Micro-Architecture and Systems Support for Emerging Non-Volatile Memories

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

in

Computer Science (Computer Engineering)

by

Meenakshi Sundaram Bhaskaran

Committee in charge:

Professor Steven Swanson, Chair  
Professor Chung Kuan Cheng  
Professor Ryan Kastner  
Professor Paul Siegel  
Professor Michael Taylor

2016
Copyright
Meenakshi Sundaram Bhaskaran, 2016
All rights reserved.
The dissertation of Meenakshi Sundaram Bhaskaran is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2016
DEDICATION

To my mother, father, and guru.
EPIGRAPH

The most enduring sort of power is realizing the power within.

Sadhguru, Isha Foundation
TABLE OF CONTENTS

Signature Page ................................................................. iii
Dedication ................................................................. iv
Epigraph ...................................................... v
Table of Contents .................................................. vi
List of Figures ....................................................... viii
List of Tables ..................................................... x
Acknowledgements ............................................ xi
Vita ............................................................... xiii
Abstract of the Dissertation ..................................... xiv

Chapter 1 Introduction ............................................. 1

Chapter 2 Background and Motivation ........................................ 6
  2.1 Technology Trends ........................................ 7
  2.2 Storage and Memory Hierarchy ........................................... 9
    2.2.1 PCIe-Attached storage ........................................ 9
    2.2.2 Hybrid-Memory Systems ......................................... 15
  2.3 Summary ..................................................... 18

Chapter 3 Bankshot — A Caching System ................................. 20
  3.1 System Overview ........................................... 22
    3.1.1 Bankshot design ........................................ 23
    3.1.2 Bankshot in action ........................................ 24
    3.1.3 Bankshot Hit Latency ......................................... 25
  3.2 System Architecture ........................................ 26
    3.2.1 The Bankshot SSD ......................................... 26
    3.2.2 LibBankshot ............................................... 28
    3.2.3 The Bankshot kernel module ................................ 28
  3.3 Design Options ......................................... 31
    3.3.1 Evaluation setup ......................................... 31
    3.3.2 Tracking cache contents ................................... 32
    3.3.3 Replacement policies ....................................... 33
    3.3.4 Tracking dirty data ....................................... 36
    3.3.5 Asynchronous write-back ................................... 41
| Figure 2.1. | PCIe SSD Interface using NVMExpress standard | 11 |
| Figure 2.2. | System stack for SSD based caches | 12 |
| Figure 2.3. | Breakdown of FlashCache hit latency | 12 |
| Figure 2.4. | PCIe overheads in accessing NVM and Hybrid Memory Architecture | 15 |
| Figure 3.1. | Bankshot system stack | 22 |
| Figure 3.2. | Architecture of the Bankshot SSD | 26 |
| Figure 3.3. | Permission management in Bankshot | 29 |
| Figure 3.4. | Bankshot read latency | 34 |
| Figure 3.5. | Bankshot write latency | 34 |
| Figure 3.6. | Bankshot 50% read / 50% write latency | 35 |
| Figure 3.7. | Breakdown of cache hit latency | 37 |
| Figure 3.8. | Bankshot cache content tracking | 39 |
| Figure 3.9. | Bankshot asynchronous write-back with software dirty tracking | 39 |
| Figure 3.10. | Bankshot asynchronous write-back with hardware dirty tracking | 40 |
| Figure 3.11. | Bankshot asynchronous write-back with hybrid dirty tracking | 40 |
| Figure 3.12. | Bankshot Hardware dirty bit tracking state machine and efficiency of asynchronous write-back schemes | 43 |
| Figure 3.13. | Bankshot application performance | 44 |
| Figure 4.1. | Willow high level SSD architecture | 49 |
| Figure 4.2. | The anatomy of an SSD App | 51 |
| Figure 4.3. | READ() implementation for Base-IO | 56 |
| Figure 4.4. | The Willow prototype architecture | 59 |
| Figure 4.5. | The Willow bridge network | 60 |
Figure 4.6. Latency break down of cache hit latency for Bankshot and Caching 63

Figure 4.7. Read latency vs. working set size for Caching, Bankshot and FlashCache 64

Figure 4.8. Write latency vs. working set size for Caching, Bankshot and FlashCache 64

Figure 4.9. RPCs to implement APPEND() 65

Figure 4.10. Performance of file append in Willow 65

Figure 5.1. Zynq SoC based evaluation Architecture for NBLD 82

Figure 5.2. Zynq SoC based software stack for NBLD 82

Figure 5.3. Latency Breakdown of 32 B fetch via the ACP port 86

Figure 5.4. Runtime for baseline, software-prefetch and NBLD 87

Figure 5.5. Emulated system CPU stall cycles for different operation mode 90

Figure A.1. Architecture of Cache Controller 96

Figure A.2. Memory Pipeline Unit Control Interface 97

Figure A.3. Memory Pipeline Unit Data Interface 98

Figure A.4. Memory Pipeline Unit Ring Decode/Encode 98

Figure A.5. Memory Pipeline Unit Split/Merge Manager 99

Figure A.6. Memory Pipeline Unit Cache Control 100
LIST OF TABLES

Table 2.1. NVM technology trend .................................................. 7
Table 3.1. Cache Hit Bandwidth: Flashcache vs. Bankshot .................. 25
Table 3.2. Bankshot Design Options ............................................ 31
Table 3.3. Bankshot Application Performance .................................. 44
Table 4.1. Implementing Willow Apps: Development time and effort for SSD Apps ................................................................. 59
Table 5.1. NBLD instruction set execution model ............................... 79
Table 5.2. Zynq SoC features and peripherals ................................. 81
Table 5.3. FPGA resource utilization for Zynq SoC based architecture ...... 86
ACKNOWLEDGEMENTS

I am grateful for all the support and guidance from many wonderful people over the years and making this journey possible for me.

Foremost, I would like to thank my advisor Professor Steven Swanson for his support and guidance. His mentorship and patience have been instrumental in molding, not just my ability to do research but also, my presentation and communication skills. The internship opportunities that became possible because of his work have been invaluable experiences throughout my Ph.D.

I would like to thank rest of my committee members for their valuable time and feedback on my research. I would like to acknowledge my colleagues in the Non-Volatile Systems Lab (NVSL) whose hard work is an integral part of my Ph.D. thesis. Special thanks to Jian Xu for his support on Bankshot and Sudharsan Seshadri, Arup De, Trevor Bunker, Yanqin Jin, Robert Y. Liu and Mark Gahagan without whom Willow would not have been possible.

Thanks also to David Roberts and Elliot Mednick from AMD Inc. for their guidance and support with Asynchronous Memory Access. I wish to also thank all my mentors, friends and colleagues from UC San Diego, Samsung and AMD Inc.

I would like to thank Ashish Venkat, Siva Nishok and Kadappan Panayappan and all my friends.

Finally, I am grateful to my parents and my sister for their love, patience, and support. Also, to my niece, Mythreyi, whose mischiefs were a constant source of joy and laughter. Thank you!

Chapter 1 and 2 and 3, in part, contains material from “Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW ’13). ACM, New
York, NY, USA and “Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in ACM SIGOPS Operating Systems Review, Volume 48, Issue 1 May 2014. The dissertation author was the primary investigator and author of these papers.

Chapter 1 and 2 and 4, contains material from “Willow: a user-programmable SSD”, by Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, which appears in In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI’14). USENIX Association, Berkeley, CA, USA, 67-80. The dissertation author was the third investigator and author of this paper.

The material presented in Chapter 1,2,3 and 4 are copyright © 2016 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 permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481.

Chapter 1 and 2 and 5, contains material as it currently being prepared for submission for publication. “Instruction Set and Micro-architecture Supporting Asynchronous Memory Access”, by Meenakshi Sundaram Bhaskaran, David Roberts, Elliot Mednick and Steven Swanson. The dissertation author was the primary investigator and author of this material.
VITA

2007 B.E, Electrical and Electronics Engineering, College of Engineering, Guindy, Chennai, India

2007 - 2010 Product Application Engineer, SanDisk Corp., Bangalore, India

2011 Research Assistant, San Diego Super Computer Center

2011 - 2016 Research Assistant, University of California, San Diego

2011 Internship, Samsung, San Jose

2014 Master of Science, University of California, San Diego

2014 Candidate of Philosophy, University of California, San Diego

2015 Internship, AMD Inc., Boston

2016 Doctor of Philosophy, University of California, San Diego

PUBLICATIONS


Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. 2014.“Bankshot: Caching slow storage in fast non-volatile memory”. In 1st Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW13.
EMERGING non-volatile memory technologies such as phase-change memory, resistive random access memory, spin-torque transfer memory and 3D XPoint memory promise to significantly increase the I/O sub-system performance. But, current disk-centric systems fall short in taking advantage of the bandwidth and latency characteristics of such memories. This dissertation presents three systems that address: hardware, system software and micro-architecture support for faster-than-flash non-volatile memories.

First, we explore system design for using emerging non-volatile memories (NVM) as a persistent cache that bridges the price and density gap between NVMs and denser
storage. Bankshot is a prototype PCIe-based intelligent cache with access latencies an order of magnitude lower than conventional SSDs. Unlike previous designs of SSD caches, Bankshot relies on the OS for heavyweight operations such as servicing misses and write-backs while allows cache hits to bypass the operating system (OS) and its associated software overhead entirely.

Second, we extend the ability to define application specific interface to emerging NVM SSDs such that a broad range of applications can benefit from low-latency, high-bandwidth access to the SSD’s data. Our prototype system, called Willow, supports concurrent execution of an application and trusted code within the SSD without compromising on file system protections. We present three SSD apps - Caching, Append and zero-out that showcase Willows capabilities. Caching extends Willows semantics to use the SSD storage as a persistent cache while file-append and zero-out extends the semantics for file system operations.

Finally, we address the challenge of accessing byte-addressable, emerging NVMs with higher than DRAM latency when attached to the processor memory bus; specifically for loads. We propose Non-Blocking Load (NBLD), an instruction set extension to mitigate pipeline stalls from long-latency memory accesses. NBLD is a non-blocking instruction that brings data into the upper levels of the cache hierarchy, however, unlike prefetch instructions, NBLD triggers the execution of application-specific code once data is resident in the cache, effectively hiding the latency of the memory.
Chapter 1

Introduction

Until the last decade, the performance of microprocessors has grown exponentially while the performance of persistent store (disk) lagged behind. Advancement in disk technology doubled storage capacity every 1-1.5 years but the access latency and bandwidth of the disk has only improved by 15x and 322x respectively [85, 35] in the last three decades.

New memory technology such as NAND flash provides two orders of magnitude better performance than disk. Even faster-than-flash non-volatile memories are starting to become commercially available. Emerging dense, non-volatile memory such as Phase-change memory (PCM), 3D XPoint, and Resistive RAM (ReRAM) have the potential to vastly improve storage subsystem performance and will eventually play important roles at many levels of the storage hierarchy. This change to the memory hierarchy will also enable exciting new applications and improve the performance and efficiency of memory intensive programs.

Despite their promise to improve performance, energy efficiency and capacity, these memories also present several challenges that system designers and processor architects must address. We evaluate a combination of hardware and software support to exploit the latency and bandwidth benefits of these memories.
One particularly attractive application for these memories is as caches within larger storage systems comprised of hard drives and/or flash-based solid state drives. At least for the time being, emerging non-volatile memories are expensive and less dense compared to NAND Flash and magnetic disk. This density and price gap makes such NVMs especially suitable as a cache for larger, more conventional storage. However, existing caching architectures, even those that use flash-based SSDs, introduce significant software overhead that can obscure the performance benefits of faster memories.

In Chapter 3 we propose Bankshot a PCIe-attached block-based write back caching system that reduces the software overheads for cache hits. Bankshot allows unmodified applications to have cache hits serviced directly from user-space by a combination of hardware and software support. Bankshot achieves this by first augmenting the SSD interface with cache specific functions, second by providing a cache manager within the operating system to handle cache misses and finally a user space library to transparently intercept applications POSIX calls.

Bankshot design allows secure user-space access to data while enforcing file system permissions in the hardware. We evaluate several design options including different cache management policies and different levels of hardware and software support for tracking dirty data and maintaining cache metadata. We find that with hardware support Bankshot improves the cache hit latency for 4 kB reads by $1.5 \times$, improves writes by $4 \times$, and improves Berkeley-DB throughput by nearly $2.9 \times$ when compared to conventional SSD-based caching systems.

In order to leverage NVM latency and bandwidth Bankshot relies on specialised SSD interface and implementing the hardware SSD prototype in Verilog took us nearly one year of graduate student time. Emerging NVM based SSDs offer the potential to improve performance for a broad class of applications. This one-at-at-time approach to specialize SSD interface limits the new class of applications from leveraging SSDs’
capabilities. To address this drawback, in Chapter 4 we explore the potential of making programmability a central feature of the SSD interface.

