K3, V3).

As the number of key-value pairs increases for a fixed-size index, the index will have more collisions per bucket of the hash table. For `KVS-IO`, the host must issue multiple I/O calls to traverse the collision chain for a given key-value operation. To reduce the I/O overhead, we implement `KVS-RPC`.

4.4.2 KVS-RPC

`KVS-RPC` keeps both key-value data and index structure in the storage and offloads key-value operations down to the storage to exploit the low latency, high internal bandwidth of NVMs and parallelism across multiple memory controllers. Figure4.4 shows shows the RPC implementation of key-value store. It partitions the index structure across multiple NVM controllers. Each partition of the index structure points to key-value pairs stored on that controller and it resolves collision using chaining. The host uses hashing to select a particular NVM controller/SPU using Ctrl ID field and dispatches key-value operations along with hash index to the associated SPU using RPC. The SPU performs those operations and returns results to the host.

The RPC version implements GET(), PUT() and DELETE() on the SPU. During GET(), the host sends bucket index with key and SPU traverses the linked list of that bucket using multiple short DMA requests and returns key-value pair with matching key otherwise returns NULL. During PUT(), SPU receives bucket index and key-value pair from the host and inserts the key-value pair to the linked list associated with that bucket. During DELETE(), SPU receives key and bucket

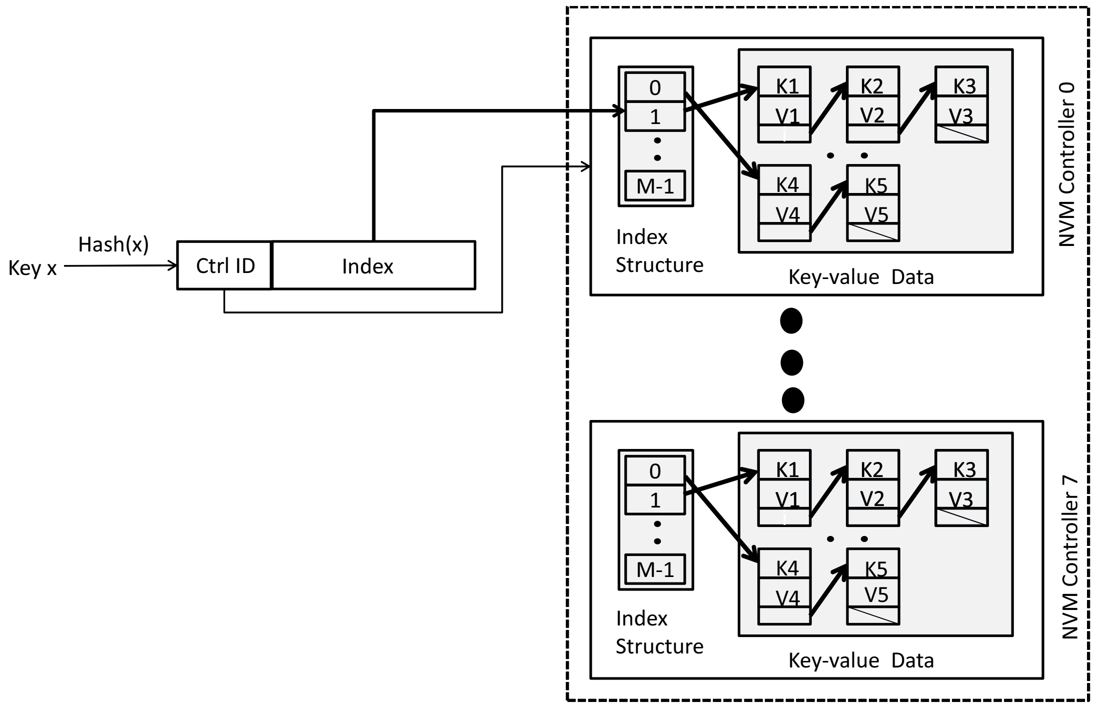

Figure 4.4: **Key-value store RPC implementation** splits the index structure across multiple memory controllers and each partition points to key-value data stored on that controller. The host applies hash function on key. The hash output has 2 fields: Ctrl ID and Index. The host selects a particular memory controller/SPU using Ctrl ID and dispatches key-value operations along with index to the associated storage processor.

index from the host, and deletes the key-value pair with matching key in that bucket.

As an example, Figure 4.5 shows the RPC implementation of GET() and we are similarly looking for key-value pair associated with a given key “K3”. Now, the computation splits between the host and SPU. It has following steps:

1. The host calculates the Ctrl ID and bucket index by applying hash function on a given key “K3”. Here we get Ctrl ID “0” and bucket index “0”.

2. The host issues an RPC to run GET() along with input arguments key “K3” and bucket index “0” on SPU 0.

3. The SPU gets the *head* pointer of the in-storage linked list from the index

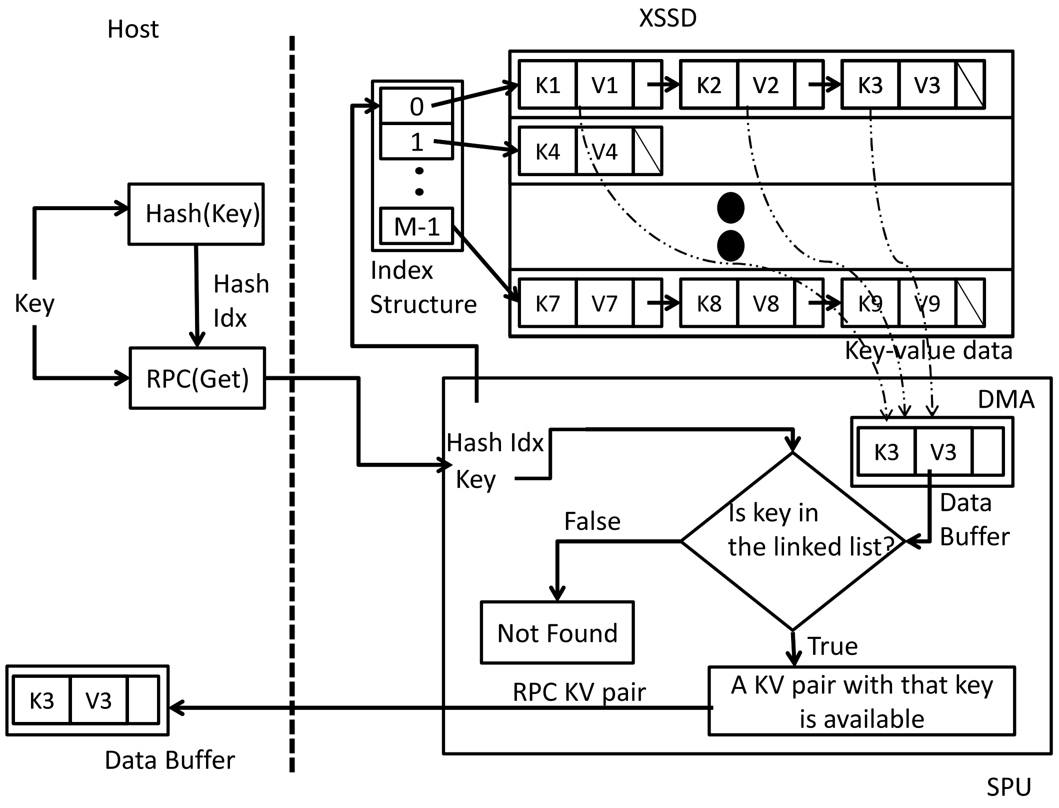

Figure 4.5: **KVS-RPC GET operation** sends key and bucket index to the SPU. SPU traverses the linked list associated with that bucket and searches for a given key. If matches it sends back key-value pair. SPU issues multiple short DMA requests to fetch data from the storage to the local data buffer.

structure to traverse the linked list.

4. The SPU issues DMA to fetch the first element (K1, V1) of the linked list, loads it to the SPU's local data buffer and compares it with a given key "K3".

5. The key does not match ($K3 \neq K1$).

6. It checks the *next* pointer to get the location of next key-value pair. Then issues DMA to fetch the next element (K2, V2) of the linked list and similarly compares it with a given key "K3".

7. The key does not match ($K3 \neq K2$).

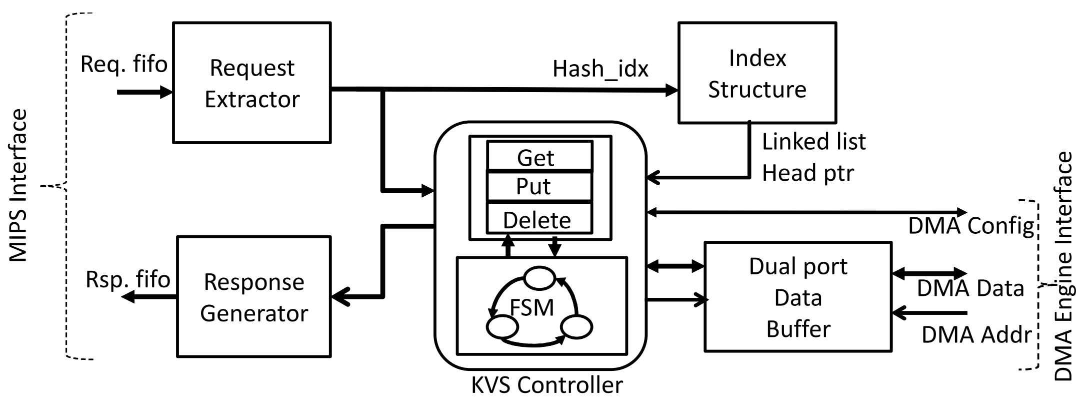

Figure 4.6: **Key-value store hardware accelerator** receives key-value operations, performs them using KVS Controller and sends back response to the MIPS. The *Get*, *Put* and *Delete* modules operate on data in a dual-port data buffer and use the DMA engine to load data from the memory controller and vice-versa.

8. Similarly, It uses the *next* pointer, reads the next element (K3, V3) of the linked list using DMA and compares it with a given key "K3".

9. Now, it matches with a given key "K3". Then SPU sends key-value pair (K3, V3) to the host using RPC. Finally, the host returns key-value pair (K3, V3).