Willow, allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. The SSD Apps running on Willow give applications low-latency, high-bandwidth access to the SSD’s contents while reducing the load that IO processing places on the host processor. We demonstrate the effectiveness and flexibility of Willow by implementing three SSD Apps and measuring their performance and development time.

Besides showcasing Willows flexibility with different classes of application, in this dissertation we also compare Willows performance to fixed-function implementation. First, we present two SSD apps called Zero-out and Append that extend the SSD semantics to support file system specific operations within the storage device. Willows flexibility allowed us to build Zero-out and Append within 6 weeks and improve file system performance by nearly $5 \times$ compared to conventional block device interface. Second, we compare Willows performance with Bankshot for caching application. Willow offers comparable performance to its equivalent hardware only implementation and defining SSD semantics in software drastically reduces the time required to implement new functions while reducing performance by just a small margin.

Willow and Bankshot both address the software and hardware challenges associated with PCIe-attached devices. However, emerging NVMs provide byte addressable access to memory and can potentially be placed on the processor memory bus and accessed via load/store instructions. This change in memory hierarchy allows architects to build systems with terabytes of main memory.

But, as the capacity of NVM increases the access latency also increases [89] which will reduce processor performance due to long-latency cache misses and the resulting pipeline stalls. In Chapter 5 we present Non-Blocking Load (NBLD) to reduce
the impact of long-latency loads.

NBLD is an instruction set extension that provides asynchronous, non-blocking memory access. The NBLD instruction initiates a memory fetch and schedule the execution of application-defined call-back that will execute once the data has arrived in the data cache.

We implemented NBLD on a hybrid memory emulation platform using the Xilinx Zynq SoC and evaluate our design for two graph workloads: PageRank and alternating least squares. We find that NBLD improves application runtime by 40% while reducing the CPU stall by 10%.

We structure the dissertation as follows: In Chapter 2 we highlight the trends in memory technologies and provide background on system architectures based on these memories. We motivate the case for hardware and software support for emerging NVM and also survey previous work on such systems. Chapter 3, 4, 5 presents the three different systems introduced above and Chapter 6 provides the summary.

Acknowledgements

This chapter contains material from “Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW ’13). ACM, New York, NY, USA. And ‘Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in ACM SIGOPS Operating Systems Review, Volume 48, Issue 1 May 2014. The dissertation author was the first investigator and author of this paper.

This chapter contains material from “Willow: a user-programmable SSD”, by Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De,
Yanqin Jin, Yang Liu, and Steven Swanson, which appears in In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI’14). USENIX Association, Berkeley, CA, USA, 67-80. The dissertation author was the third investigator and author of this paper.

This chapter contains material from “Instruction Set and Micro-architecture Supporting Asynchronous Memory Access”, by Meenakshi Sundaram Bhaskaran, David Roberts, Elliot Mednick and Steven Swanson, which is currently being prepared for submission. The dissertation author was the first investigator and author of this paper.
Chapter 2

Emerging Memory Technologies: Background and Motivation

The growth in internet led to the proliferation of social networks, e-commerce and web searches which create large volumes of digital information. For decades, computer systems have relied on disk to store and access the data. While other system components have gotten faster and more flexible hard disk performance has remained stubbornly poor. This enabled system designers to hide the overheads of software and hardware overheads associated with disk-centric optimizations. However, the impending arrival of phase change memory (PCM), spin-torque (STT-RAM), and resistive (ReRAM) memories is transforming the way computer scientists think about memory hierarchies, processor architectures, storage systems, operating systems and applications. In this chapter, we provide background on the trends in memory technologies - their bandwidth, and latency characteristics and highlight the novel challenges to system designers and architects in embracing these new memories. The chapter also provides an overview of conventional DRAM and disk-centric hardware/software stack.
Table 2.1. **NVM technology trend**: Properties of non-volatile storage technology

<table>
<thead>
<tr>
<th>Technology</th>
<th>Density ($F^2$)</th>
<th>Latency (ns)</th>
<th>Endurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic RAM (DRAM)</td>
<td>6-12</td>
<td>10</td>
<td>&gt; $10^{16}$</td>
</tr>
<tr>
<td>Hard Disk Drives</td>
<td>2-3</td>
<td>5-8×$10^6$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>NAND Flash</td>
<td>4</td>
<td>$10^4 - 10^5$</td>
<td>$10^3$</td>
</tr>
<tr>
<td>Phase Change Memory (PCM)</td>
<td>4-16</td>
<td>10-150</td>
<td>100-1000</td>
</tr>
<tr>
<td>Resistive RAM (ReRAM)</td>
<td>6-10</td>
<td>10-1000</td>
<td>10-1000</td>
</tr>
<tr>
<td>Spin Torque Transfer RAM (STT-RAM)</td>
<td>6-100</td>
<td>6-35</td>
<td>6-90</td>
</tr>
</tbody>
</table>

### 2.1 Technology Trends

Fast, scalable non-volatile main memory (NVMM) technologies that promise byte addressable access to storage will soon be commercially available [27, 68, 31, 16, 99, 97]. Various research groups have built complete prototype memory devices of PCM, STT-RAM and ReRAM which allows us to understand their characteristics and their implications for system designers.

Table 2.1 summarizes the latency, density and endurance characteristics of some of the most promising memory technologies based on a recent survey [89]. We explore the characteristics of these memories with DRAM as a reference point. DRAM which has nearly infinite endurance, has almost equal read/write access latencies. However, DRAM suffers from CMOS scaling challenges and also cannot persist data on a power loss. In order to persist data, until the last decade systems primarily used hard-disk drives.

Hard-disk drives have not changed much in their performance compared to other system components. They provide high storage density, but owing to the mechanical movement of arms needed to read or write data, the read/write performance of disk drives is several orders of magnitude lower compared to DRAM.

In contrast, emerging non-volatile memories promise to be available in both DRAM like the form factor and as solid-state drives. Of all NVM technology NAND-
Flash is one of the most commonly used technology which continues to proliferate embedded systems to data centers. High-speed PCIe flash SSDs are available commercially and they provide almost two orders of magnitude better latency and bandwidth compared to disk. Recently manufacturers have been able to shrink the process geometries and add multiple bits to a NAND memory cell pushing for high-density devices. However, management of NAND flash requires a flash translation layer (FTL) in order to hide the idiosyncrasies resulting from asymmetric read/write times and out of place updates.

Unlike NAND-Flash, Phase-change memories store bits not as charges but by changing the resistive state of the material [10]. PCM uses a chalcogenide-metal layer that changes state when applied heat and cooled rapidly (amorphous state) or slowly (crystalline state). PCM allows in-place updates and has the potential to replace DRAM owing to its byte-addressable nature and low read/write latency. While PCM endurance is lower compared to DRAM, recent work [78] has shown that PCM memory technology does not require complicated FTL for management. Recently, Intel and Micron announced 3D XPoint memory [2], which promises byte-addressable memory at 1000 times faster than NAND flash memory. 3D XPoint [60] is primarily Phase Change Memory with a switch that is expected to be available in DIMM form factors and as PCIe-SSD drives.

Other promising memory technologies include - spin-torque transfer memory (STT-RAM) and Resistive Memory (ReRAM). STT-RAM [68] uses a magnetic tunnel junction (MJT) to store bits. The orientation (parallel or anti-parallel state) of the magnetic layer within the MJT determines the junction resistance and the value of the bit stored. On the other hand, ReRAM [31] uses a material that changes its resistance based on the direction of current flow through the device. The passive element then remembers the resistance even after the flow of current stops thereby storing information. Both these technologies provide similar benefits as PCM with improved latency and bandwidth at
potentially higher endurance.

All these memories will soon be commercially available. In the next section, we present the conventional storage stack and how attaching NVM to memory/IO hierarchy presents novel challenges for system designers.

2.2 Storage and Memory Hierarchy

While emerging memory technologies have the potential to replace not just hard-disk drives but even DRAM as main memory, there are tradeoffs in terms of cost, capacity, latency, and endurance. A study by International Data Corporation [36], projects exponential growth in digital data, with doubling of information every two years from 2012 to 2020. Alteast for the near future, systems will continue to use both emerging NVM and disk and/or DRAM to address the storage needs of modern applications.

In this dissertation we focus on two systems architectures. First, a system where emerging NVM is attached via PCIe, a high-speed peripheral interface. Second, NVM in DIMM form factor attached to the processor memory bus. Below we discuss the opportunities and challenges for system designers in building such systems.

2.2.1 PCIe-Attached storage

The emergence of fast, non-volatile, solid-state memories has signaled the beginning of the end for painfully slow storage systems, and this demands a fundamental rethinking of the interface between storage software and the storage device. With new memory technologies, going beyond the block I/O interface provides significant performance improvement.

For decades, the performance of disk-based storage systems has made the interface between storage software and hardware largely irrelevant to overall storage system performance. Several factors have conspired to limit innovation at the interface between
storage hardware and software.

First, disks are inherently slow. Spinning disks can only deliver a few tens to a couple hundred of megabytes-per-second of bandwidth. Latency is equally poor: the fastest disks take milliseconds to perform IO operations. Performance by both these measures falls far below the abilities of modern system interconnects like SATA, SCSI, PCIe.

Second, optimizing for sequential access is necessary for good performance from a disk, so any interface that allows for efficient sequential access is “good enough.” Conventional SATA and SCSI interfaces are adequate in this regard.

Finally, disks are so much slower than processors (a modern processor can execute \( \sim \) 33 million instruction in the time it takes a disk to perform a 4 kB access), that implementing new storage interfaces in the kernel, for example, to support caching, does not add significantly to overall latency. Nor does it consume much CPU time, since the disk’s limited performance bounds the amount of processing the kernel must do. A simple block-based interface makes sense when sequential access is the only way to achieve good performance, and even those inevitably take many milliseconds to complete.

Recent work has illustrated some of the possibilities and potential benefits of redefining the storage interface. For instance, an SSD can support complex atomic operations [20, 69, 74], native caching operations [82], a large, sparse storage address space [41], delegating storage allocation decisions to the SSD [101], and offloading file system permission checks to hardware [13].

Modern, NAND-flash based SSDs that attach directly to the PCIe bus are commercially available. With the standardization of NVM Express [34], a new software interface that is optimized for next generation SSDs, high-bandwidth and low latency access to storage requires redefining the hardware/software interfaces. NVMExpress design primarily focuses on reducing the access latency, but also allows designers to
create new interfaces independent of the physical implementation of the NVM storage. Figure 2.1 depicts a conventional storage system with a high-end, PCIe-attached SSD. A host system connects to the SSD via NVMeExpress [95] over PCIe, and the operating system sends commands and receives responses over that communication channel. The commands are all storage-specific (e.g., read or write a block) and there is a point-to-point connection between the host operating system and the storage device. Below we describe two use cases for emerging NVMs attached via PCIe interface and the need for hardware/software optimization to take full advantage their performance. Below, we look at one such optimization for SSD interface for caching purposes.

**Block Caching**

With the rapid growth of data, storing all this information in emerging memory technologies that promise improvement in persistent storage performance is not the economically viable solution (at least for the time being). One particularly attractive application for these memories is as caches or tiering layers within larger storage systems.
Figure 2.2. System stack for SSD based caches: The system stack for conventional SSD caching system where the file system and operating system set policy for protection and sharing of data.

Figure 2.3. Breakdown of hit latency: Software and hardware contributions to hit latencies for a 4 kB Flashcache Write (a) and Flashcache Read (b) is shown.
comprised of hard drives and/or flash-based solid state drives (SSDs).

Several groups have proposed systems that use flash-based SSDs as caches for conventional disks, either by implementing caching-specific SSDs [82, 83] or by providing generic OS support for caching the contents of one block device on another [11, 75, 45, 87, 56, 6, 15, 49]. Replacing flash-based SSDs using emerging NVM-based SSDs [12, 5, 96] provides an economically feasible option to dramatically improve performance by integrating fast NVMs into storage systems. However, existing caching architectures, even those that use flash-based SSDs, introduce significant software overhead that can obscure the performance benefits of faster NVMs. Below, we break down the overheads associated with conventional flash-based SSD caches.

Existing SSD caching systems use an OS-resident cache manager (Figure 2.2) that exposes a block device interface while caching the contents of a conventional block device called the backing store, typically a hard disk or SSD. The cache manager tracks dirty data and maintains a mapping between cache locations and the backing store in order to transparently satisfy requests for frequently accessed data.

This approach leverages existing support in the kernel for composable block devices, but it also introduces extra software overhead and requires an operating system interaction on every access to either the cache or the backing store. For next-generation PCM- or ReRAM-based SSDs that provide sub-10µs latencies [13, 96, 46, 93] the software overhead will dominate the cache performance. This overhead is especially counter-productive when it comes to cache hits since those are the accesses that these caches would speed up. By comparison, the impact of software performance on cache misses is much smaller, since those require accesses to slow flash-based SSDs or disks.

Figure 2.3 measures the latency overhead for servicing a 4 kB cache hit for a PCIe SSD that uses PCM. For high-end SATA SSDs, with typical latencies of 200 µs for reads and 1 ms for writes, the software overhead only increases the cache hit latency by
3%. However, for a PCM SSD, the software overhead increases the hit latency by 36% and 76% for reads and writes. With this in mind, we present Bankshot a caching system whose primary design goal is to cache data while minimizing hit latency.

**Software defined interface**

Based on previous work in building high-performance solid-state drives [12, 14, 20, 13] and optimizing the software that interfaces to them, large performance gains are attainable if we can change the behavior of the storage device to match the needs of the applications. The performance gains for Bankshot from hardware/software interface modifications provided were large, but building hardware to support rich behaviors in SSDs is problematic for two reasons. First, implementation costs were high, since building and debugging hardware, even in FPGAs, is time-consuming. Second, the hardware we built for Bankshot systems had limited flexibility. Together, these two limitations drastically reduce the range of applications that can benefit from application-specific functions in an SSD.

Even though SSD controllers already incorporate significant programmability in terms of on-device processors SSD designers can only afford to add features that a large user base can make use of and must target large, profitable application. Further, one requires detailed knowledge of the SSD’s architecture and have privileged access to the SSD hardware to develop interface extensions. These make it impossible for anyone but the SSD manufacturer to add new features to an SSD or change the way the SSD behaves.

We propose to replace inflexible, one-off hardware extensions with a more generic programming model and interface to the SSD. The resulting *Software-defined SSD (Willow)* will allow us to implement all of the above schemes in software, but will also allow application developers to build the customized “SSD apps” for their applications.
Figure 2.4. PCIe overheads in accessing NVM and Hybrid Memory Architecture: (a) shows the overheads measured using Moneta-D [13] a PCIe SSD that provides userspace access and (b) shows emerging memory technologies such as PCM and ReRAM augment DRAM main memory.

2.2.2 Hybrid-Memory Systems

While Bankshot and Willow provided low software overheads in accessing emerging NVM, the hardware overheads associated with PCIe-based access overshadows the benefits of byte-addressable sub-μs NVM. Figure 2.4(a) breakdowns the software and hardware components associated with a 4 kB user-space read from Bankshot like PCIe SSD. In the near future, we will see systems whose main memory systems consist of emerging NVM and DRAM attached to the processor memory bus. This allows systems to exploit byte-addressable, in-place update property of NVMs that offer close to DRAM latency (PCM, ReRAM). Figure 2.4(b) shows such a Hybrid memory system. These systems that offer higher node density and lower energy are a good fit for data center applications which require large memory. In this section, we describe microarchitecture and instruction set challenges associated with placing NVM on processor memory bus and explores some previous attempts to address them. One of the major challenges
with NVM is that the capacity scaling of NVM comes at the cost of increased access latency for read/write operations [89, 19, 47, 97, 99] compared to DRAM. Read and write latencies are projected to range from 38 ns to 2 $\mu$s and 40 ns to 100 $\mu$s, respectively. In addition, the asymmetric read and write latency characteristics of NVM can cause the reads to be queued behind a write operation introducing large variations.

Existing processor pipelines cannot handle memory accesses at these latencies without significant performance degradation. In large part, this is because typical applications do not have enough instruction- or memory-level parallelism to hide latencies of that magnitude. Furthermore, although these latencies are too long for the pipelines to effectively hide, they are too short for other techniques for dealing with latency (e.g., spinning in software or suspending the thread in the OS) to be viable.

Architects have proposed solutions to improve the latency of performance critical reads [98, 77, 76] but continue to rely on conventional loads to perform the read operation. This increase in latency can result in cache misses that can stall the load operation for thousands of clock cycle degrading the utilization and performance of the system significantly.

Clapp et.al. [18] quantify the impact of memory latency and bandwidth on big-data, HPC and enterprise workloads and find that increased load latency affects the instruction throughput and cycles per instruction (CPI). Further, a recent study of large scale graph applications on non-volatile main memory [59] determined that increased access latency of NVM can reduce system performance by 4x compared to a system with only DRAM. The study finds aggressive speculation and out-order execution of modern processors along with cache prefetching mechanisms are not sufficient to hide the access latencies for random dependent reads. There is a need for micro-architectural and instruction set support to take advantage of the density and energy propositions offered by the new memory technologies.
ISA and Micro-Architecture for New Memories

Much of the recent work on processor support for NVM [102, 38, 65, 72] has focused on addressing write characteristics of NVM. But applications continue to rely on micro-architecture features such as prefetching, speculation, out of order execution and/or ISA primitives such as software prefetch instructions to address load operation latency.

To the best of our knowledge, there has been no previous work that optimizes the micro-architecture for high latency local memory load accesses through asynchronous function calls. Informing memory operations [32] and Decoupling Loads [33] are the closest works. Informing memory operations sets a condition bit or invokes a specific code on a cache miss (or when the data is not readily available). However, NBLD differs from ‘Informed’ operations by executing a specific code when the data is available. Decoupled Loads defines two new ‘nano’ instruction for performing load operations - load.D$ and load.wb. load.D$ performs the data access and load.wb writes the data into an architectural register for program use. While NBLD performs the data read operation similar to load.D$ instruction, the register write-back and function execution are asynchronous.

Intel recently introduced support for persistent main memory with PCOMMIT, CLWB and CLFLUSHOPT [38], also researchers [22, 72] have proposed architecture support to address the challenges with memory persistency. All these solutions address the semantics of store operation to persistent main memory while NBLD targets performance critical load operation.

We present Non-Blocking Load (NBLD), an instruction set extension to mitigate the effects of for long-latency memory accesses. NBLD is a non-blocking instruction that brings data into the upper levels of the cache hierarchy, however, unlike prefetch instructions, NBLD triggers the execution of application-specific code once data is
resident in the cache, effectively hiding the latency of the memory.

### 2.3 Summary

Storage performance has picked steam with the proliferation of NAND flash SSDs. In this chapter, we looked at the emergence of new memory technologies such as PCM, ReRAM, STT-RAM that provide byte addressable access to persistent memory at near DRAM latency and how it will find use in different levels of the memory/storage hierarchy. Minimizing software overheads, optimizing storage interface and redefining micro-architectures are necessary for system designers to exploit the full potential of NVM technologies. In the following chapters, we describe three systems: Bankshot, Willow and Non-Blocking Load that addresses the novel challenges arising from NVM.

### Acknowledgements

This chapter contains material from “Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW ’13). ACM, New York, NY, USA. And ‘Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in ACM SIGOPS Operating Systems Review, Volume 48, Issue 1 May 2014. The dissertation author was the first investigator and author of this paper.

This chapter contains material from “Willow: a user-programmable SSD”, by Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, which appears in In Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI’14).
USENIX Association, Berkeley, CA, USA, 67-80. The dissertation author was the third investigator and author of this paper.

This chapter contains material from “Instruction Set and Micro-architecture Supporting Asynchronous Memory Access”, by Meenakshi Sundaram Bhaskaran, David Roberts, Elliot Mednick and Steven Swanson, which is currently being prepared for submission. The dissertation author was the first investigator and author of this paper.
Chapter 3

Bankshot — A Caching System

As discussed in the previous chapter, fast NVMs will eventually play important roles at many levels of the storage hierarchy. They hold particular promise as a caching layer in applications where data sizes are too large to economically fit in the NVM itself. In this scenario, a smaller NVM-based storage device caches the data on a larger, conventional device (e.g., flash-based SSDs or hard disks).

However as seen in Chapter 2, relying on the OS to access the cache and implement caching policy imposes large software overhead and limits the performance improvements the cache can provide. For flash-based caches, the overhead is small compared to the SSD’s access time, but for fast NVMs, the software overhead for cache hits adds significantly to total latency, dramatically reducing the benefits of caching.

In this chapter, we describe Bankshot, a caching system built to exploit the performance benefits of fast NVMs as a cache for slower, more conventional block devices like flash-based SSDs and hard disks. Bankshot minimizes cache hit latency by allowing applications to access the cache hardware—the Bankshot SSD—without OS intervention. The OS accesses the slower backing store only when a cache miss makes it necessary. In addition, the Bankshot SSD provides hardware support to detect cache hits and misses, allows for recovery of dirty data after power failures, collects data usage information for cache management policies, and tracks which blocks are dirty for efficient
write-back.

The result is a caching system with a unique set of properties:

- **Very fast cache hits** Since cache hits do not incur system call, OS, or file system overhead, applications benefit fully from the short latencies of next-generation NVMs.

- **Write-back caching** Tracking the mapping between cache contents and backing store locations in hardware means that Bankshot is safe to use as a write-back cache.

- **Efficient eviction** The Bankshot SSD tracks which data in the cache are dirty and provides an efficient mechanism for querying that information.

- **Data Usage Tracking** The Bankshot SSD collects data access statistics that allows the OS to implement Least Recently Used (LRU) management policies.

- **File system awareness** Bankshot caches file system extents rather than storage device blocks. This allows it to naturally prefetch contiguous regions of cached files and minimize the amount of metadata the cache must maintain. Despite this, Bankshot does not require any modifications to the file system.

We explore the design space of Bankshot by implementing caching functions (hit detection, metadata maintenance, and dirtiness tracking) in hardware, software, and a combination of the two. Our study finds that a small amount of hardware support can provide reductions in both cache access time and cache miss rate.

We compare Bankshot with Facebook’s Flashcache [87], an open source cache driver that uses SSDs as block caches. Bankshot’s hit time is 4\times faster than Flashcache. Bankshot outperforms Flashcache by 2.9\times running a Berkeley DB (BDB) benchmark.
The remainder of this chapter is organized as follows: Section 3.1 and Section 3.2 we describe the caching system and its hardware and software components. We compare different design options for Bankshot, and we present our evaluation of Bankshot in Section 3.3 and Section 3.4. Section 3.5 describes Bankshot with respect to previous work. Finally, we present our Section 3.6 summarizes this chapter.

3.1 System Overview

This section we describe the various components of Bankshot that enable it to eliminate the file system and OS overhead and allow unmodified applications have cache hits serviced directly from userspace.

Three components—the Bankshot SSD, LibBankshot, and the cache manager—work together to provide Bankshot’s low-latency caching capabilities. Figure 3.1 shows
the system architecture for Bankshot.

The *Bankshot SSD* is a specialized SSD that implements cache-specific functions. The SSD exposes a virtualized interface for direct user-space access, similar to [13]. The direct-access mechanism gives each application a *virtual channel* that allows it to issue and complete I/O commands. A hardware permission mechanism in the SSD prevents the application from accessing arbitrary data.

The second component in Bankshot is *LibBankshot*: a userspace library that transparently interposes on file access system calls. It provides a POSIX-compliant interface to the cache and can issue and complete I/O requests to the Bankshot SSD directly from the userspace without operating system intervention.

Finally, the *cache manager* is a kernel module that utilizes the Bankshot SSD as a caching device for the backing store. The cache manager handles cache space allocation, evictions, and write-backs.

### 3.1.1 Bankshot design

Three aspects of the Bankshot design set it apart from existing SSD caching systems.

- **User-space cache hits**: Bankshot provides applications access to the Bankshot SSD directly via LibBankshot so that cache hits do not incur any operating system or file system overhead.

- **Extents-based caching**: Bankshot uses file extents rather than blocks. In particular, it leverages file system protection and file layout information to safely allow direct access from user-space and to detect cache misses in hardware.

- **Intelligent hardware**: Bankshot uses hardware built into the Bankshot SSD to collect usage statistics and dirty information for cache blocks that the OS can use.
3.1.2 Bankshot in action

To illustrate how the software components and the Bankshot SSD interact, we describe Bankshot’s operation under three scenarios: a read miss, a write hit, and a read miss requiring an eviction.

A miss to a cold cache Consider a read miss that requires the system to load new data into an empty cache. First, LibBankshot intercepts a read() system call and determines whether it targets a file residing on the backing store that Bankshot caches. If it does, LibBankshot must determine whether it is a hit or a miss. LibBankshot maintains a per-process logical map of the cache’s layout for this purpose. LibBankshot uses the logical map to determine whether the targeted extent is in the cache and, if it is, where it resides.

If the access is a miss, LibBankshot issues a system call to the cache manager that allocates space for the extent in the cache. The cache manager updates the allocation information in an in-kernel physical map that contains a complete index of the cache’s contents. Then it copies the data from the backing store to the Bankshot SSD. It also writes the information that maps from the cached extent to backing store extent to the Bankshot SSD to support recovery after power failure. When the system call completes, LibBankshot reissues the request, which will now be a hit.

A hit On a write hit, the logical map tells LibBankshot where the cached extent resides, and LibBankshot issues the write directly to the Bankshot SSD. The Bankshot SSD automatically marks the modified extent as dirty. We explore the performance impact of this hardware support in Section 3.3.