The RPC version significantly reduces data movement between the host and the storage through PCIe channel. Also, the DMA operations are 3.3$\times$ faster than the I/O. In this example, the I/O version issues 3 I/O and the RPC version issues 1 RPC Req/Rsp + 3 DMA $\sim$ 2 I/O. So, the RPC version saves one I/O execution time.

We also implemented the same functions in hardware accelerator to improve performance. Figure 4.6 shows the architecture of the key-value store hardware accelerator. The hardware accelerator receives key-value operations from the MIPS using memory-mapped interface and enqueues to the request FIFO. Then it extracts information, applies hash index to get the head pointer of the linked list from the index structure and sends to the KVS controller. The KVS controller has a central state machine to receive information about key-value operations, dispatch them to appropriate hardware modules such as *Get*, *Put* and *Delete* to

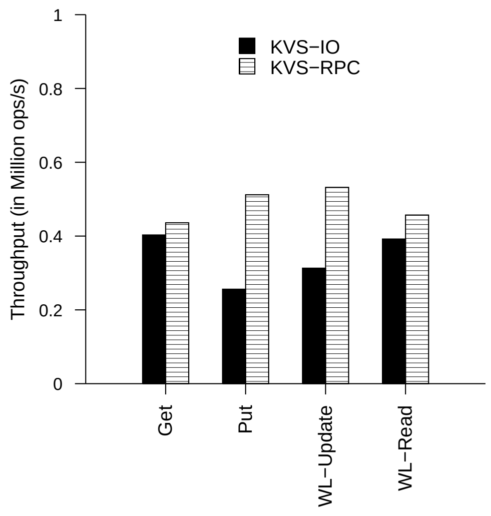

Figure 4.7: **MemcacheDB performance** Using `KVS-RPC` Get() achieves 8% improvement as compared to `KVS-IO`. Put() achieves 2× as compared I/O version by reducing data movement between the storage and host. `KVS-RPC` performs better for write-heavy workloads unlike `KVS-IO`.

perform various key-value operations and finally, sends back response to the MIPS via response generator.

`KVS-RPC` has made two major enhancements to improve performance and better utilize fast NVMs.

Avoid redundant data transfer The I/O based key-value store must issue multiple I/O requests to random storage locations to traverse the collision chain. The chain elements must be read into memory and traverse the cache hierarchy just to access the address of the next element of the chain. In the RPC based key-value store, the SPU provides faster linked list traversal as compared to the conventional I/O version by avoiding PCIe and software driver overhead. SPU only sends back the final result to the host. It significantly reduces data transfer between the host and storage with increasing chain length.

Exploit parallelism across multiple controllers The RPC based key-value store dispatches key-value operations to different SPUs to exploit the parallelism across multiple NVM controllers. SPU exploits the fine grain parallelism within a controller and keeps data movement local to effectively utilize the NVM bandwidth.

4.4.3 Evaluation

We used MemcacheDB [Chu] to evaluate both implementations of key-value store. MemcacheDB [Chu] is a persistent version of memcached [mema], the popular distributed key-value store. MemcacheDB has a client-server architecture, and, for this experiment, we run it on a single computer acting as both clients (using 16 threads configuration) and server.

First, we evaluate the performance of GET() and PUT() operations and then measure the overall performance for two workloads A) update heavy (50% PUT() / 50% GET()) and B) read heavy (5% PUT() / 95% GET()). We run all tests with 16-byte keys and 1024-byte values.

Figure 4.7 shows the performance comparison between the I/O based and RPC based implementations MemcacheDB. `KVS-IO` performs 403 GET() Kops/sec and 256 PUT() Kops/sec. `KVS-RPC` performs 436 GET() Kops/sec and 512 PUT() Kops/sec and outperforms `KVS-IO` by 1.08× and 2× respectively. `KVS-IO` performs poorly during PUT() due to poor utilization of PCIe whereas `KVS-RPC` operates close to the storage and significantly reduces data movement between the host and the storage. `KVS-RPC` nicely exploits the PCIe and parallelism across multiple NVM controllers. `KVS-IO` consumes 817.44 J and 1295 J energy to perform one million GET() and PUT() operations whereas `KVS-RPC` requires 700 J and 560 J for one million GET() and PUT() operations and saves 1.16× and 2.3× energy as compared to the I/O version.

`KVS-RPC` achieves 530 Kops/sec and 457 Kops/sec for update heavy and read-heavy workloads respectively. `KVS-IO` performs poorly on update heavy workloads and achieves less than 85% of read heavy workloads performance due to poor utilization of PCIe and ring network. For update heavy workloads, `KVS-IO` performs 313 Kops/sec and consumes 1042.32 J to perform one million operations

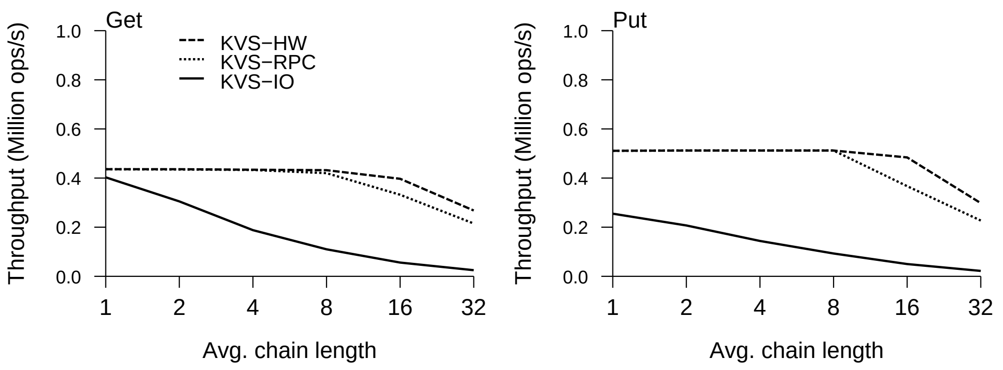

Figure 4.8: **Key-value operations as a function of chain length** `KVS-RPC` performs better than `KVS-IO` with increasing chain length and speedup ranges from 1.08× to 8.6×.

where as `KVS-RPC` performs 532 Kops/sec, and saves 1.7× energy as compared to `KVS-IO`. For read heavy workloads, `KVS-RPC` performs 457 Kops/sec and outperforms `KVS-IO` by 16.6%. `KVS-RPC` can exploit parallelism across multiple memory controllers and achieves 1.16× improvement compared read heavy workloads.

As the number of key-value pairs increases, so does the number of entries per bucket. Figure 4.8 shows affect of rising bucket chain length on Get() and Put() performance. `KVS-IO` performance drops more than 30% with eight links per chain while `KVS-RPC` see very small decreases in performance and nicely utilize PCIe bandwidth. `KVS-RPC` achieves 430,000 Get() ops/s and 510,000 Put() ops/s with chain length 8 and outperforms `KVS-IO` by 3.9×. `KVS-RPC`'s performance drops beyond chain length 8 as it requires more processing power to traverse long chain. To further improve performance, we have developed `KVS-HW` that implements key-value operations using the hardware accelerator. We observed `KVS-HW` achieves better performance as compared to `KVS-RPC` beyond chain length 8 by exploiting FPGA-based hardware acceleration.

We also study the impact of chain length on energy-delay product. Figure 4.9 shows the energy-delay product (EDP) for `KVS-IO` and `KVS-RPC` for update heavy workloads. The EDP `KVS-IO` increases quickly with with increasing chain length due to high I/O overhead. `KVS-RPC` traverses chain using SPU and improves EDP by significantly eliminating I/O and utilizing in-storage processing.

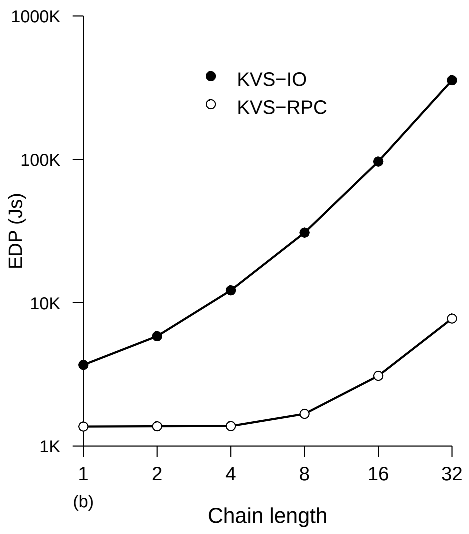

Figure 4.9: **The KVS EDP as a function of chain length** `KVS-RPC` shows up to 300% improvement in EDP as compared to `KVS-IO` by performing key-value operations using SPU.

4.5 B+ Tree

The B$^+$ tree is often used in filesystem and relational database management systems for indexing. In B$^+$ tree indexing data pointers are stored only at the leaf nodes. Assume the branching factor of B$^+$ tree is b then the number of childrens q of a given node must be less than b. This application traverses a B$^+$ tree to find a match for a given key and returns associated value. Figure 4.10 shows the B$^+$ tree traversal algorithm to find the key k. It starts from the root node and searches for a leaf node that contains key k. In each node, it checks different sub-interval to find the range of particular next child pointer that holds that key and finally it reaches to the leaf node. Then it searches the leaf node for a given key and returns the associated value.

4.5.1 B+tree-IO

The I/O version `B+tree-IO` implements above algorithm and performs tree traversal on the host. It traverses a particular path from the root to the leaf based

```
Algorithm 4.5.1: BplustreeTraversal(int fd_btree,
        int64_t root_node,
        int k)

n ← block containing root node of B^+tree;
Read block n;
comment: Traverse tree for a given key
while n is not leaf node
  do {
        q ← tree pointers in node n;
        comment: $n.key_i$ represents $i$th search key of node n
        comment: $n.p_i$ represents $i$th pointer of node n
        if $k < n.k_1$
          then { $n \leftarrow n.p_1$
          else if $k \geq n.k_{q-1}$
          then { $n \leftarrow n.p_q$
          else {
                 comment: Search for i such that $n.k_{i-1} < K \leq n.k_i$
                 $n \leftarrow n.p_i$
        Read block n;
comment: Find key k in node n and read the value v else return NULL
if key matches
  then {
         Read value v;
         return (v);
  else { return (NULL);
```

Figure 4.10: **B$^+$ tree traversal algorithm** to find a match for a given key and return associated value.