An eviction and a miss If a miss occurs and the cache is full, Bankshot must evict one or more extents and, if needed, write-back modified data to the underlying storage. For efficiency reasons, Bankshot usually evicts multiple extents at once (more on this in
Table 3.1. **Cache Hit Bandwidth: Flashcache vs. Bankshot**: OS and File System bypass in Bankshot provides raw SSD performance while servicing cache hits.

<table>
<thead>
<tr>
<th></th>
<th>Flashcache (MB/s)</th>
<th>Bankshot (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>1607.12</td>
<td>1721.27</td>
</tr>
<tr>
<td>Write</td>
<td>222.90</td>
<td>1738.16</td>
</tr>
<tr>
<td>50% Read, 50% Write</td>
<td>280.07</td>
<td>2845.55</td>
</tr>
</tbody>
</table>

Section 3.2) and the kernel cache manager handles this process.

The first step is to identify “victim” extents to evict from the cache to make room. The cache manager chooses victims based on its replacement policy and revokes victims’ permissions in the Bankshot SSD. This blocks applications’ access to the extents and ensures that the cache manager will have exclusive access to the extent during the eviction.

Next, the cache manager must determine whether the extent is dirty. If it is, it must write the data back to the underlying storage. The cache manager issues a query request to the Bankshot SSD to retrieve the dirty data information for the cache blocks and writes back the dirty blocks. Finally, the cache manager loads data for the new extents into the cache sets appropriate permissions in the SSD and returns control to the application.

### 3.1.3 Bankshot Hit Latency

Figure 3.7 compares the hit latency of Bankshot with Flashcache for 4 kB I/Os (we discuss our evaluation methodology in Section 3.3). Facebook’s Flashcache [87] is an open-source cache driver that caches one block device using another.

User-space access in Bankshot reduces the software overhead for read hits by 5 µs and write hits by 30 µs compared to Flashcache. Further, by servicing hits from user-space Bankshot achieves an order of magnitude improvement in bandwidth (Table 3.1).
Figure 3.2. Architecture of the Bankshot SSD: To support cache operations, Bankshot adds hardware support to existing prototype SSD. Each cache controller manages and stores cache meta data and dirty bits. The cache LRU unit is responsible for tracking the application’s access pattern to assist the kernel with the eviction policy.

Eliminating the operating system and file system overheads is especially beneficial for writes since it reduces lock contention [50].

We explore cache eviction policies, mechanisms for identifying dirty data, and mechanisms for maintaining persistent information about cache layout metadata in Section 3.3.

3.2 System Architecture

In this section, we describe details of the hardware platform we use to implement the Bankshot SSD then present the implementation of the three components in the Bankshot stack (See Figure 3.1).

3.2.1 The Bankshot SSD

The design of the Bankshot SSD adds a small amount of caching-specific hardware to a high-performance SSD built from fast NVMs. Figure 3.2 shows the internals of
the Bankshot SSD.

The SSD consists of a set of cache management hardware modules in front of the NVM array with each of these modules implementing a high-performance NVM controller. We describe the cache-specific hardware mechanisms (including the LRU unit) below. The caching modules attach to an internal ring network that runs at 3.9 GB/s.

The host-facing interface of the Bankshot SSD implements virtualization and permission check mechanisms to support direct user-space access, similar to those in Caulfield et al. [13]. The virtualization interface module provides the host with multiple virtual hardware interfaces for the SSD. The OS allocates one interface or virtual channel to each process and sets up the channel registers in the SSD. Applications use the virtual channels to issue and complete I/O requests. The Bankshot SSD maintains a per-channel permission table describing the regions of the SSD that each application can access. If a channel tries to access data without proper permissions, the access will fail.

Bankshot uses an in-order hardware controller to ensure atomicity of I/O operation when accessed by multiple readers and writers. The SSD’s scoreboard unit dispatches all request in-order which are in turn processed in-order by the cache controller.

The remaining hardware includes the host-facing PIO and DMA interfaces. These components are responsible for accepting I/O requests, executing them, and notifying the host when they are complete. Communication with the host occurs over a PCIe 1.1×8 interface, which runs at 2 GB/s, full-duplex.

We implement the Bankshot SSD on the BEE3 prototyping platform [37]. The BEE3 provides four FPGAs which each host 16 GB of DDR2 DRAM and one PCIe interface. The design runs at 250 MHz. To emulate the performance of PCM using DRAM, we use a modified DRAM controller that allows us to set the read and write latency. We use the latencies employed by Lee et al. [52]— 48 ns and 150 ns for array reads and writes, respectively. The raw hardware can perform a 4 kB read in 8.3 µs and
a 4 kB write in 8.1 $\mu$s, and it can sustain 1.8 M random 512 B IOPs.

### 3.2.2 LibBankshot

LibBankshot transparently intercepts I/O system calls (e.g., `open()`, `close()`, `read()`, and `write()`) made by unmodified applications using LD_PRELOAD and handles cache hits in Bankshot. It also registers the application with the cache manager and invokes the cache manager to service cache misses.

LibBankshot maintains a per-process logical map which maps a file’s offset to a cache address for each inode accessed by the process. The logical map is a red-black tree and uses 80 Bytes per node. We reduce the DRAM overhead of this map by storing information as extents for data region within the files accessed by the application. It uses this map to determine whether an access is a hit or a miss without invoking the OS or the file system.

For correctness, LibBankshot must keep the logical map consistent with the contents of the Bankshot SSD as the cache manager replaces data in the cache in response to cache misses. Bankshot updates the logical map lazily upon a permission miss in the hardware. When the cache manager evicts data, it also invalidates existing permission entries in the Bankshot SSD. This forces the processes with stale logical maps to invoke the cache manager and update their data structures.

### 3.2.3 The Bankshot kernel module

The kernel module provides two functions. First, it manages the Bankshot SSD’s virtualized interface and the permission tables for each application. Second, it combines the Bankshot SSD and another block device (the backing store) into a single, reliable, cached block device. Third, it ensures consistency of data viewed by applications that do not use LibBankshot.
Figure 3.3. Permission management in Bankshot: File system maps file offsets to backing store address and the cache manager maps the backing store address to chunks. Each file maps to one or many chunks.

Permission management Bankshot uses the permission check mechanism in the SSD to allow processes to share the cache data. The cache manager is responsible for verifying process permissions via a system call and installing permission entries in the SSD.

The permission table in the Bankshot SSD can hold only 8,192 entries. If Bankshot used one permission entry per extent, the system would quickly run out of permission table entries, leading to unnecessary misses when a request to the hardware fails, even though the data is present.

To avoid this problem, Bankshot divides the cache into chunks and uses one permission entry per chunk. Our prototype implementation divides the cache into 2 MB chunks to reduce the number of permission entries in the hardware. A custom ASIC implementation of the controller will be able to support smaller chunk sizes.

Each chunk can contain multiple (not necessarily contiguous) extents from a single file, and a single file (or even a single extent) can occupy multiple chunks (as shown in Figure 3.3). Bankshot evictions occur at the granularity of chunks, so a single
eviction may need many write-back operations, one for each extent in the chunk.

**Caching**  The cache manager uses an in-memory B+Tree [21], called the *physical map* to map backing store file extents to the Bankshot SSD extents. As the kernel manages the files in 4 kB pages, the cache manager also divides the SSD into 4 kB blocks and uses a fully associative map for the management of cache contents. For cache misses, the cache manager retrieves the mapping information from file extents to backing store extents using the *fiemap* (File Extents Map) inode operation.

Bankshot caches file extents rather than disk blocks for three reasons. First, to set permissions in the hardware Bankshot must be aware of file system permissions, and modern file systems index data as extents, not blocks. Second, managing extents make it easy to prefetch file data and ensures that write-backs can result in long, sequential accesses. Third, extent-based mappings reduce the size of the maps that LibBankshot and the cache manager must maintain, improving performance and reducing memory requirements.

**Cache eviction**  On a capacity miss, the cache manager identifies a candidate chunk and evicts the data by a multi-step process. First, the cache manager sets the blocks’ eviction flag to prevent the manager from servicing request to the data in the chunk. Second, it removes the permission entry for the chunk from the SSD to prevent LibBankshot from servicing cache hits from the chunk during the eviction. Third, it writes the data to the backing store if the block is dirty. Finally, it clears the blocks’ eviction flag and sets the invalid flag. It also deletes the corresponding key from the physical map.

**Cache recovery**  In order to provide a durable write-back cache, Bankshot tracks the backing store address associated with each cache block. Bankshot uses a dedicated region of the SSD to store the backing store addresses and the cache manager uses this information to reconstruct the in-memory physical map data structure after a power
Table 3.2. Bankshot Design Options: Summary of different design options in hardware and software evaluated in Bankshot. The left column gives the label used in the figures.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tracking Cache Contents</strong></td>
<td></td>
</tr>
<tr>
<td>SM</td>
<td>Cache manager updates map explicitly</td>
</tr>
<tr>
<td>HM</td>
<td>The Bankshot SSD maintains mapping information</td>
</tr>
<tr>
<td><strong>Replacement Policy</strong></td>
<td></td>
</tr>
<tr>
<td>RR</td>
<td>Cache manager uses a software circular buffer of chunks</td>
</tr>
<tr>
<td>LRU</td>
<td>Uses a recency list in Bankshot SSD</td>
</tr>
<tr>
<td><strong>Tracking of Dirty Data</strong></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>Coarse grained dirty bit tracking in cache manager</td>
</tr>
<tr>
<td>HW</td>
<td>Per cache block dirty bit in SSD</td>
</tr>
<tr>
<td>Hybrid</td>
<td>Combination of both hardware/software scheme</td>
</tr>
<tr>
<td><strong>Asynchronous Write Back</strong></td>
<td></td>
</tr>
<tr>
<td>async-RR</td>
<td>Round robin write back policy</td>
</tr>
<tr>
<td>async-LRU</td>
<td>LRU write back policy</td>
</tr>
<tr>
<td>Elevator</td>
<td>writes closest dirty block to maximize data locality</td>
</tr>
</tbody>
</table>

failure or reboot.

3.3 Design Options

The baseline design in Section 3.2 provides basic caching functions, but there is a range of possible Bankshot design points that vary in the amount of hardware support they provide and the policies they use to manage the cache. In this section, we explore this design space by evaluating versions of Bankshot that rely on software, hardware, or a combination of the two to implement caching functions. Table 3.2 summarizes the different design options we consider.

3.3.1 Evaluation setup

To evaluate Bankshot design options we use a combination of micro-benchmarks and applications. We also compare Bankshot to Facebook’s Flashcache [87].

Flashcache uses the Linux kernel’s device mapper functionality to register an
SSD as a write-back or write-through cache for disk. Flashcache divides the cache address space into sets of 512 4 kB blocks and uses a hash table with linear probing within sets for management. Flashcache periodically scans the metadata table and cleans blocks within a set when the dirty block threshold of the set exceeds the limit. Flashcache also prioritizes eviction of clean blocks over dirty blocks in a set to reduce the overhead of write-backs during cache allocations. We configure Flashcache with LRU replacement policy.

For both Bankshot and Flashcache, we use a 250 GB 7200 RPM Seagate hard drives as the backing store. We configure Flashcache to use the Bankshot SSD as the cache device, but it does not use any of the caching features. We perform all experiments on a 16-core 2.9 GHz Intel Xeon X5647 host machine with 8 GB DRAM. We minimize the use of page cache by running the experiments with 0_DIRECT flag as page cache is volatile and not durable. Further, bypassing the page cache provides closer to raw SSD performance and allows us to full advantage of the low latency access to storage [13, 96].

For micro-benchmark tests we use the Flexible IO Tester (fio) [28] to measure the average access latency for read, write, and 50% read/50% write (mixed) workloads for different file sizes (1 GB to 128 GB). The fio tool generates a Zipf [4] workload such that 90% of accesses are to 10% of the data. We limit the cache size to 1 GB to keep testing times manageable. The file system on the backing store is XFS [39]. We run all tests twice for twenty minutes on a warmed, clean cache.

### 3.3.2 Tracking cache contents

To support write-back caching, Bankshot tracks the mapping between cached extents and their ultimate home on the backing device, and it must preserve the mapping across reboots and power failures. Since Bankshot allows applications to modify data directly, metadata information is written to the cache on every cache allocation. We have
experimented with two ways of maintaining this persistent mapping information.

*Software mapping (SM)* In the SM scheme, the cache manager writes the mapping information to the device with a synchronous I/O operation after it fills the cache with valid data.

*Hardware mapping (HM)* In the HM scheme, the kernel passes the mapping information and the cache fill data to the Bankshot SSD via a special write operation and the Bankshot SSD records the mapping information in a dedicated map storage area.

The scoreboard unit in the Bankshot SSD associates the backing store address to every 4 kB block dispatched to the cache controller (Figure 3.2). The controller then persists data and address atomically. We envisage the SSD to use a small supercap that will allow the controllers to persist 4 kB of data and 8 B of address in case of power failure.