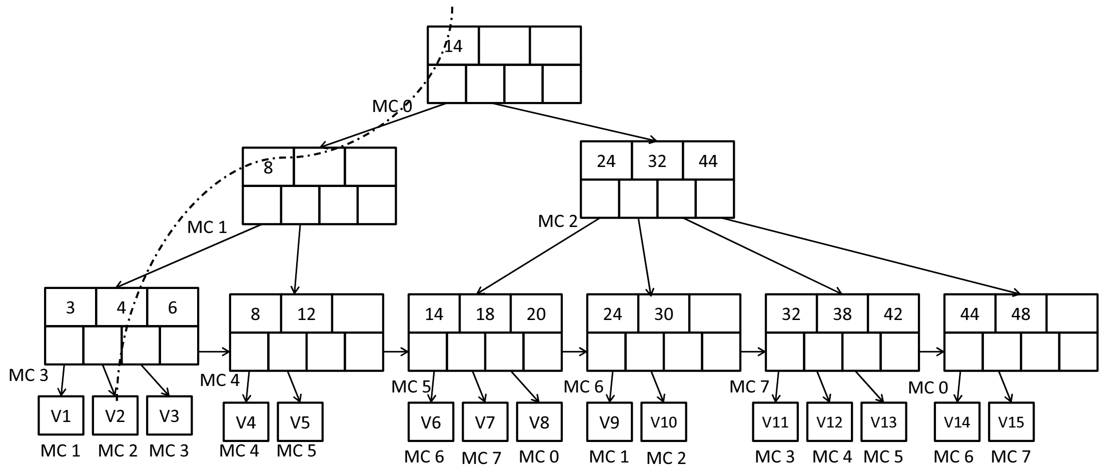
14
MC 0
8
MC 1
24 32 44
MC 2
3 4 6
MC 3
8 12
MC 4
14 18 20
MC 5
24 30
MC 6
32 38 42
MC 7
44 48
MC 0
V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
MC 1 MC 2 MC 3 MC 4 MC 5 MC 6 MC 7 MC 0 MC 1 MC 2 MC 3 MC 4 MC 5 MC 6 MC 7

Figure 4.11: **B$^+$ tree example** `B+tree-IO` issues I/O to read different nodes for a given key starting from the root node to the leaf node as shown in dotted line. `B+tree-RPC` brings those nodes from the local or remote NVM controllers to the SPU using DMA. B$^+$ tree data are striped across multiple NVM controllers from MC 0 to MC 7.

on a given key and brings different nodes on that path to the host using I/O. When it reaches the leaf node then it searches the key on that node, if matches it issues I/O to read the corresponding value from the storage.

As an example, Figure 4.11 shows a B$^+$tree where each node can have at most 3 keys. Nodes are striped across multiple NVM controllers from MC 0 to MC 7. The I/O version has same latency to access data from different NVM controllers. We need to traverse the B$^+$tree to search a given key “4” as shown in dotted line. It has following steps:

1. The host issues the I/O read to fetch the root node from the storage to the host.

2. It compares key “4” with a set of keys in that node to find the appropriate child node. Here, key “4” is less than key “14” of the root node and it selects the left pointer to read the next child node.

3. It again issues the I/O read to fetch the child node and similarly compares keys in that node with the given key to select the next child node. Now, key

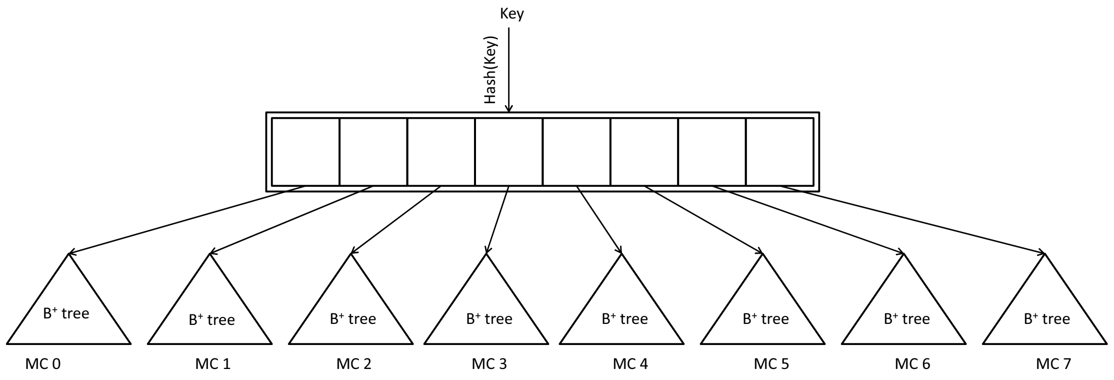

Figure 4.12: **The RPC optimization on B$^+$ tree** implements B$^+$ tree on each NVM controller. The host applies hash function on a key to find the SPU ID/NVM Controller ID and dispatches it to that SPU to perform traversal on that NVM Controller. It eliminates remote accesses to different NVM controllers over the ring network.

"4" is less than key "8" and it selects the left pointer.

4. Then it issues the I/O read to fetch the leaf node and searches for a given key in that node. If it matches read the corresponding value otherwise return NULL. Here, we find key "4" and read corresponding value "V2".

4.5.2 B+tree-RPC

The RPC version `B+tree-RPC` initially implements same algorithm and brings tree node data to the SPU using small DMA read, finds the child pointer and keeps on traversing tree from the root to leaf. Then it reads the corresponding value and returns it to the host using RPC. The RPC version has following steps to search key "4" on B$^+$tree.

1. The host issues RPC to run Traversal() along with input arguments key and send it to SPU. In this it sends to SPU 0 with key "4".

2. The SPU reads the root node using local DMA from NVM Controller MC 0 and checks keys to find the appropriate pointer for child node. In this case we find the child node at NVM Controller MC 1.

3. The SPU reads the child node from NVM Controller MC 1 using remote DMA and checks the set of keys in that node to find the next child node. In this case, we get the leaf node at NVM Controller MC 3

4. The SPU reads the leaf node using remote DMA and searches for a given key "4" and if it matches read the value otherwise return NULL. In this case we find key "4" and read corresponding value "V2" using remote DMA calls.

5. Finally, the SPU sends value "V2" to the host using RPC.

During tree traversal, as data striped across different NVM controllers, we need to bring data from the remote NVM controller over the shared ring network and the ring bandwidth (3.7 GB/s) limits the overall performance. To overcome this problem, we implement B^+ tree on each NVM controller and using hashing to select particular B^+ tree as shown in Figure 4.12. The host basically applies hash function on a given key to get the SPU ID and then dispatches it to that SPU. SPU performs tree traversal on that NVM controller that completely eliminates remote memory accesses to further improve performance.

4.5.3 Evaluation

We evaluate the performance of B^+ tree with node size 4 KB. We use 16-byte keys and 1024-byte values for this test. `B+tree-IO` performs 116,568 search ops/sec whereas `B+tree-RPC` base version performs 244,342 search ops/sec and 2.09× speedup as compared to the I/O version by performing those operations on SPU. However, the SPU needs to access remote memory controller over the ring network. Our optimized `B+tree-RPC` resolves that issue and performs 507,806 search ops/sec and achieves 2.07× speedup over the base RPC version. `B+tree-RPC` requires 628.07 J energy for one million search and that improves energy efficiency by 5.07× as compared to `B+tree-IO`. This huge improvement comes from the reduction of data transfer between the host and the storage, parallelism across multiple NVM controllers and efficient computation in the SSD.

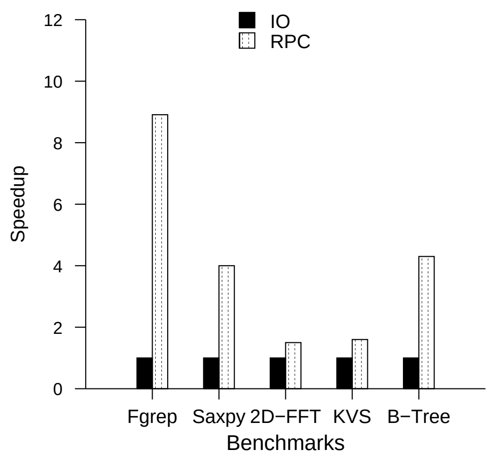

Figure 4.13: **The performance improvement of RPC based implementation** of different applications. XSSD improves performance by up to 8.91× for stream applications and up to 4.39× for applications with random accesses to storage as compared to the I/O version.

4.6 Summary

Figure 4.13 and Figure 4.14 show the performance and energy efficiency of different applications for I/O and the RPC based implementation. We use two types of applications for evaluation: stream applications (e.g., Fgrep-8, Saxpy and 2D-FFT) and random access applications (e.g., key-value store and B^+). The stream applications need more processing power and use XSSD's hardware accelerator. The hardware accelerator facilitates adequate processing power to efficiently utilize huge NVM bandwidth. It significantly improves performance and energy efficiency of the system. The RPC implementation fgrep-8 achieves 8.91× performance improvement and 10.27× energy efficiency as compared to the I/O version. The I/O version of saxpy exploits the full duplex of PCIe and the RPC version shows 4.7× performance improvement and 5.2× energy efficiency. The RPC version of 2D-FFT splits computation between the host and SPUs, and achieves 44% and 66% improvement in performance and energy efficiency respectively. The random access applications such as key-value store and B^+tree often wait for loading

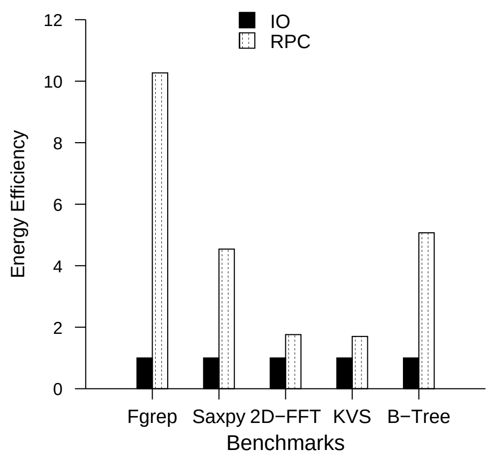

Figure 4.14: **The energy efficiency of RPC based implementation** of different applications. XSSD improves energy efficiency by up to 10.27× for stream applications and up to 5.07× for applications with random accesses to storage.

data from the NVM controller to the processor's local memory, and need very little processing. The key-value store achieves more than 1.5× performance improvement and shows better scalability with increasing key-value pairs. The RPC version of B$^+$ outperforms the I/O version by 335%. Overall, the RPC implementation of different applications optimizes for XSSD and significantly improves the performance and energy efficiency as compared to the conventional I/O version.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation author was the third investigator and author of this paper.