In both schemes, the kernel driver scans the reverse mapping table at startup to rebuild the kernel data structure.

Figure 3.8 shows the latency breakdown for writing the cache content map in software and hardware. The hardware-managed scheme reduces the latency by 20 µs, and we use it in our further experiments.

### 3.3.3 Replacement policies

We explore two replacement policies in Bankshot: Round Robin (RR) and Least Recently Used (LRU).

*Round robin* The RR replacement policy evicts chunks in order starting from the beginning of the Bankshot SSD and wrapping around when it reaches the end. RR is a sensible software-only policy because the cache manager cannot track cache usage data, since most accesses bypass the kernel. The disadvantage of RR is that it can cause the
Figure 3.4. **Bankshot read latency**: I/O latency for different file sizes.

Figure 3.5. **Bankshot write latency**: I/O latency for different file sizes.
least recently used Adding hardware support in the Bankshot SSD to track LRU information for chunks allows Bankshot to implement usage-aware caching policies. The hardware maintains a doubly-linked list of chunk numbers in LRU order and updates the list on every access. The updates occur off the critical path of normal accesses.

The cache manager can dequeue the LRU chunk number from the hardware using a special command. The software only queries the LRU information on a cache miss and each query to the hardware incurs 5 \( \mu s \) of latency compared to the hundreds or thousands of microseconds it takes to move data to and from the backing store.

Figure 3.4, Figure 3.5 and Figure 3.6 shows the average latency for read, write, and mixed workload under both policies. Baseline+LRU hardware support for data usage tracking provides 2\( \times \) improvement in write latency compared to Baseline+RR for file sizes larger than 16 GB. Also, unlike Flashcache, which suffers from both capacity
and conflict misses, Bankshot’s fully associative mapping eliminates conflict misses and provides consistent write latency. Further, direct user-space access reduces the cache hit latency for read I/Os (1 GB file size) from an average of 106 $\mu$s to 11.5 $\mu$s. Baseline+LRU improves the average read latency by 10% over Baseline+RR.

### 3.3.4 Tracking dirty data

Bankshot needs to know when data in the cache have changed to reduce the write-back I/O to the backing store. Tracking dirty data presents a problem similar to tracking LRU information: since applications access the Bankshot SSD directly, the cache manager is unaware of writes to cached data. There are two options. First, the cache manager can use the Bankshot SSD’s permission mechanism to detect the initial write to an extent from software. Second, we can add hardware support to the Bankshot SSD to track dirty data directly.

**Tracking dirty data in software** Each cache block in a chunk has a bit in the cache manager that indicates the dirtiness of the block. When the cache manager copies a clean data block into the Bankshot SSD it provides the application only with read permission to the chunk. If the application attempts data modification it will fail in the hardware. This forces the application to request the cache manager to service the cache miss. The cache manager marks the dirty bit for the cache chunk and updates the Bankshot SSD permission table with write permissions for the chunk. The permissions cache miss occurs only on the first write to a chunk and the cache manager services this request without needing to access the backing store.

One drawback to this approach is that the cache manager handles hardware permissions on a per-chunk basis and will mark the entire chunk as dirty. Bankshot will need to write-back the whole chunk on an eviction.

**Tracking dirty data in hardware** Adding hardware support for tracking dirty data
can eliminate needless write-backs and reduce the number of permission misses. The Bankshot SSD stores a dirty bit for every 4 kB cache block in a fixed region of its non-volatile memory. On every write from the application, the cache controller sets the dirty bit. Once the hardware sets the dirty bit, only the cache manager can clear the bit. During an eviction, the cache manager queries the dirty bits to determine which blocks it needs to write-back.

Unlike a software dirty bit, a hardware dirty bit allows applications to modify data in the cache without taking a cache miss. Also, tracking dirty information in the hardware incurs less than 250 ns of overhead on the write data path.

However, hardware tracking of dirty bits requires the cache manager to issue an I/O to the Bankshot SSD on a capacity miss. For chunks with dirty blocks, the savings in useless write-backs amortizes the cost of extra I/O.

Hybrid dirty data tracking We can combine the hardware and software dirty data tracking mechanisms to provide the best of both worlds. The cache manager can rely on the software mechanism to determine when a chunk contains some dirty blocks and then use the hardware mechanism to determine which blocks are actually dirty.
Figure 3.4, Figure 3.5 and Figure 3.6 shows the average latency for different dirty bit schemes. For write-only workloads, software dirty bit (Baseline+LRU) incurs the minimum overhead and achieves up to 5× improvement in latency compared to Flashcache. The HW+LRU and Hybrid+LRU schemes are 40% slower than Baseline+LRU but still 2×-3× faster than Flashcache. As all the data is dirty for a write-only workload, querying the hardware incurs an extra latency cost compared to a software-only scheme. For a read-only workload, Bankshot achieves a 10% improvement in latency over Flashcache on average. Even though Bankshot improves read latency for cache hits by 50% compared to Flashcache (see Figure 3.7), Bankshot’s fully associative mapping only improves the cache hit ratio by 4%. The backing store performance dominates Bankshot’s and Flashcache’s improved read latency.

For a 50% read/50% write workload, Hybrid+LRU achieves on average a 28% improvement in latency compared to Baseline+LRU, as seen in Figure 3.6. Hybrid+LRU achieves this improvement in performance by eliminating the need to write-back clean data. For a 1 GB file size, the hybrid scheme achieves an average access latency of 15.8 µs while Flashcache’s access latency is 52 µs. Beyond 2 GB and up to 32 GB, the average access latency increases as the larger number of misses requires increased disk reads resulting in similar performance for Bankshot and Flashcache. However, beyond 64 GB, Flashcache shows 10% improvement in latency over Hybrid+LRU scheme because of two reasons. First, Flashcache always performs asynchronous write back. Second, Flashcache improves write miss latency by preferring to evict clean blocks over dirty blocks in its LRU. For software or hardware only schemes, Bankshot’s performance is less than Flashcache. Software only schemes perform redundant write back and hardware only scheme incur extra IO for querying dirty bits from the Bankshot SSD.
Figure 3.8. **Bankshot cache content tracking**: Hardware vs software-based cache content tracking

Figure 3.9. **Bankshot asynchronous write-back with software dirty tracking**
Figure 3.10. Bankshot asynchronous write-back with hardware dirty tracking

Figure 3.11. Bankshot asynchronous write-back with hybrid dirty tracking
3.3.5 Asynchronous write-back

The chunk eviction mechanism schedules write-back to disk on demand which can require two or more I/Os to the backing store to complete the miss. Write-back on an eviction underutilizes the backing store bandwidth and can also generate random reads and writes that affect the backing store access latency. Asynchronous write-back of data blocks can improve system bandwidth by moving write-backs to the background and scheduling them to avoid random accesses.

Bankshot runs a kernel thread called cleaner thread in the background to perform asynchronous write-back. The cache manager periodically triggers the cleaner thread to write dirty data to backing store. First, the thread selects a chunk for cleaning and updates the state of the chunk to indicate that the cache write back is in-progress. Second, it checks the dirty data information for the cache block. If it is dirty, it writes the data to the backing store. Finally, it marks the block as clean and clears the in-progress flag.

We explore three different schemes for asynchronous write-back async-RR, async-LRU and elevator.

Async-RR The async-RR write-back scheme works in tandem with the RR replacement policy. The cache manager treats the chunks in the cache as a circular buffer with two pointers: a fill pointer and an asynchronous clean pointer. The algorithm triggers the cleaner thread when the clean block count drops below the minimum threshold. The thread cleans dirty chunks until the number of clean blocks is more than the maximum threshold and then moves the asynchronous clean pointer forward.

Async-LRU Async-LRU cleans the least recently used chunks. Similar to the LRU replacement policy, the async-LRU scheme requires the support of the Bankshot SSD for identifying LRU chunks. Along with a command to retrieve the LRU chunk, the Bankshot SSD lets the OS retrieve the last 256 chunks in the LRU list. If the number of
dirty blocks in the list is below a minimum threshold it cleans chunks in the background until the pool of clean chunks is above a threshold.

The async-RR and async-LRU schemes write-back dirty data to the backing store to maximize the number of clean victim chunks. The write request from these schemes and the I/O requests from cache misses can produce random accesses to the backing store, degrading performance.

**Elevator** Elevator opportunistically schedules cleaning to maximize locality. When the cleaner thread needs to clean a block, it traverses the physical map to select a file extent that has the minimum seek distance from most recently accessed backing store address and cleans the dirty blocks in that extent. The resulting sequential IO access for elevator scheme can benefit both disk and SSDs [62].

**Supporting asynchronous write-back** Asynchronous write-back needs to modify the dirty information of cache blocks. The cleaner thread can revoke write permissions in the hardware before modifying the software dirty bit. However, naively implementing asynchronous cleaning can lead to a race condition between the cleaner thread and the applications. Since the cleaner thread must query the dirty bits in the hardware, write-back the data, and then clear the bits while at the same time allowing applications to service hits from the Bankshot SSD.

Figure 3.12(a) shows how four dirtiness states can eliminate the race condition. The cleaner thread queries the dirty information of a block using the ‘checkout’ command. The cache controller in the SSD processes the command by marking dirty blocks as ‘un-modified’. The cleaner thread can then write the dirty data to the backing store. If an application writes between the ‘check-out’ and ‘check-in’ command the cache controller updates the state of ‘un-modified’ blocks as ‘modified’. Once the write-back completes, the cleaner thread issues a ‘check-in’ command to the SSD. The cache controller sets
Figure 3.12. Bankshot Hardware dirty bit tracking state machine and efficiency of asynchronous write-back schemes: The SSD uses 2-bits per 4 kB block for dirty information tracking as shown in (a). Performing asynchronous write back using hybrid dirty tracking provides better cache efficiency as shown in (b).

‘un-modified’ blocks as clean and ‘modified’ blocks as dirty.

To study the usefulness of the asynchronous write-back schemes we use the write workload to measure the average I/O latency and an average number of clean blocks in the cache. Figure 3.9, Figure 3.10 and Figure 3.11 shows the average latency for writes under different dirty bit and asynchronous write-back schemes, and Figure 3.12(b) represents the average percentage of clean blocks in the cache. Performing asynchronous write-backs in the software-only dirty scheme increases the average latency while maintaining only 10% of the cache blocks as clean (Figure 3.12(b)). For both hardware-only and hybrid schemes, elevator write-back not only reduces the average latency by up to 20%, it also maintains 40% of the cache in the clean state.
Table 3.3. Bankshot Application Performance: Berkeley-DB key-value workloads

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Read:Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTree</td>
<td>5.1GB</td>
<td>3:1</td>
</tr>
<tr>
<td>Hash</td>
<td>41GB</td>
<td>9:1</td>
</tr>
</tbody>
</table>

Figure 3.13. Bankshot application performance: BTree achieves 2.9× speedup over Flashcache
3.4 Applications

To study the application performance of Bankshot we use key-value store benchmarks under BDB. Table 3.3 describes the two BDB workloads, BTree and Hash. For these experiments, we limit cache capacity to 2 GB to reduce the time needed to warm up the cache and create the BDB database. We run the experiment using 16 threads and run them for one hour to ensure the cache is warm.

Figure 3.13 compares the performance of the different design options discussed in Section 3.3 against Flashcache. Bankshot provides up to $3 \times$ average speedup over Flashcache for BDB-BTree. The HW+LRU provides a 10% performance improvement over the Baseline+RR. Adding elevator write-back decreases the performance by up to 14% compared to Baseline+RR while maintaining more than 20% of the cache blocks clean.

Bankshot for BDB-Hash is only 15% faster than Flashcache because BDB-Hash includes many low-locality reads. Further, for BDB-Hash, LRU replacement policy experiences a 47% hit ratio compared to 67% for BDB-BTree.

3.5 Related Work

Bankshot is similar in some respects to previous research proposals and some commercial products. In this section, we place Bankshot in the context of systems that use SSDs as caches and those that provide storage interfaces specialized for caching.

Flash-based SSD caches Many storage vendors provide caching solutions that utilize PCIe flash-based SSDs as a write-back or write-through cache [30, 92, 66]. Cache managers for SSDs are also available in the open source community [87, 7]. Linux 3.9 includes support for using SSDs as disk caches using the device mapper framework and bcache [7]. However, all these systems suffer from the large software overhead that
Bankshot aims to eliminate.

Previous flash-based write-back caches [87, 7] also suffer from two additional drawbacks. First, caching systems that use flash-based SSDs implement complex schemes to coalesce metadata information across multiple cache blocks to leverage the page-write property of NAND. These management schemes result in varying cache latencies. With emerging NVMs that provide a byte-addressable interface for in-place updates and endurance several orders of magnitude better than NAND flash [40], the complex schemes add layers of software that contribute to nearly 37% of cache hit latency (Figure 3.7). Secondly, requiring separate I/O for data and metadata introduces inconsistencies between cache contents and cache metadata [87]. Bankshot’s specialized hardware eliminates these problems.