Chapter 5

Related Work

Several research projects have proposed colocation of computations and data in storage and memory hierarchies. Here we discuss some of the most prominent previous work on intelligent storage and intelligent memory. Then we compare XSSD with previous work.

5.1 Intelligent Storage

In the storage hierarchy, in the 80's DeWitt et al. [DH81] proposed application-specific processing on storage to exploit the disk bandwidth of database machines. However, the cost and the complexity of the application-specific hardware on disk did not allow vendors to commercialize this design. Two decades later, Active Disks [AUS98, RGF98] projects tried to take advantage of processing power on individual disk drives to run application-level code. Similarly, Intelligent Disks (IDISKs) [KPH98] were designed for decision support database servers and data warehousing workloads, and facilitated communication among multiple IDISKs using fast serial links. Recently, the Smart SSD [KsKMP13] project proposed map-reduce style data processing on flash-based SSDs to exploit the large internal bandwidth and parallelism across multiple flash channels. The Smart SSD proposal also improved the performance of various database query operations [DKP$^+$13, KOP$^+$11]. The BlueDBM [JLFA14] project proposed a high performance and scalable distributed flash storage for "Big Data" analytics.

5.1.1 Active Disks

We discuss two projects on Active Disks [RGF98, AUS98] to exploit the available computation power of disks for application level processing. The first project [RGF98] presented the feasibility of the Active Disks in the late 90's. Generally, the conventional disk consisted of a control processor, memory, a SCSI interface and a set of chips to perform error-correcting code (ECC), disk movement and SCSI processing. During that time, the Trident chip integrated specialized individual chips into a single ASIC for next-generation disks. They proposed to integrate a general-purpose processor on the disk to perform application-level processing along with various disk related operations. They also leveraged parallelism across multiple Active Disks and demonstrated up to 2× performance gain on database, data mining and image processing applications.

The second project [AUS98] proposed a stream-based programming model. The host application code interacted with application code on disks (also known as disklets) using streams. There were three types of stream: *disk-resident* stream that consisted of disk's file data, *host-resident* stream that sent information from the host application to disklets, and *pipe* stream that sent result from one disklet to other disklet. The disklet received input streams, performed application-specific disklet operations and finally produced output streams. The disklet had a initialization function, a application-specific processing function and a finalization function. They expressed applications as a number of streams and disklets, and mapped to multiple Active Disks for parallel processing. Simulation results showed that Active Disks outperformed the conventional disks by 1.07-3.15× for a set of applications including database select, nearest neighbor search, external sort, datacubes, and image processing.

5.1.2 Intelligent Disks (IDISKs)

In 90's decision support systems (DSS) were growing at the rate of 35% of database server sale [KPH98]. The standard disk-based system in a cluster was not adequate for DSS and was limited by the I/O bus bottleneck and packaging issues. In addition, the conventional host system was not optimized for DSS computations. To overcome those issues Kimberly et al. [KPH98] proposed Intelligent Disks (IDISKs) that replaced the standard disk-based systems with IDISKs in a cluster to exploit the processing capability of the disk and disk-to-disk direct communication. The IDISK consisted of an embedded processor, memory, and a fast serial interface. A crossbar switch connected multiple IDISKs using fast serial links. IDISKs ran a lightweight operating system for managing resources. They demonstrated effectiveness of IDISKs by offloading various DSS computations such as scan, sort and join from the host to embedded disk processors.

5.1.3 Smart SSD

Smart SSD [KsKMP13] leveraged the computation capability of an SSD and the aggregate bandwidth of multiple flash channels for bulk-data processing. They offloaded various data-intensive tasks (also known as tasklets) to the SSD and proposed map-reduce style processing [DG04] in which the host acted as a master node and the in-storage processing (ISP) engine acted as a slave node and performed *map* and intermediate *combine* tasks. The host performed *shuffle* and *reduce* tasks. They adopted an object-based protocol for low level communication between the host and the SSD. The SSD firmware provided three APIs to use tasklets: create object, execute object, and read object. The object-based I/O library at the host dispatched requests and received responses via SSD firmware APIs. For bulk-data processing, the Smart SSD achieved up to 4$\times$ speedup and up to 2$\times$ energy efficiency as compared to the conventional I/O-based host implementation.

Smart SSD also accelerated various database query operations [DKP$^+$13]. They implemented the Microsoft SQL server on the Smart SSD to exploit the internal SSD bandwidth and general-purpose processing capability, and achieved up to 2.7$\times$ performance improvement and up to 3.0$\times$ energy saving for various SQL query operations.

5.1.4 BlueDBM

BlueDBM (Blue Database Machine) [JLFA14] proposed a high performance and scalable flash-based storage architecture. It had multiple nodes for large scale data processing and those nodes were connected via ethernet. Each node was composed of a host PC and a PCIe-attached flash storage. The storage controller received requests from the local and remote nodes and used a two layer tagging scheme to efficiently handle those requests. It also exploited the parallelism across multiple flash channels and facilitated direct access to remote flash controllers using fast serial interconnect without the host intervention. The FPGA-based hardware accelerator in flash storage improved performance by efficient data processing and achieved more than $3\times$ performance improvement as compared to the software-only approach.

5.2 Intelligent Memory

In the memory hierarchy, the intelligent memory or processing in memory (PIM) integrated processors and memory in the same chip to address the processor-memory communication bottleneck. This idea initially proposed in several research systems such as Terasys [GHI95], J-machine [DFK+92] and EXECUBE [Kog94] but mainly suffered from the poor density of SRAM. The high density of DRAM made intelligent memory designs appealing, and there were several research works in late 90's [PAC+97, OCS98, KHH+99, BKF+99, HKK+99]. Among the most prominent, IRAM [PAC+97] integrated processor and memory in same die, Active Pages [OCS98] integrated reconfigurable logic for specialized computation on memory pages and connected with the host using the memory bus, and FlexRam [KHH+99] replaced the conventional DRAM chips with intelligent memory chips. However, those designs were not commercialized due to manufacturing and cost issues. Recently, 3D-IC technology again resurged the interest of PIM architecture in industry and academia [LGBT05, JEZ+05, MAS+06]. It basically stacked the high performance logic layer with memory layer. Micron's Hybrid Memory Cube (HMC) [Hyb] is working on 3D stacking of logic and DRAM, and the logic layer can be used to perform non-compute logic (e.g., memory controller, built-in-self-test), fixed-function hardware accelerators, and general purpose processing.

5.2.1 IRAM

The IRAM project integrated the processor and DRAM into a single chip to improve performance and energy efficiency of general-purpose processing [PAC+97, FPC+97]. IRAM [KP02] consisted of a scalar 64-bit RISC processor, a vector processor and eight 256-bit on-chip DRAM. The vector processor acted as a coprocessor and had four 64-bit datapaths for parallel data processing. IRAM extended the MIPS ISA with vector instructions for integer and floating point operations to improve performance. IRAM prototyped on 0.18 um CMOS technology and had more than 125 million transistors. It consumed 2 watts at 200 MHz and ran up to 9.6 giga-ops.

5.2.2 Active Pages

The Active Pages project integrated reconfigurable logic with DRAM to exploit the huge internal bandwidth and low latency of DRAM. It preserved the parallel memory bus interface to connect with the host for seamless transition to new DRAM chips. An active page was composed of data that resided in DRAM and a set of functions that were implemented on reconfigurable logic to operate on those data. The computation distributed between the host and the active page. The host performed complex control intensive operations and the active page performed index comparison and scatter/gather operations. The host communicated with the active page using memory read and write operations. The reconfigurable logic received virtual address from the host and performed the virtual to physical address transformation and then accessed physical pages. It generated interrupt to the host when the address location fell outside the page. The host received that information from the active page, checked page table and sent it to appropriate active page. This took extra time but simplified the hardware and software design. ActiveOS [OCS99] used this system and supported concurrent execution of multiple applications. It showed significant performance improvement over the conventional system.

5.2.3 FlexRAM

FlexRAM [KHH+99, FRF+03] replaced conventional DRAM chips with the FlexRAM processor-memory chips to exploit the low latency and high bandwidth of DRAM. It mainly addressed the programmability issues of this heterogeneous (the host processor and memory processors) machine and proposed a solution based on compiler directives. The FlexRAM chip had 64 MB DRAM organized in 64 banks, 64 PArrays and 1 P.Mem. The PArray was a simple general-purpose processor with 2 KB instruction cache and 8 KB data cache. The host and PArrays connected via FlexRAM chip controller (FXCC). The host dispatched computations using memory-mapped registers in FXCC and shared memory interface. The P.Mem was a superscalar processor to perform serial operations and coordinate among multiple PArrays. It significantly reduced the high latency accesses from the host to distribute and collect results from multiple PArrays within same chip. The software handled data coherency to maintain the consistent view of data from the host and PArrays. FlexRAM was programmed using CFlex, a scalable compiler directive based approach to split computations between the host and PArrays and handle various synchronization issues. CFlex directives were similar to OpenMP directives and kept information about the parallel code segment, execution strategy and data layout.

5.3 Comparison

XSSD is based on emerging fast byte-addressable non-volatile memory (NVM) technologies which is fundamentally different than previous work on intelligent storage such as Active Disks, IDisks, Smart SSD and BlueDBM. We will argue that these new technologies make this a compelling time to revisit the previous work and reasses the conclusions made at the time.

Active Disks and IDISKs are based on disk, and Smart SSD and BlueDBM are based on flash. Both of these technologies (disk and flash) lag behind main memory whereas NVMs are approaching DRAM-like performance with lower power consumption and higher density as process technology scales. XSSD addresses several challenges with NVM-based SSD architecture, interconnect and software stack, and proposes a new SSD architecture and a user-friendly programming framework to offload application-specific computations down to the SSD. Previous work primarily focuses on improving performance and supports streaming applications to exploit the disk characteristics such as fast sequential access. Smart SSD proposes map-reduce style data processing and BlueDBM provides infrastructure for bulk data processing whereas XSSD offers a more flexible RPC-based programming model. XSSD is not limited to streaming applications and supports a wide range of I/O intensive applications with small random accesses to the storage such as key-value store query and B+tree traversal. XSSD's highly parallel heterogeneous processing architecture achieves up to 8.27$\times$ performance improvement and 10.27$\times$ energy efficiency as compared to the I/O based implementation.

Previous work on intelligent memory addresses the processor-memory communication bottleneck and integrates processor and DRAM in same die connecting with the host using the memory bus. Intelligent memory focusses on various data structures on memory with small datasets and facilitates fine-grained synchronization between the host and memory. The overhead of accessing data through memory bus is much smaller ($\sim 300\times$) than I/O bus (PCIe). XSSD adopts various ideas from intelligent memory, focuses on large scale persistent data structures and supports coarse-grained synchronization. XSSD connects with the host using PCIe which facilitates better scalability and more advantages for offloading computations in storage (by eliminating the PCIe and software overhead) as compared to the parallel memory bus interface.

5.4 Stream Programming

The stream programming [TKA02] is a very popular programmig model to utilize massively parallel hardware resources of modern computers. Presently, serveral processors support stream programming such as GPGPU, Cell [Hof05] and multi-core CPU. The stream programming model breaks a program into a set of kernels that operate on input data streams and produce output data streams. It requires coarse-grained synchronization between kernels and streams. A stream program can be expressed by a directed graph where nodes represent different kernels and edges represent streams. There are several applications expressed as a stream program such as scientific simulations, audio and video processing.

We have seen several stream programming languages such as Brook [BFH+04], Streamit [TKA02], CUDA [NBGS08] and OpenCL [Khr08]. Presently, OpenCL is one of the most popular stream programming languages to develop applications on different processors such as multi-core CPUs, GPGPUs (AMD and NVIDIA) and Cell. Stream programs are easily mapped into parallel hardware. Optimus [HKM+08] uses the stream programming model to automatically generate fixed-function hardware for FPGAs. However, the stream programming model is limited to a set of applications. Some applications are quite difficult to express in the stream programming model such as key-value store and B^+ tree. XSSD considers a broader perspective and supports both types (stream and random access) of applications.

5.5 Key-value Store

Recently, several research works have been done on flash based key-value stores such as FlashStore [DSL10], BufferHash [AMK+10], SkimpyStash [DSL11], SILT [LFAK11] and FAWN-KV [AFK+09] to exploit flash characteristics for improving performance and reducing main memory usage.

FlashStore [DSL10] is a persistent key-value store which improves performance by using flash-aware data structures and algorithms. Here flash acts as a non-volatile cache between the main memory and the disk. FlashStore comprises of several components such as write buffer, read cache, cuckoo hasing [PR01], recency bit vector and disk-presence Bloom filter to reduce the number of I/Os. FlashStore requires 6 bytes main memory per key and achieves 5× speedup as compared to flash-based BerkeleyDB [OBS]. This huge performance gain comes from a very few accesses to flash (1 flash read/lookup) and the sequential write to flash during insertion. However, FlashStore suffers from high main memory usage with increasing key-value datasets.

BufferHash [AMK+10] proposes for data-intensive networked systems to support fast query and frequent updates. It divides the flash storage into a number of supertables. A super table consists of a buffer, an incarnation table and a set of Bloom filters. The in-memory buffer is basically a hash table to store the index structure and key-value pairs. When the buffer is full, it writes to flash. The incarnation table on flash holds the chain of those buffers (arranged chronologically). BufferHash keeps a Bloom filter for each buffer to avoid redundant search to all buffers during lookup. BufferHash requires around 4 bytes index per key and the lookup requires multiple accesses to the main memory to check the appropriate Bloom filter and then performs a flash read to load a buffer from flash. However, BufferHash suffers from poor storage utilization since buffers in the incarnation table are usually 50% occupied based on hash table load factor.

SkimpyStash [DSL11] aims to reduce the main memory usage per key-value pair. It hashes multiple keys into the same bucket of the hash table and resolves collisions with a linked list. The hash table in main memory holds the tail pointer of the linked list and a Bloom filter for each bucket. The Bloom filter per bucket holds the information about keys in that bucket and helps to decide whether the given key exists in a bucket before blindly following the linked list pointers to lookup the key-value pair in flash. Each key-value pair also contains a flash pointer to locate its predecessor in the linked list. SkimpyStash reduces the main memory usage to 1.3 bytes per key by using a fewer number of buckets with average linked list size 10. However, this requires multiple I/O reads (5 flash reads/lookup) during a lookup.

SILT [LFAK11] further reduces the memory usage to 0.7 bytes/key by using a chain of three different basic key-value stores such as LogStore, HashStore and SortedStore and orchestrates adequate transformations (LogStore to HashStore and HashStore to SortedStore). It retrieves key-value pairs using on average 1.01 flash reads each and scales to billion of key-value pairs for the conventional system. However, those trasformations adversely affect the overall performance.

FAWN-KV [AFK+09] is a distributed key-value store and has better energy efficiency as compared to the conventional host system by using low-power FAWN-DS node with 32 GB Intel SATA Flash SSD. However, it requires large main memory for indexing (6 bytes/key) and has long query latency due to the slow ring network.

However, the index structure on main memory limits the number of key-value pairs stored on flash and will lead to major scalability issues. As the number of key-value pairs increases for a fixed-size index, the index will have more collisions per bucket of the hash table and the host must issue multiple I/O calls to traverse the linked list for a given key-value operation. To overcome this problem XSSD based key-value store keeps both key-value data and index structure in the SSD and offloads linked list traversal operations down to the storage to exploit the low latency, high internal bandwidth of NVMs and parallelism across multiple NVM controllers.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation author was the third investigator and author of this paper.

Chapter 6

Infrastructure

XSSD comprises of several components such as MIPS processor, ring network, NVM controller, DMA controller, request scheduler and PCIe. Here we discuss the design details of three main components: MIPS processocer, ring network and NVM controller. We significantly optimize those components to best utilize NVM technologies and FPGA resources.

6.1 MIPS

We design the MIPS processor that provides flexibility in the implementation of our XSSD design. A processor allows us to implement new features more easily in software rather than in hardware. With software based approach, we can keep a stable hardware design and make functional enhancements through software changes. It significantly reduces the overall development time of an application program.

Figure 6.1 shows the MIPS processor that comprises of MIPS core, 32 KB instruction memory, 32 KB data memory, program loader and program debugger. We implement the Harvard architecture with separate instruction and data buses to simultaneously access the instruction and data memory to improve performance. It uses RISC architecture that facilitates a simpler instruction set and faster performance. On average, it executes one instruction per cycle. The processor runs at 125 MHz.

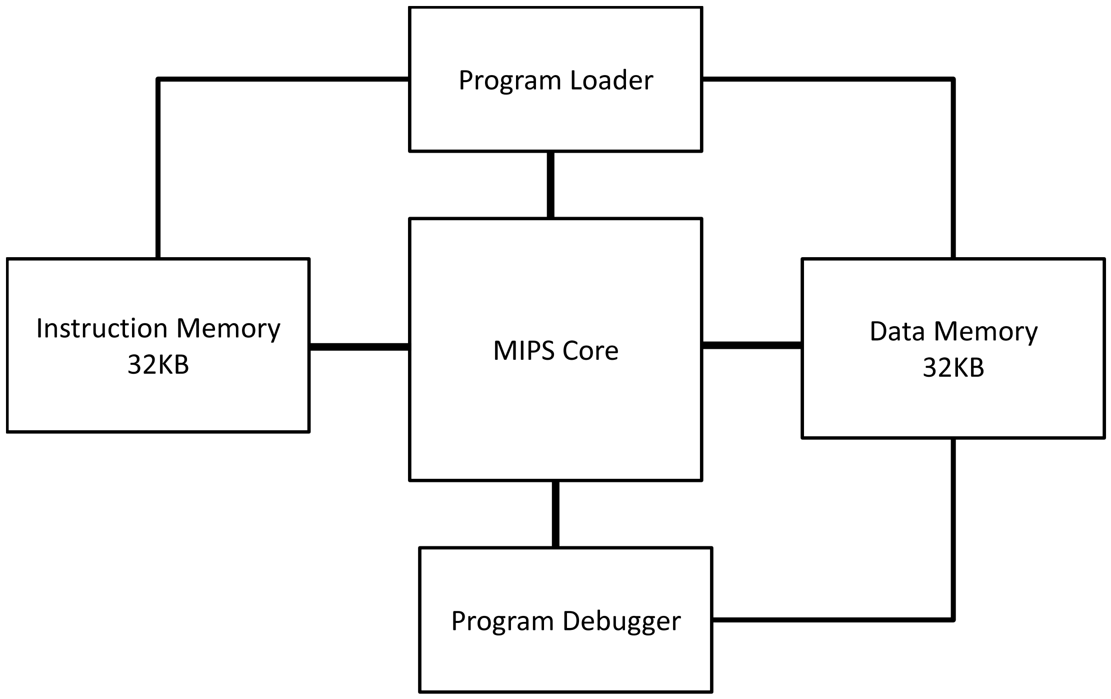

Figure 6.1: **MIPS block diagram** comprises of MIPS core, 32 KB instruction memory, 32 KB data memory, program loader and program debugger. The MIPS core has 5-stage pipeline and runs at 125 MHz. It has Harvard architecture and executes about one instruction per cycle on average. The program loader loads program to the processor's memory and the program debugger holds various debugging information to debug application software.

The program loader is responsible for downloading program to the MIPS processor. The host first configures the program loader with starting address of memory and data size information, and then sends program binary code to the program loader over the ring network. The program loader receives program binary code from the host over the ring network and loads it to processor's instruction and data memory.

The program debugger provides several features to debug application software.