*Specialized caching interface* The Bankshot design draws from previous work on specialized hardware interfaces for SSD caches [82, 83, 1, 61, 81]. FlashTier [82] proposes a specialized interface for caching specific services in flash-based SSDs. FlashTier reduces the multiple levels of address translation by combining the cache address map with the Flash Translation Layer (FTL) of the SSD. It provides a consistent and durable cache through specialized SSD commands and uses the garbage collector within the SSD to perform silent eviction of clean data. While Bankshot shares some features with FlashTier it differs in the following two ways.

First, Bankshot develops the interface required to fully benefit from using emerging fast NVMs such as PCM and STT-RAM as caches. For instance, recent work [78] has shown that these memory technologies do not require complicated FTL for management. Further, they allow in-place updates, thus eliminating the need for garbage collection.

Second, Bankshot implements all cache address space management functionality from within the OS. Unlike FlashTier, Bankshot performs lookups and inserts using the host CPU and DRAM. This allows Bankshot to leverage the file-level information
available in the kernel along with simple LRU logic in the hardware to make better eviction decisions.

3.6 Summary

This chapter described Bankshot, a caching system for fast, emerging NVMs. Bankshot reduces the OS and file system overhead of servicing cache hits by allowing applications to directly modify the data in the cache. Reducing software overhead and adding hardware support for reliability and dirty data tracking provides significant performance benefits for writes. We examined a broad set of design options for Bankshot and found limited hardware support is helpful in improving performance and that a combination of hardware and software mechanisms can lead to large performance gains compared to existing kernel-only schemes.

Acknowledgements

This chapter contains material from “Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW ’13). ACM, New York, NY, USA. And ‘Bankshot: caching slow storage in fast non-volatile memory.”, by Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson, which appears in ACM SIGOPS Operating Systems Review, Volume 48, Issue 1 May 2014. The dissertation author was the first investigator and author of this paper.
Chapter 4

A User-Programmable SSD

The scope of possible new interfaces to storage is enormously broad and includes both general-purpose and application-specific approaches. The previous chapter has illustrated some of the possibilities and the potential benefits of specialized interface for caching operations. The new interfaces allowed applications to leverage SSDs’ low latency, ample internal bandwidth, and on-board computational resources, and they can lead to huge improvements in performance.

Other researchers have implemented functionality in hardware and found, as we did, that it was really hard and time consuming. First, adding features is complex and requires access to SSD internals, so only the SSD manufacturer can add them. Second, the code must be trusted, since it can access or destroy any of the data in the SSD. Third, to be cost-effective for manufacturers to develop, market, and maintain, the new features must be useful to many users and/or across many applications. Selecting widely applicable interfaces for complex use cases is very difficult.

To overcome these limitations, this chapter looks at the ease with which application developers can extend the SSD functionality using Willow. We first introduce Willow system components that allows application, file system, and operating system programmers to install customized (and potentially untrusted) SSD Apps that can modify and extend the SSD’s behavior.
Figure 4.1. Willow high level architecture: Although, like conventional SSD, Willow (a) contain programmable components. Willow’s computation resources (b) are visible to the programmer and provide a flexible programming model.

Applications will be able to exploit this kind of programmability in (at least) four different ways.

- **Data-dependent logic:** Many storage applications perform data-dependent read and write operations to manipulate on-disk data structures. Each data-dependent operation requires a round-trip between a conventional SSD and the host across the system bus (i.e., PCIe, SATA, or SAS) and through the operating system, adding latency and increasing host-side software costs.

- **Semantic extensions:** Storage features like caching and logging require changes to the *semantics* of storage accesses. For instance, a write to a caching device could include setting a dirty bit for the affected blocks.

- **Privileged execution:** Executing privileged code in the SSD will allow it to take over operating and file system functions. Recent work [13] shows that issuing a request to an SSD via an OS-bypass interface is faster than a system call, so
running some trusted code in the SSD would improve performance.

- **Data intensive computations:** Moving data-intensive computations to the storage system has many applications, and previous work has explored this direction in disks [80, 3, 44] and SSDs [42, 9, 91] with promising results.

Willow focuses on the first three of these use cases and demonstrates that adding generic programmability to the SSD interface can significantly reduce the cost and complexity of adding new features. We describe a prototype implementation of Willow based on emulated PCM memory that supports a wide range of applications. Then, we highlight some of the design decisions in building the prototype. We report on our experience implementing a couple of example SSD Apps. The results show that Willow allows programmers to quickly add new features to an SSD and that applications can realize significant gains by offloading functionality to Willow.

This chapter provides an overview of Willow, its programming model, and our prototype in Sections 4.1 and 4.2. Section 4.3 describes our three SSD Apps, Section 4.4 places our work in the context of other approaches to integrating programmability into storage devices and Section 4.5 concludes.

### 4.1 System Design

Willow revisits the interface that the storage device exposes to the rest of the system, and provides the hardware necessary to support that interface efficiently. This section describes the system from the programmer’s perspective, paying particular attention to the programming model and hardware/software interface. Section 4.2 describes the prototype hardware in more detail.
Figure 4.2. The anatomy of an SSD App: The boldface elements depict three components of an SSD App: a userspace library, the SPU code, and an optional kernel driver. In the typical use case, a conventional file system manages the contents of Willow, and the Willow driver grants access to file extents based on file system permissions.

4.1.1 Willow system components

Modern, high-end SSDs contain several (often many) embedded, programmable processors, but that programmability is not visible to the host system or to applications. Figure 4.1(a) shows the picture of the Willow SSD. Willow’s components resemble those in a conventional SSD: it contains several storage processor units (SPUs), each of which includes a microprocessor, an interface to the inter-SPU interconnect, and access to an array of non-volatile memory. Each SPU runs a very small operating system called SPU-OS that manages and enforces security (see Section 4.1.6 below).

The interface that Willow provides is very different from the interface of a conventional SSD. On the host side, the Willow driver creates and manages a set of objects called Host RPC Endpoints (HREs) that allow the OS and applications to communicate with SPUs. The HRE is a data structure that the kernel creates and allocates to a process. It provides a unique identifier called the HRE ID for sending and receiving RPC requests.
and lets the process send and receive those requests via DMA transfers between userspace memory and the Willow SSD. The SPUs and HREs communicate over a flexible network using a simple, flexible RPC-based mechanism. The RPC mechanism is generic and does not provide any storage-specific functionality. SPUs can send RPCs to HREs and vice versa.

The final component of Willow is programmable functionality in the form of SSD Apps. Each SSD App consists of three elements: a set of RPC handlers that the Willow kernel driver installs at each SPU on behalf of the application, a library that an application uses to access the SSD App, and a kernel module, if the SSD App requires kernel support. Multiple SSD Apps can be active at the same time.

Below, we describe the high-level system model, the programming model, and the security model for both SPUs and HREs.

4.1.2 The Willow Usage Model

Willow’s design can support many different usage models (e.g., a system could use it as a tightly-coupled network of “wimpy” compute nodes with associated storage). Here, however, we focus on using Willow as a conventional storage device that also provides programmability features. This model is particularly useful because it allows for the incremental adoption of Willow’s features and ensures that legacy applications can use Willow without modification.

In this model, Willow runs an SSD App called Base-I0 that provides basic block device functionality (i.e., reading and writing data from and to storage locations). Base-I0 stripes data across the SPUs (and their associated banks of non-volatile memory) in 8 kB segments. Base-I0 (and all the other SSD Apps we present in this paper) runs identical code at each SPU. We have found it useful to organize data and computation in this way, but Willow does not require it.
A conventional file system manages the space on Willow and sets permissions that govern access to the data it holds. The file system uses the Base-I0 block device interface to maintain metadata and provide data access to applications that do not use Willow’s programmability.

To exploit Willow’s programmability, an application needs to install and use an additional SSD App. Figure 4.2 illustrates this process for an SSD App called Direct-I0 that provides an OS-bypass interface that avoids system call and file system overheads for common-case reads and writes (similar to [13]). The figure shows the software components that comprise Direct-I0 in bold. To use Direct-I0, the application uses the Direct-I0’s userspace library, libDirectI0. The library asks the operating system to install Direct-I0 in Willow and requests an HRE from the Willow driver to allow it to communicate with the Willow SSD.

Direct-I0 also includes a kernel module that libDirectI0 invokes when it needs to open a file on behalf of the application. The Direct-I0 kernel module asks the Willow driver to grant the application permission to access the file. The driver requests the necessary permission information from the file system and issues trusted RPCs to SPU-OS to install the permission for the file extents the application needs to access in the SPU-OS permission table. Modern file systems already include the ability to query permissions from inside the kernel, so no changes to the file system are necessary.

Base-I0 and Direct-I0 are “standard equipment” on Willow since they provide functions that are useful for many other SSD Apps. In particular, other SSD Apps can leverage Direct-I0’s functionality to implement arbitrary, untrusted operations on file data.
4.1.3 Building an SSD App

SSD Apps comprise interacting components running in multiple locations: in the client application (e.g., libDirectIO), in the host-side kernel (e.g., the Direct-I0 kernel module), and in the Willow SSD. To minimize complexity, the code in all three locations uses a common set of interfaces to implement SSD App functionality. In the host application and the kernel, the HRE library implements these interfaces, while in Willow, SPU-OS implements them. The interfaces provide the following capabilities:

1. **Send an RPC request:** SPUs and HREs can issue RPC requests to SPUs, and SPUs can issue RPCs to HREs. RPC delivery is non-reliable (due to limited buffering at the receiver), and all-or-nothing (i.e., the recipient will not receive a partial message). The sender is notified upon successful (or failed) delivery of the message. Willow supports both synchronous and asynchronous RPCs.

2. **Receive an RPC request:** RPC requests carry an **RPC ID** that specifies which SSD App they target and which handler they should invoke. When an RPC request arrives at an SPU or HRE, the runtime (i.e., the HRE library or SPU-OS) invokes the correct handler for the request.

3. **Send an RPC response:** RPC responses are short, fixed-length messages that include a result code and information about the request it responds to. RPC response delivery is reliable.

4. **Initiate a data transfer:** An RPC handler can asynchronously transfer data between the network interface, local memory, and the local non-volatile memory (for SPUs only).

5. **Allocate local memory:** SSD Apps can declare static variables to allocate space in the SPU’s local data memory, but they cannot allocate SPU memory dynamically. The code on the host can allocate data statically or on the heap.
6. **General purpose computation**: SSD Apps are written in C, although the standard libraries are not available on the SPUs.

In addition to these interfaces, the host-side HRE library also provides facilities to request HREs from the Willow driver and install SSD Apps.

This set of interfaces has proved sufficient to implement a wide range of different applications (see Section 4.3), and we have found them flexible and easy to use. However, as we gain more experience building SSD Apps, we expect that opportunities for optimization, new capabilities, and bug-preventing restrictions on SSD Apps will become apparent.

### 4.1.4 The SPU Architecture

In modern SSDs (and in our prototype), the embedded processor that runs the SSD’s firmware offers only modest performance and limited local memory capacity compared to the bandwidth that non-volatile memory and the SSD’s internal interconnect can deliver.

In addition, concerns about power consumption (which argue for lower clock speeds) and cost (which argue for simple processors) suggest this situation will persist, especially as memory bandwidths continue to grow. These constraints shape both the Willow hardware we propose and the details of the RPC mechanism we provide.

The SPU has four hardware components we use to implement the SSD App toolkit (Figure 4.1(b)):

1. **SPU processor**: The processor provides modest performance (perhaps 100s of MIPS) and kilobytes of per-SPU instruction and data memory.
2. **Local non-volatile memory**: The array of non-volatile memory can read or write data at over 1 GB/s.
3. **Network interface**: The network provides gigabytes-per-second of bandwidth to
// RPCHdr_t part of the RPC interface
void Read_Handler (RPCHdr_t *request_hdr) {
    // Parse the incoming RPC
    BaseIOCmd_t cmd;
    // DMA the IO command header
    RPCReceiveBytes(&cmd, sizeof(BaseIOCmd_t));
    // Allocate response
    RPCResp_t response_hdr;
    // Populate the response
    RPCCreateResponse(request_hdr, &response_hdr, RPC_SUCCESS);
    // Send the response
    RPCSendResponse(response_hdr);

    // Send the read data back via a second RPC
    CPUID_t dst = request_hdr->src;
    RPCStartRequest(dst, // Destination PU
                    sizeof(IOCmd_t) + cmd.length, // Request body length
                    READ_COMPLETE_HANDLER); // Read completion RPC ID

    RPCAppendRequest(LOCAL_MEMORY_PORT, // Source DMA port
                     sizeof(BaseIOCmd_t), // IO command header size
                     &cmd); // IO command header address

    RPCAppendRequest(NV_MEMORY_PORT, // Source DMA Port
                     cmd.length, // Bytes to read
                     cmd.addr); // Read address

    RPCFinishRequest(); // Complete the request
}

Figure 4.3. READ() implementation for Base-IO: Handling a READ() requires parsing the header on the RPC request and then sending requested data from non-volatile memory back to host via another RPC.

match the bandwidth of the local non-volatile memory array and the link bandwidth to the host system.

4. Programmable DMA controller: The DMA controller routes data between non-volatile memory, the network port, and the processor’s local data memory. It can handle the full bandwidth of the network and local non-volatile memory.

The DMA controller is central to the design of both the SPU and the RPC mechanism since it allows the modestly powerful processor to handle high-bandwidth
streams of data. We describe the RPC interface in the following section.

4.1.5 The RPC Interface

The RPC mechanism’s design reflects the constraints of the hardware described above. Given the modest performance of the SPU processor and its limited local memory, buffering entire RPC messages at the SPU processor is not practical. Instead, the RPC library parses and assembles RPC requests in stages. The code in Figure 4.3 illustrates how this works for a simplified version of the \texttt{READ()} RPC from \texttt{Base-IO}.

When an RPC arrives, SPU-OS copies the RPC header into a local buffer using DMA and passes the buffer to the appropriate handler (\texttt{ReadHandler}). That handler uses the DMA controller to transfer the RPC parameters into the SPU processor’s local memory (\texttt{RPCReceiveBytes}). The header contains generic information (e.g., the source of the RPC request and its size), while the parameters include command-specific values (e.g., the read or write address). The handler uses one or more DMA requests to process the remainder of the request. This can include moving part of the request to the processor’s local memory for examination or performing bulk transfers between the network port and the non-volatile memory bank (e.g., to implement a write). In the example, no additional DMA transfers are needed.

The handler sends a fixed-sized response to the RPC request (\texttt{RPCCreateResponse} and \texttt{RPCSendResponse}). Willow guarantees the reliable delivery of fixed-size responses (acks or nacks) by guaranteeing space to receive them when the RPC is sent. If the SSD App needs to send a response that is longer than 32 bits (e.g., to return the data for a read), it must issue an RPC to the sender. If there is insufficient buffer space at the receiver, the inter-SPU communication network can drop packets. In practice, however, dropped packets are exceedingly rare.

The process of issuing an RPC to return the data follows a similar sequence
of steps. The SPU gives the network port the destination and length of the message (RPCStartRequest). Then it prepares any headers in local memory and uses the DMA controller to transfer them to the network interface (RPCAppendRequest). Further DMA requests can transfer data from non-volatile memory or processor memory to the network interface to complete the request. In this case, the SSD App transfers the read data from the non-volatile memory. Finally, it makes a call to signal the end of the message (RPCFinishRequest).

### 4.1.6 Protection and sharing in Willow

To enforce operating system protection SPU must allow an SSD App to access data stored in Willow only if the process that initiated the current RPC has access rights to that data. To limit access to data in the non-volatile memory banks, SPU-OS maintains a set of permissions for each process at each SPU. Every time the SSD App uses the DMA controller to move data to or from the non-volatile memory, SPU-OS checks that the permissions for the originating HRE (and therefore the originating process) allow it. The worst-case permission check latency is 2 μs.

The host-side kernel driver installs extent-based permission entries on behalf of a process by issuing privileged RPCs to SPU-OS. Since the SPU-OS permission table is fixed size, it may evict permissions if space runs short. If a request needs an evicted permission entry, a “permission miss” occurs, and the DMA transfer will fail. In response, SPU-OS issues an RPC to the kernel. The kernel forwards the request to the SSD App’s kernel module (if it has one), and that kernel module is responsible for resolving the miss.

**Limiting access to OS only RPCs:** A combination of hardware and software feature enables SSD Apps to create RPCs that can be issued only from the host-side kernel. To implement kernel-only RPCs, we use the convention that a zero in the high-order bit of the HRE ID means the HRE belongs to the kernel. RPC implementations can check the
Table 4.1. Implementing Willow Apps: Development time and effort for SSD Apps: Implementing and testing each SSD App required no more than 1700 lines of code.

<table>
<thead>
<tr>
<th>Name</th>
<th>Fixed-function LOC (Verilog)</th>
<th>Devel. Time (Person-months)</th>
<th>Willow LOC (C)</th>
<th>Devel. Time (Person-months)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caching</td>
<td>1336</td>
<td>12</td>
<td>728</td>
<td>1</td>
</tr>
<tr>
<td>Zero-out &amp; Append</td>
<td>n/a</td>
<td>n/a</td>
<td>1688</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Figure 4.4. The Willow prototype architecture: A ring network connects the SPUs to each other and to a bridge that provides communication to the host.

ID and return failure when a non-kernel HRE invokes a protected RPC.

4.2 The Willow Prototype

We have constructed a prototype Willow SSD that implements all of the functionality described in the previous section. This section provides details about the design.

The prototype has eight SPUs and a total storage capacity of 64 GB (see Figure 4.4) and is implemented using a BEE3 FPGA-based prototyping system [37].

Each of the four FPGAs that make up a BEE3 hosts two SPUs, each attached to an 8 GB bank of DDR2 DRAM. We use the DRAM combined with a customized
memory controller to emulate phase change memory with a read latency of 48 ns and a write latency of 150 ns. The memory controller implements start-gap wear-leveling [79].

The SPU processor is a 125 MHz RISC processor with an MIPS-like instruction set. It executes nearly one instruction per cycle, on average. We use the MIPS version of GCC to generate executable code for it. For debugging, it provides a virtual serial port and a rich set of performance counters and status registers to the host. The processor has 32 kB of local data memory and 32 kB of local instruction memory.

The kernel driver statically allocates space in the SPU memory to SSD Apps, which constrains the number and size of SSD Apps that can run at once. SPU-OS maintains a permission table in the local data memory that can hold 768 entries and

---

**Figure 4.5. The Willow bridge network**: The bridge extends the intra-SPU network to the HREs on the host. Communication occurs via queue pairs, one per HRE that reside in host memory.
occupies 20 kB of data memory.

The ring in Willow uses round-robin, token-based arbitration, so only one SPU may be sending a message at any time. To send a message, the SPU’s network interface waits for the token to arrive, takes possession of it, and transmits its data. To receive a message, the interface watches the header of messages on the ring to identify messages it should remove from the ring. The ring is 128 bits wide and runs at 250 MHz for a total of 3.7 GB/s of bisection bandwidth.

For communication with the HREs on the host, a bridge connects the ring to the PCIe link. Figure 4.5 shows its internal organization. The bridge serves as a hardware proxy for the HREs. For each of the HREs, the bridge maintains an upstream (host-bound) and downstream (Willow-bound) queue. This queue-based interface is similar to the scheme that NVMExpress [95] uses to issue and complete IO requests. The bridge in our prototype Willow supports up to 1024 queue pairs, so it can support 1024 HREs on the host.

The bridge also helps enforce security in Willow. Messages from HREs to SPUs travel over the bridge and the bridge sets the originating HRE fields on those messages depending on which HRE queue they came in on. Since processes can send messages only via the queues for the HREs they control, processes cannot send forged RPC requests. Sudharsan et.al [86], provide a detailed description of Willow hardware platform.

4.3 User Applications

Willow makes it easy for storage system engineers to improve performance by incorporating new capabilities into a storage device. We use Willow to implement three different SSD Apps and comparing their performance to implementations that use a conventional storage interface.

The three applications are: caching, zeroing out extents and appending data to a
file in the Ext4 filesystem. For caching, we also compare with Bankshot described in Chapter 3. Table 4.1 briefly describes all three apps and provides some statistics about their implementations. We discuss each in detail below.

4.3.1 Caching

Chapter 3 presented Bankshot caching SSD [8] which reduces the software latency for cache hits by nearly 75% using a customized SSD.

We implemented an SSD App called Caching that turns Willow into a caching SSD and provides the features similar to Bankshot. It combines basic reads and writes of Direct-I0 with cache-specific RPCs for writing data and retrieving information about dirty blocks. It services cache hits directly from user space using Direct-I0’s OS-bypass interface. For misses, Caching invokes a kernel-based cache manager.

Caching transforms Willow into a specialized caching SSD rather than providing application-specific features on top of normal cache access. Instead of using the file system’s extent-based protection policy, Caching uses a specialized permission mechanism based on large, fixed-size cache chunks (or groups of blocks) that make more efficient use of the SPU’s limited local memory. Caching’s kernel module uses a privileged kernel-only RPC to install the specialized permission entries and to manage the cache’s contents.

To measure Caching’s performance we use the Flexible IO Tester (Fio) [28]. We configure Fio to generate Zipf-distributed [4] accesses such that 90% of accesses are to 10% of the data. We vary the file size from 1 GB to 128 GB. We use a 1 GB cache and report average latency after the cache is warm. The backing store is a hard disk.

Figure 4.7 shows the average read and Figure 4.8 write latency for 4 kB accesses to Bankshot, Flashcache and Caching. Because it is a kernel module, Flashcache uses the Base-I0 rather than Direct-I0. Caching’s fully associative allocation policy allows
Figure 4.6. Latency break down of cache hit latency for Bankshot and Caching performance is limited by the software performance of the PU. State of that art SSDs have processors that provide better performance compared to our prototype MIPS processor for more efficient use of cache space, and its ability to allow direct user space access reduces software overheads. Caching reduces the miss rate by 6%-23% and improves the cache hit latency for write by 4.5% and read by 3.5×. Combined, these improve read latency by between 1.1 and 3.5× and writes by up to 4.5% and 1.8×.

Hardware only implementation of the caching functionality in Bankshot improves hit latency for reads by 90% and writes by 63%. However, this overhead can be eliminated entirely by using a 1 GHz SPU (as shown in Figure 4.6). Further, Figure 4.7 and Figure 4.8 demonstrates that the cost of implementing caching in software instead of hardware is no more than 17% for writes and 8% for reads.
Figure 4.7. Read latency vs. working set size for Caching, Bankshot and FlashCache: Caching offers improved average latency for 4 kB reads compared to FlashCache and Bankshot. As the file sizes grow beyond the cache size of 1 GB, latency approaches that of the backing disk.

Figure 4.8. Write latency vs. working set size for Caching, Bankshot and FlashCache: Caching offers improved average latency for 4 kB writes compared to FlashCache and Bankshot. As the file sizes grow beyond the cache size of 1 GB, latency approaches that of the backing disk.
Figure 4.9. RPCs to implement `APPEND()`: The coordinator SPU delegates writing the appended data to the SPUs that host the affected memory banks. Those SPUs notify the host on completion.

Figure 4.10. Performance of file append in Willow: Append provides better performance than relying on the operating system to append data to a file.
4.3.2 File system offload

File systems present several opportunities for offloading functionality to Willow to improve performance. We have created two SSD App called Zero-out and Append that exploits some of these opportunities. Zero-out allows the host software to write zero’s to an extent in the SSD and eliminate the transfer of zero filled buffers. Append allows Direct-I0 to append data to a file (and update the appropriate metadata) from userspace.

Zeroing an extent Applications zero-fill or write out the same pattern to datablocks to guarantee enhanced security and prevent data leak during reallocation of freed space to a different process. Along with user applications, even filesystems need to zero fill the unused inode blocks during initialization for correctness. To accelerate software performance, the T10 workgroup for SCSI [90] standard introduced WRITE-SAME command that can write the same 512 B buffer to multiple contiguous sectors on the disk.

While this reduces some of the redundant data transfer on the SCSI interconnect, software needs to allocate and transfer a minimum of 512 B buffer. Zero-out implements WRITESAME() while making use of Direct-I0 functionality as well. The WRITESAME() eliminates the need for transferring an entire block of data by combining a 64-bit data with command. To measure Zero-out’s performance we modified Ext4 file system to issue WRITESAME() instead of blk-dev-zeroout request to the Willow SSD. By eliminating the redundant data-transfer Zero-Out reduces the latency of a 4 kB zeroing operation from 10.2 $\mu$s to 7.4 $\mu$s.

Appending to a file Direct-I0 reduces overheads for most read and write operations by allowing them to bypass the operating system, but it cannot do the same for append operations, since appends require updates to file system metadata. We can extend the OS bypass interface to include appends by building a trusted SSD App that can coordinate
with the file system to maintain the correct file length.

Append builds upon Direct-I0 (and libDirectIO) and works with a modified version of the Ext4 file system to manage file lengths. The first time an application tries to append to a file, it asks the file system to delegate control of the file’s length to Append. In response, the file system uses a trusted RPC to tell Append where the last extent in the file resides. The file system also sets a flag in the inode to indicate that Append has ownership of the file’s logical length. Since the file system allocates space in 4 kB blocks and may pre-allocate space for the file, the physical length of the file is often much longer than the logical length. The physical length remains under the file system’s control.

After that, the application can send APPEND() RPCs directly to Willow. Figure 4.9 illustrates the sequence of RPCs involved. The APPEND() RPCs include the file’s inode number and the data to append. The application sends the RPC to the SPU whose ID is the inode number modulo the number of SPUs in Willow.