- **Register file access** - support debug access to the register file to see the contents of general-purpose registers from the host
- **Memory access** - support debug access to data memory from the host

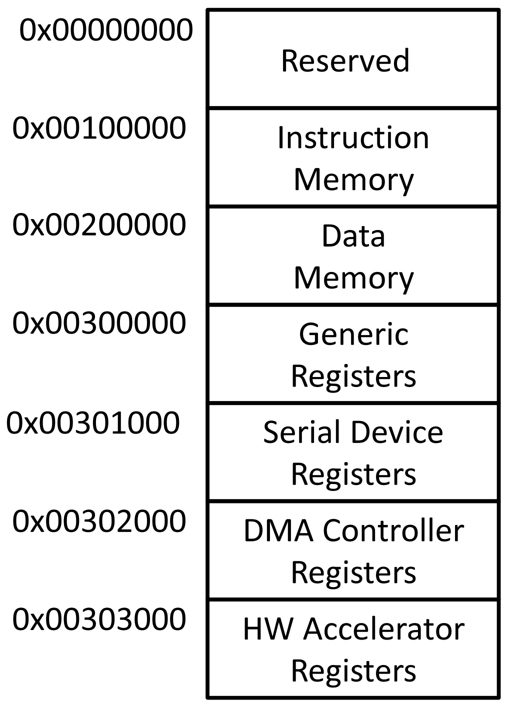

Figure 6.2: **MIPS system address space** comprises of instruction memory, data memory and various memory mapped peripherals.

- **Program counter access** - support debug access to see the program counter from the host
- **Single step execution** - support single step execution to debug an application

The program debugger has a state machine to control various debug signals and has several debug registers to hold information about the register file, data memory and program counter. It uses virtual serial interface to send debug information over the ring network.

The MIPS processor supports to connect different peripherals such as DMA controller and application-specific hardware accelerators as a memory mapped device. It communicates with them using simple register and shared-memory interface.

Figure 6.2 shows the address space of our MIPS based system. The in-

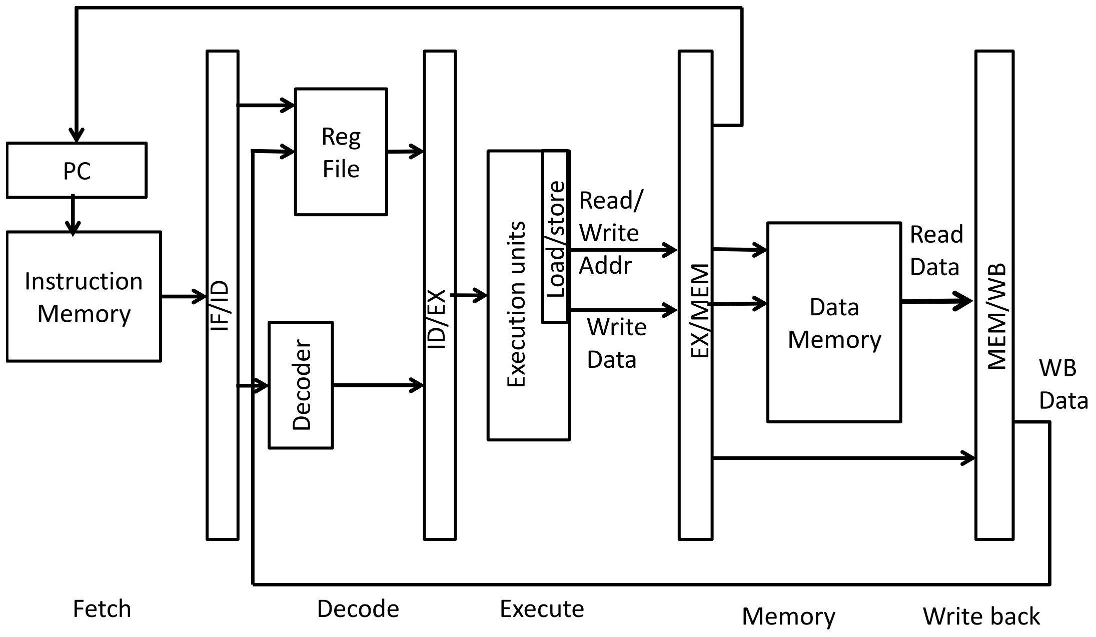

Figure 6.3: **MIPS pipeline architecture** has 5 pipeline statges: Fetch, Decode, Execute, Memory and Writeback.

struction memory starts from 0x100000 and has 1 MB program space. The data memory starts from 0x200000 and has 1 MB space. However, we use 32 KB instruction and 32 KB data memory due to FPGA resource limitations. We set the stack pointer at the bottom of the data memory. Then we allocate various memory mapped devices such as serial UART, DMA controller and hardware accelerator.

6.1.1 Pipeline Architecture

MIPS processor uses a 32-bit registers and 5-stage pipeline as shown in Figure 6.3. We implement bypass logic to forward results back through the pipeline and allow most instructions to be effectively executed in a single cycle. We implement hazard detection logic to check read-after-write (RAW) hazards and stall the pipeline. It starts executing again after resolving dependency problem.

The five pipeline stages are:

- **Fetch** - The instruction is read from the instruction memory for a given address.

- **Decode** - The instruction is decoded, and then we fetch operands from register file or bypass logic.
- **Execute** - It performs an operation based on given instruction. Simple instructions finish in this stage (e.g., addition, subtraction and logical operations).
- **Memory** - It performs memory access in this stage. For example, store instruction write data to the memory.
- **Writeback** - It updates the register file with the instruction output. For example, load instruction reads data from the memory and update a register in this stage.

6.1.2 Software Tool Chain

We use several utility programs to improve productivity.

Compiler and Linker

We use GCC MIPS cross-compiler to compile C/C++ source code. The boot code is implemented in assembly language to initialize registers (e.g., stack pointer and global pointer) and memory, and then jump to the main program. The linker script controls memory layout (e.g., instruction memory, data memory, stack, global memory etc.). MIPS linker uses this script to generate the final program. We provide a Makefile to simplify the build process.

Loader

We develop loader utility to download programs on MIPS. First, it resets MIPS processor and sends starting memory address and data size information to the program loader. Then it downloads program to the instruction and data memory of the MIPS using program loader. When download completes, it sets program counter (PC) and deasserts reset signal.

Debugger

The debugger utility provides several features to debug the application code running on MIPS. It receives debug information from the MIPS virtual serial device (printf statements) over the ring network and prints them using the host minicom terminal. We provide more visibility to the MIPS core. A programmer can use the program debugger hardware to see the contents of the register file, data memory and program counter. We support single step execution for debugging. It also detects the stack overflow error and stalls the processor.

Profiler

We also implement profiling support to capture the execution time of different functions running on MIPS with minimal software overhead. The profiler has a MIPS memory buffer and it assigns unique memory location for each function to store the profiling information (e.g., function call time and frequency). It uses hardware support to read the 64-bit timer counter (e.g., start time and end time of a function), calculate the execution time of the function and finally, write it to the memory buffer. The host utility program reads that memory buffer and dumps it to a file. It significantly reduces the load on the MIPS processor to capture the profiling information. We extensively use this profiler to optimize various functions in our RPC library and applications.

6.2 Ring Network

We use token ring network for connecting different components such as request scheduler, memory controllers and SPUs. The router is an essential part of this framework which provides the reliable communication across different components. Each router has a unique ID (*router_id*). Each component called client connects with a router to communicate with other components.

Figure 6.4 shows the router architecture. The router moves a small pulse called token around the network. The possession of the token grants the right to transmit. At a time, there is only one sender and one receiver. The sender client

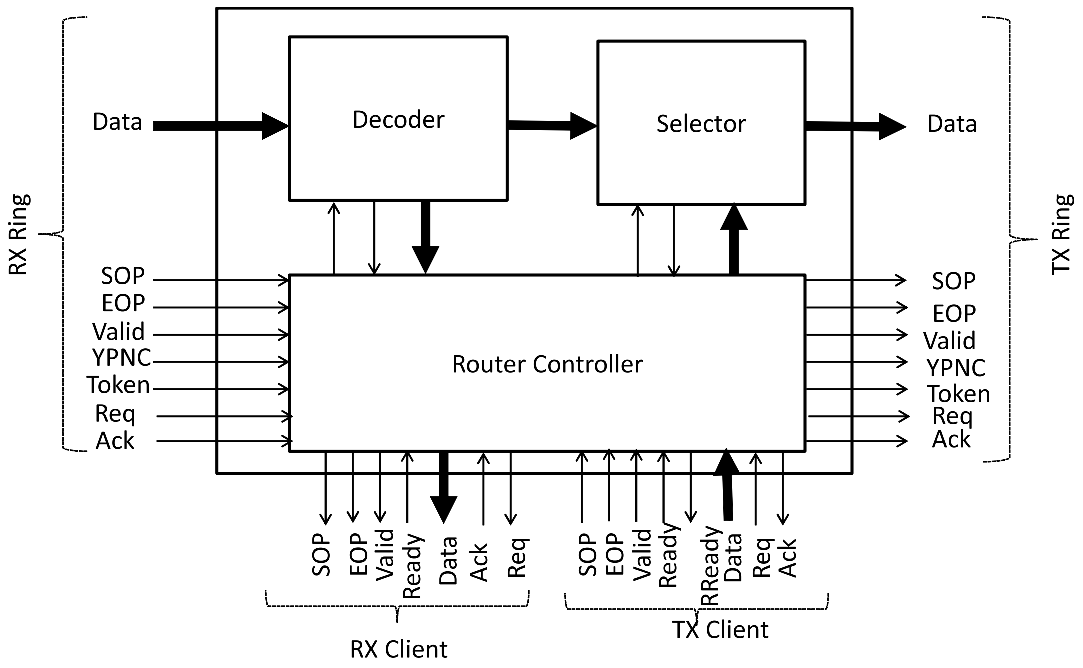

Figure 6.4: **Router architecture** receives data along with control signals from the ring network, sends it to corresponding client or forwards it to other router. It holds token and sends data from the client to destination over the ring network.

holds the token to send a request to the receiver client. It releases the token after sending the request. If a client that receives the token has no information to send, it forwards the token to the next router otherwise it seizes the token, and sends a request to other client over the ring network.

The router is responsible for decoding the incoming request and if it matches with the *router_id* then it sends it to that client for further processing otherwise forwards it to the next router. When a request receives by the receiver client it sends acknowledgment to the sender router.

Router controller has two state machines: FSM_TX and FSM_RX. The FSM_TX is active when it receives the token and the client has data to send (In TX client, ready is high) otherwise it is in idle state. Selector module has a multiplexer to select data from TX client interface or RX ring interface based control signals from the router controller. Then it dispatches them to the network via TX ring interface. During transmit, it sends the header with SOP (start of packet) and valid signals, then it sends the payload with valid signal only and finally, it sends last 16 bytes with EOP (end of packet) and valid signals. The FSM_RX is always active and checks different control signals from the ring network. When it finds SOP and valid signals (indicate header data) then it triggers decoder to check the destination router field of the header and if it matches with the *router_id* then it sends that packet to the client via RX client interface.

We implement two types of flow control for reliable communication over the ring network.

- **Request-acknowledge flow control** - When the sender wants to transmit a packet to the receiver. First, it ensures that the receiver client has enough space to receive the packet and then only it sends the packet. Initially, the sender sends a request with packet size information to the receiver. The receiver gets the request, checks the available space. If there is enough space then it sends *ack* otherwise returns *nack*. In case of *ack*, the sender transmits the packet over the ring network otherwise it releases the token for other clients. This flow control nicely utilizes the ring bandwidth for large packets and does not stall the ring network. However, it has initial handshaking overhead.

- **Young packet not consume (YPNC) flow control** - Here we do not use initial handshaking between the sender and receiver. So, it may possible that the receiver does not consume the full packet. In that case, the receiver sends notification to the router associated with that client. The router asserts YPNC signal and leaves the remaining portion of that packet on the ring network. Those remaining data keep on circulating over the ring network. When the receiver is ready again, the router receives those data from the ring network and sends to the client. Since this flow control does not need initial handshaking and it is particularly suitable for small packets. However, it wastes ring bandwidth and stalls other clients to send packets when a router asserts YPNC.

We run ring network at 250 MHz and transfer 128 bit data per cycle. So, overall we achieve 3.7 GB/s. The round trip latency is 88 ns.

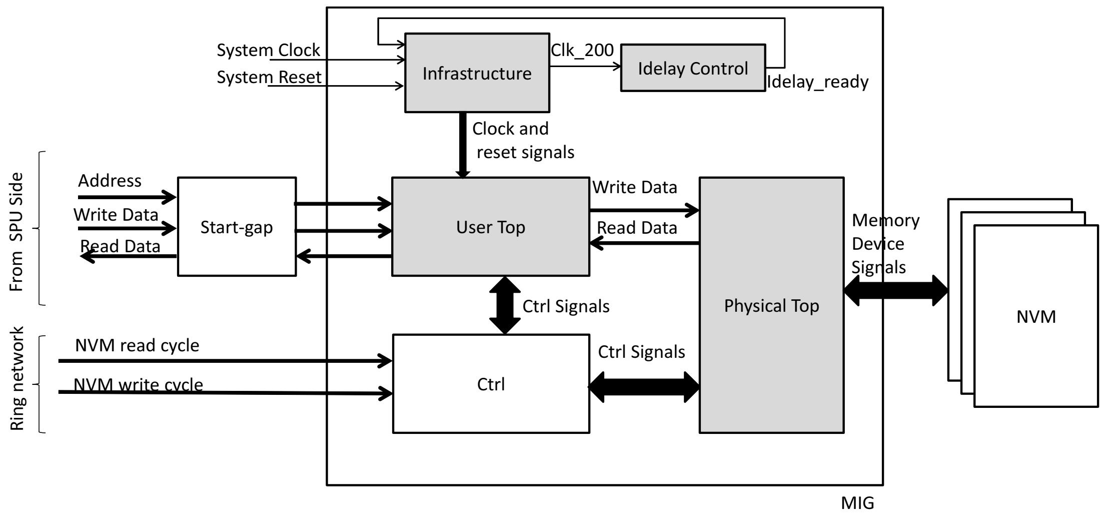

Figure 6.5: **NVM controller architecture** extends the existing Xilinx MIG DDR2 controller (grey boxes) with start-gap. The start-gap is a simple wear-leveling technique to distribute wear across the chip to increase lifetime as phase change memory suffers from endurance issues. We enhance the control logic to emulate any NVM technologies by changing read and write time.

6.3 NVM Controller

NVM controller manages the data stored on NVM and contains the logic necessary to read and write to NVM. It receives read and write requests from SPU. It can fetch data either from the processor's memory or from the RX ring network buffer. Similarly, it sends data to the processor or the TX ring network buffer. It connects with dual-rank DIMM using DDR2 interface with total capacity 8 GB. It runs at 250 MHz and has peak bandwidth of 3.72 GB/s.

Figure 6.5 shows the NVM controller architecture. We extend the existing Xilinx MIG DDR2 controller [Xil] with start-gap. We enhance the control logic to emulate emerging NVM technologies. The existing MIG DDR2 controller comprises of five main components: infrastructure, idelay control, user top, physical top and ctrl. The infrastructure module has a PLL that takes system clock and reset as inputs and produces various clocks and reset signals for the memory controller. The idelay control takes 200 MHz clock signal and performs calibration for each idelay element based on request from the physical top to correctly capture

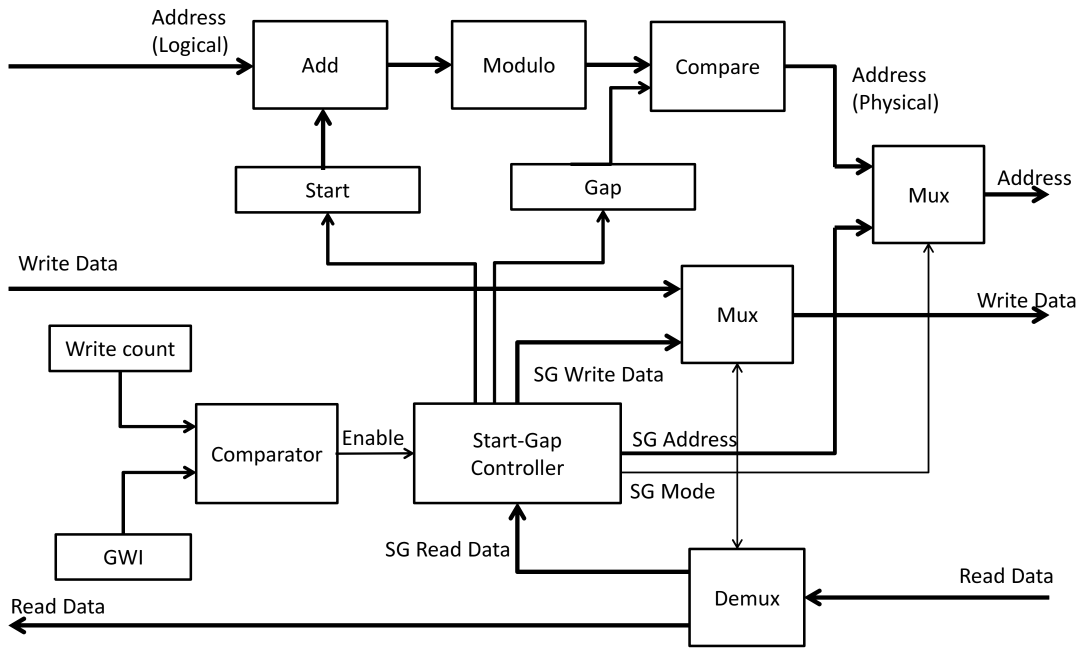

Figure 6.6: **Start-gap architecture** computes logical to physical address transformation using 3-stage pipeline logic. For every gap write interval (GWI) writes, start-gap controller initiates start-gap operation and updates start and gap registers and swaps gap line.

incoming data on the I/O pin. The user top provides user interface and receives read and write requests from the user. It has three FIFOs: address FIFO, read FIFO and write FIFO. The address FIFO holds the information about the given address and command (read or write). The write FIFO holds data. In case of write, it reads the address from the address FIFO and data from the write FIFO, and dispatches them to the physical top for writing to the memory chip. In case of read, It reads the address from the address FIFO and dispatches to the physical top to read data from the memory array and enqueue them read FIFO. The physical top comprises of various I/O blocks (IOBs), IDDR and ODDR primitives to capture data at double data rate from the memory chip. It drives various control signals to the memory chip such as RAS_N, CAS_N and WE_N from the ctrl module and performs initialization and calibration of strobes and data signals using idelay control.

We assume NVM chips will have similar internal architecture as DRAM chips and we need to consider the impact of internal row buffer for different memory accesses. So, we extend the ctrl logic to accurately emulate various NVM technologies. The ctrl produces all control signals for the DDR2 memory interface. It reads address and command from the address FIFO and generates various control signals. It performs refresh operation to preserve data in DRAM array. It supports multiple-bank mode to simultaneously open multiple banks to improve performance. We modify the existing ctrl state machine to add latency between the read address strobe (RAS) and column address strobe (CAS) commands during reads. We extend the pre-charge latency after a write by inserting delay. We configure those delay from the host over the ring network. We cannot stop DRAM refresh to preserve data which is not required for these NVM technologies.

We implement start-gap wear leveling [QKF$^+$09] to uniformly distribute wear across the chip for non-uniform write accesses as phase change memory cells suffer from limited endurance (1000,000 writes). It divides the memory array into lines where N+1 is the total number of lines (+1 for gap line) and L is the length of each line in bytes. It uses two registers. *start* and *gap*. Initially, the *start* register points to the first line of the memory array and the *gap* register points to the last line of the memory array. It uses gap write interval parameter to initiate start-gap operation. For phase change memory, we configure gap write interval $W = 100$. For every W writes, the gap line swaps data with its previous line. When gap reaches to the first line and receives W writes, then it increments start register to point to the next line and swaps gap line with the last line of the memory array. It changes logical address (LA) to physical address mapping (PA). We use the formula from [QKF$^+$09] (if $(LA + start)\%N \geq gap$ then $PA = (LA + start)\%N + 1$ else $PA = (LA + start)\%N$).

As the memory controller has very stringent timing requirement, we implement it using 3-stage pipeline logic where $S1$ and $S2$ represent intermediate registers.

- **Add** - It performs addition of LA with *start* register ($S1 = (LA + start)$).
- **Modulo** - Then it computes modulo N ($S2 = S1\%N$).

- **Compare** - Finally, It compares with *gap* register and updates PA accordingly (if $S2 \geq gap$ then $PA = S2 + 1$ else $PA = S2$).

Figure 6.6 shows the start-gap architecture. It has two modes of operations: normal and start-gap mode. During normal mode, it only performs logical address to physical address transformation. We use 3-stage pipeline logic (add, modulo and compare) for the address transformation. When the number of writes reaches to gap write interval, it triggers start-gap mode. During start-gap mode, start-gap controller issues a read and a write request of L bytes to swap the gap line with the previous line. Then it updates *gap* register to point the new gap line and switches back to the normal mode.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine*. The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories", by Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and

Steven Swanson, wwhich appears in *MICRO-43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* The dissertation author was the second 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 "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation author was the third investigator and author of this paper.

Chapter 7

Conclusion and Future Work

This thesis provides insight on what radical impacts on storage systems be if NVM technologies evolve as a potential replacement for existing storage technologies such as disks and flash. We present XSSD architecture to offload computation in storage to exploit fast NVMs and reduce the redundant data traffic across PCIe. It significantly eliminates the PCIe and software overheads, and exposes low latency and high bandwidth NVMs to end application. XSSD offers a flexible user-friendly RPC-based programming framework to implement applications on storage. We implement various data-intensive applications and achieve speedup and energy efficiency of 1.5-8.91$\times$ and 1.7-10.27$\times$ respectively.

There are several ongoing and future work directions.

Support multiple applications we need to extend XSSD such that it can support concurrent execution of multiple applications on storage to efficiently utilize fast NVMs and compute resources of XSSD. We need a lightweight scheduler to execute multiple applications on SPUs and maintain the status of different applications. We need to support demand paging for concurrent execution of multiple programs on SPU. We also need an elegant protection mechanism to ensure authenticate access of resources by different applications.

Support storage services XSSD primarily focuses on application-specific computations and in-storage data structures. It can be extended to support various storage services such as caching, transactions and permission management. It can facilitate different applications to take advantage of those services transparently.

Database and "Big Data" applications We can enhance XSSD to support various database operations such as scan, join, sort, group-by and aggregate primitives to improve performance. As "Big Data" applications are increasingly limited by the storage performance, we can build a large scale system of XSSDs that can offer adequate storage, computation and communication bandwidth to support those applications.

Acknowledgments

This chapter contains material from "Minerva: Accelerating Data Analysis in Next-Generation SSDs", by Arup De, Maya Gokhale, Rajesh K. Gupta, and Steven Swanson, which appears in *FCCM'13: Proceedings of the 21st IEEE International Symposium on Field-Programmable Custom Computing Machine.* The dissertation author was the first investigator and author of this paper. The material in this chapter is copyright ©2013 by the Institute of Electrical and Electronics Engineers (IEEE). 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 IEEE 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., IEEE, Inc., telephone +1 (732) 562-3966, or copyrights@ieee.org.

This chapter contains material from "Willow: A User-Programmable SSD", by Sundaram Bhaskaran, Trevor Bunker, Arup De, Mark Gahagan, Yanqin Jin, Robert Liu, Sudharsan Seshadri and Steven Swanson, which has been submitted for possible publication by USENIX in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, (OSDI'14).* The dissertation

author was the third investigator and author of this paper.

Bibliography

[AFK$^+$09] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[AMK$^+$10] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large cams for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 29–29, Berkeley, CA, USA, 2010. USENIX Association.

[AUS98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation, 1998.

[BC95] Rajesh Bordawekar and Alok Choudhary. Communication strategies for out-of-core programs on distributed memory machines. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 395–403, New York, NY, USA, 1995. ACM.

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

[BFH$^+$04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[BKF$^+$99] Jay B. Brockman, Peter M. Kogge, Vincent Freeh, Shannon K. Kuntz, and Thomas Sterling. Microservers: A new memory semantics for massively parallel computing, 1999.

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

[cas] The apache cassandra project. http://cassandra.apache.org/.

[CCA$^+$11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *SIGARCH Comput. Archit. News*, 39(1):105–118, March 2011.

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

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

[Chu71] L. O. Chua. Memristor-The missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507–519, September 1971.

[CMD$^+$12] Adrian M. Caulfield, Todor I. Mollov, Arup De, Joel Coburn, Rajesh K. Gupta, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '12, 2012.

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

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

[DFK$^+$92] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Letbin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE MICRO*, 12:23–39, 1992.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DH81] David J. DeWitt and Paula B. Hawthorn. A performance evaluation of data base machine architectures (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 199–214. VLDB Endowment, 1981.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[DKP$^+$13] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010.

[DSL11] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 25–36, New York, NY, USA, 2011. ACM.

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

[FI09] Fusion-IO. ioxtreme: Extreme performance for extreme users, 2009. http://community.fusionio.com/media/p/463.aspx.

[FPC$^+$97] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce Mcgaughy, David Patterson, Tom Anderson, and Katherine Yelick. The energy efficiency of iram architectures. In *In the 24th Annual International Symposium on Computer Architecture*, pages 327–337, 1997.

[FRF$^+$03] Basilio B. Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas. Programming the flexram parallel intelligent memory system. *SIGPLAN Not.*, 38(10):49–60, June 2003.

[GHI95] Maya Gokhale, William Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *IEEE Computer*, 28(4):23–31, 1995.

[GIS10] Xiaochen Guo, Engin Ipek, and Tolga Soyata. Resistive computation: Avoiding the power wall with low-leakage, stt-mram based computing. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 371–382, New York, NY, USA, 2010. ACM.

[HKK+99] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, and Jeff Lacoss. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *In Supercomputing*, 1999.

[HKM+08] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 41–50, New York, NY, USA, 2008. ACM.

[Hof05] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[Hyb] http://hybridmemorycube.org.

[Int] Intel. Intel xeon processor e5-2690 datasheet.

[Int08] Intel. Intel system controller hub datasheet, 2008. http://download.intel.com/design/chipsets/embedded/datashts/319537.pdf.

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

[JEZ+05] Philip Jacob, Okan Erdogan, Aamir Zia, Paul M. Belemjian, Russell P. Kraft, and John F. McDonald. Predicting the performance of a 3d processor-memory chip stack. *IEEE Design & Test of Computers*, 22(6):540–547, 2005.

[JLFA14] Sang-Woo Jun, Ming Liu, Kermin Elliott Fleming, and Arvind. Scalable multi-access flash store for big data analytics. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '14, New York, NY, USA, 2014. ACM.

[KBC+05] T. Kolda, D. Brown, J. Corones, J Critchlow, T. Eliassi-Rad, L. Getoor, B. Hendrickson, V. Kumar, D. Lambert, C. Matarazzo, K. McCurley, M. Merrill, N. Samatova, D. Speck, R. Srikant,

J. Thomas, M. Wertheimer, and P.C Wong. Data sciences technology for homeland security information management and knowledge discovery. 2005.

[KBCcL03] Sriram Krishnamoorthy, Gerald Baumgartner, Daniel Cociorva, and Chi chung Lam. On efficient out-of-core matrix transposition. Technical report, 2003.

[KHH$^+$99] Yi Kang, Michael (Wei) Huang, Wei Huang, Seung moon Yoo, Zhenzhou Ge, Vinh Lam, Diana Keen, Zhenzhou Ce, Vinh Lain, Pratap Pattnaik, and Josep Torrellas. Flexram: Toward an advanced intelligent memory system. In *In International Conference on Computer Design*, page pages, 1999.

[Khr08] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[KK09] M.H. Kryder and Chang Soo Kim. After hard drives - what comes next? *Magnetics, IEEE Transactions on*, 45(10):3406–3413, Oct 2009.

[Kog94] Peter M. Kogge. Execube-a new architecture for scaleable mpps. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, pages 77–84, Washington, DC, USA, 1994. IEEE Computer Society.

[KOP$^+$11] S. Kim, H. Oh, C. Park, S. Cho, and S-W. Lee. Fast, energy efficient scan inside flash memory ssds. In *Proceedings of ADMS 2011*, 2011.

[KP02] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *In Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 283–293, 2002.

[KPH98] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27:42–52, September 1998.

[KsKMP13] Yangwook Kang, Yang suk Kee, E.L. Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12, 2013.

[LAS$^+$09] 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.

[LFAK11] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[LGBT05] Christianto C. Liu, Ilya Ganusov, Martin Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gapwith 3d ic technology. *IEEE Des. Test*, 22(6):556–564, November 2005.

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

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[MAC$^+$11] John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[MAS$^+$06] Shashidhar Mysore, Banit Agrawal, Navin Srivastava, Sheng-Chih Lin, Kaustav Banerjee, and Tim Sherwood. Introspective 3d chips. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 264–273, New York, NY, USA, 2006. ACM.

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

[memb] An (incomplete) survey of new memory technologies. www.hpcresearch.nl/talks/pr_ws_10.pdf.

[NBGS08] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[OBS] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db.

[OCS98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active pages: A computation model for intelligent memory. In *IN INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE*, pages 192–203, 1998.

[OCS99] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Activeos: Virtualizing intelligent memory. *Computer Design, International Conference on*, 0:202, 1999.

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

[PAC$^+$97] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17:34–44, March 1997.

[PCM] http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.

[PHM10] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Journal of Algorithms*, page 2004, 2001.

[QKF$^+$09] 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.

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

[RGF98] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag.

[Vir10] Virident. Virident tachion pcie ssd, 2010.

[vol] The voldemort project. http://project-voldemort.com/.

[VSG$^+$10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 205–218, New York, NY, USA, 2010. ACM.

[VTRC11] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[WCC$^+$07] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[Wil08] R. Williams. How we found the missing memristor. *IEEE Spectr.*, 45(12):28–35, December 2008.

[Xbo] http://www.xboxmp.com/.

[Xil] Memory interface generator (mig). http://www.xilinx.com/products/intellectual-property/MIG.htm.

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.