When the SPU receives an APPEND() RPC, it checks to see whether the application has permissions to append to the file and whether appended data will fit in the physical length of the file. If the permission exists and the data will fit, Append issues a special WRITE() to the SPUs that manage the memory that is the target of the append (there may be more than one depending on the size and alignment of the update). While the writes are underway, APPEND() logs the updated length to persistent storage (for crash recovery), and sends a response to the application.

This response does not signal the completion of the APPEND(). Instead, it contains the number of WRITE()s that the coordinating SPU issued and the starting address of the append operation. The WRITE()s for the append notify the host-side application (rather than the coordinating SPU) when they are complete via an APPENDDONE() RPC. When the application has received all of the APPENDDONE() RPCs, it knows the APPEND() is complete. If any of the writes fail, the application needs to re-issue the write using
Direct-I0.

If the append data will not fit in the physical length of the file, Append sends an “insufficient space” response to the host-side application. The host-side application then invokes the file system to allocate physical space for the file and notify the SPU.

If the file system needs to access the file’s length, it retrieves it from the SSD and updates its in-memory data structures.

Figure 4.10 compares the performance of file appends using Append and using Base-I0. For Base-I0 we open the file with O_DSYNC, which provides the same durability guarantees as Append. The appends are 1 kB. We modify Ext4 to pre-allocate 64 MB of physical extents. Append improves append latency by $2.5 \times$ and bandwidth by between $4 \times$ and $5.7 \times$ with multiple threads.

4.4 Related Work

Many projects (and some commercial products) have integrated compute capabilities into storage devices, but most of them focus on offloading bulk computation to an active hard drive or (more recently) an SSD.

In the 1970s and 1980s, many advocates of specialized database machines pressed for custom hardware, including processor-per-track or processor-per-head hard disks to achieve processing at the storage device. None of these approaches turned out to be successful due to high design complexity and manufacturing cost.

Several systems, including CASSM [88], RAP [70], and RARES [55] provided a processor for each disk track. However, the extra logic required to enable processing ability on each track limited storage density, drove up costs and prevented processor-per-track from finding wide use.

Processor-per-head techniques followed, with the goal of reducing costs by associating processing logic with each read/write head of a moving head hard disk. The
Ohio State Data Base Computer (DBC)[43] and SURE [53] each took this approach. These systems demonstrated good performance for simple search tasks, but could not handle more complex computation such as joins or aggregation.

Two different projects, each named Active Disks, continued the trend toward fewer processors, providing just one CPU per disk. The first [80] focused on multimedia, database, and other scan-based operations, and their analysis mainly addressed performance considerations. The second [3] provided a more complete system architecture but supported only stream-based computations called disklets.

Several systems [84, 23] targeted (or have been applied to) databases with programmable in-storage processing resources and some integrated FPGAs [63, 64]. IDisk [44] focused on decision support databases and considered several different software organizations, ranging from running a full-fledged database on each disk to just executing data-intensive kernels (e.g., scans and joins). Willow resembles the more general-purpose programming models for IDisks.

Recently researchers have extended these ideas to SSDs [25, 48], and several groups have proposed offloading bulk computation to SSDs. The work in [42] implements Map-Reduce [24]-style computations in an SSD, and two groups [9, 91] have proposed offloading data analysis for HPC applications to the SSD’s processor. Samsung is shipping an SSD with a key-value interface.

Projects that place general computation power into other hardware components, such as programmable NICs, have also been proposed [29, 94, 58]. These devices allow for application-specific code to be placed within the NIC in order to offload network-related computation. This in turn reduces the load on the host OS and CPU in a similar manner to Willow.

Most of these projects focus on bulk computation, and we see that as a reasonable use case for Willow as well, although it would require a faster processor. However,
Willow goes beyond bulk processing to include modifying the semantics of the device and allowing programmers to implement complex, control-intensive operations in the SSD itself. Some programmable NICs have taken this approach. Many projects [20, 69, 74, 8, 82, 41, 101, 13, 17] have shown that moving these operations to the SSD is valuable, and making the SSD programmable will open up many new opportunities for performance improvement for both application and operating system code.

4.5 Conclusion

Solid state storage technologies offer dramatic increases in flexibility compared to conventional disk-based storage, and the interface that we use to communicate with storage needs to be equally flexible. Willow offers programmers the ability to implement customized SSD features to support particular applications. The programming interface is simple and general enough to enable a wide range of SSD Apps that can improve performance on a wide range of applications.

Acknowledgements

This chapter contains material from “Willow: a user-programmable SSD”, by Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson, which appears in In Proceedings of the 11thUSENIX conference on Operating Systems Design and Implementation (OSDI’14). USENIX Association, Berkeley, CA, USA, 67-80. The dissertation author was the third investigator and author of this paper.
Chapter 5

Asynchronous Memory Access

In the previous two chapters, we presented storage architecture that accesses non-volatile memory as I/O device via the PCIe-bus. In this chapter, we present micro-architecture support for NVM attached to the processor memory bus.

As seen in Chapter 2, there are overheads inherent to the PCIe bus which overshadow the potential benefits of accessing NVM as dense byte-addressable memories. NVMs with DRAM-like interface can be placed on the processor memory bus and accessed via load/store instructions. This direct access to NVM eliminates I/O overheads associated with PCIe-based storage devices but can impact overall system performance due to long-latency cache misses and the resulting pipeline stalls.

To reduce the impact of long-latency loads, we propose Non-Blocking Load (NBLD), an instruction set extension that provides asynchronous, non-blocking memory access. NBLD allows the application to initiate a memory fetch and schedule the execution of application-defined call-back that will execute once the data has arrived in the L1 cache.

NBLD combines aspects of asynchronous IO operations and conventional prefetch instructions. By using the completion of the memory fetch to trigger the execution of an application-specific function, NBLD can hide highly variable latencies more effectively that normal prefetching. In the following sections, we describe the semantics of NBLD.
and the microarchitectural support it requires. We specifically address the following:

- **Instruction semantics**-Applications can execute NBLD instructions from user space and the micro-architecture guarantees to invoke the application defined function once and only once.

- **Micro-architectural states**-Architecture support for NBLD introduces a new independent hardware block with small modifications to the frontend of the pipeline and it uses existing memory subsystem for performing the data fetch operation.

- **Memory consistency**-The memory consistency model for NBLD is similar to a system that supports multiple hardware threads.

We implement a prototype system using the Xilinx Zynq System on Chip (SoC) evaluation kit and evaluate NBLD’s performance using Graph applications. The remainder of the chapter is organized as follows - Section 5.1 describes the Instruction Set Architecture (ISA) and micro-architecture modifications required to support Non-Blocking Loads. We describe the prototype system including the hardware and software components used to evaluate the design in Section 5.2. Section 5.3 presents the performance evaluation of NBLD system and finally we provide our summary in Section 5.4.

### 5.1 Non-Blocking Loads

Non-Blocking Load allows an application to bring the contents of a memory address into the processor’s cache and, when the data has arrived, asynchronously trigger the execution of a call back instruction sequence.

This kind of asynchronous programming is commonplace in application and operating system code that executes variable latency operations (e.g., IO requests) that
does not want to “block” on the completion of the operation. By issuing a non-blocking request with a callback, the code frees itself to perform other work.

Asynchronous execution is a powerful programming construct, but implementing it in a processor is challenging because changes to the ISA can have long-lasting and unforeseen consequences and hardware resources are expensive. In order to address these challenges, we designed NBLD to meet four requirements:

- **R1**: NBLD should be easy for programmers to use and use correctly.
- **R2**: NBLD must execute correctly in the presence of context switches (and interrupts) without significantly increasing the complexity of context switch operations.
- **R3**: NBLD should not “pin” cache lines since this increases cache complexity and raises the possibility of deadlock.
- **R4**: NBLD should have minimal impact on processor’s microarchitecture. In particular, ISA implementations should have the flexibility on how many microarchitectural resources to devote to implementing NBLD.

This section describes the instruction’s semantics and the microarchitectural structures that enable NBLD and explain how they meet the above requirements.

### 5.1.1 Instruction Semantics

NBLD necessitates two actions. The first is the *prefetch*. It requests that the memory hierarchy bring the contents of a memory location into the upper-levels of the memory hierarchy. Second is the *execution*, and it triggers the execution of a sequence of instructions. In the expected use case, that sequence of instructions will access the requested data, and the hope is that they will execute more quickly because of the prefetched data.
To implement these two operations, we introduce four new instructions

- **Memory fetch** [ASYNC.FETCH va f h]: Data prefetch instruction that takes three input arguments - effective Load Virtual Address (va), the address of the function to execute (f) and the argument passed to the callback function (h).

- **Callback handle** [ASYNC.HNDL rD]: The asynchronous callback routine that needs to retrieve the argument pointer associated with the ASYNC.FETCH instruction can use this instruction. This instruction copies the callback function argument into the destination register rD.

- **Callback return** [ASYNC.RET]: Transfers the program control to the address at which the asynchronous function execution occurred. Every callback instruction sequence must end with this instruction. Further, executing the instruction outside of the callback instruction sequence is a NOP.

- **Execution Barrier** [ASYNC.B, ASYNC.B.FORCE]: Wait for the execution of outstanding ASYNC.FETCH operations completes.

The function f is subject to several constraints. First, it can take a single argument and access the argument via ASYNC.HNDL (which will provide h in practice). Second, it must restore all registers it modifies during its execution (i.e., all registers are callee-saved).

### 5.1.2 Using Non-Blocking Loads

The four instructions allow the programmer to access the data when it is available in the cache thereby reducing the CPU stalls. In this section, we first describe the semantics of the instructions and then show how the design of NBLD meets requirement R1.
An application invokes `ASYNC.FETCH` instruction as a part of its normal program flow. In order to execute, the application requires no special privileges or exclusive access to a physical resource.

Applications that invoke the `ASYNC.FETCH` instruction can continue to execute the following instructions in the program. When the data becomes available in the processor caches, the micro-architecture branches to the function-\( f \) and utilizes the data in the cache. Micro-architectures that support non-blocking loads guarantee that:

*For every `ASYNC.FETCH` instruction the application commits, the processor branches to the function-\( f \) exactly once.*

This ensures that the application does not experience deadlock situation and can continue to make forward progress, allowing us to meet requirement \( R1 \). The invocation of callback sequence does not change the processor mode or protection ring. In the callback sequence, the application can access the handle argument via `ASYNC.HNDL` instruction and the data at address-\( \text{va} \) via load instruction. In the common case, the data access will hit the cache reducing the CPU stalls. The callback code then restores the register states and transfers the control to the next instruction in the normal flow of execution by executing the `ASYNC.RET` instruction. Applications can force execution of callbacks by invoking the `ASYNC.B` or `ASYNC.B.FORCE` instruction.

In the next section, we describe the micro-architecture changes in the processor that are necessary to satisfy the ISA.

### 5.1.3 Implementation

The semantics for NBLD allows for great flexibility in how a processor implements the instructions, allowing us to meet requirement \( R4 \). First, we describe a *minimal* implementation that requires least modifications to the micro-architecture and then an *aggressive* implementation that exploits the advantages offered by NBLD.
Minimal Implementation: The simplest implementation only requires a new register \( rH \). Upon execution of ASYNC.FETCH it could skip the data prefetch and treat it as a write from \( h \) to register \( rH \) followed by a non-conditional, register-indirect branch and link (or call) to \( f \). The architecture would then treat ASYNC.HNDL as a read from the new register and ASYNC.RET acts as a normal return instruction. The implementation treats ASYNC.B/ASYNC.B.FORCE as NOPs.

Aggressive Implementation: An implementation that fully exploits NBLD to improve performance would allow applications to have many outstanding ASYNC.FETCH instructions. Such an implementation would require changes to the processor’s control and a new microarchitectural table to track ASYNC.FETCH instructions. It also introduces two new registers \( rH \) and \( rNPC \). The ASYNC.HNDL instruction uses the \( rH \) register to access the callback code argument. While \( rNPC \) is a micro-architectural register that stores the return address for ASYNC.RET instruction. The OS on context switch, can save and restore both \( rH \) and \( rNPC \) registers allowing the design to meet requirement \( R2 \). We describe the aspects of the implementation below for each instruction and provide the summary of execution model in Table 5.1

**ASYNC.FETCH**

When the processor frontend dispatches an ASYNC.FETCH instruction there are several steps involved before it retires the instruction.

First, it reads the operands from the register file. Second, it verifies the permission of its operands. Unlike normal load instruction the micro-architecture verifies the permissions of function-\( f \) by probing the I-TLB. This check ensures that the application has the necessary privileges to execute the callback instruction sequence. The instructions can proceed to the commit stage once the permission checks are successful.

The CPU commits the instruction by writing operands to a special hardware table
called the *Async Entry Table (AET)*. Upon successful commit, the front-end retires the instruction.

**Async Entry Table (AET)**

The AET is a per core hardware table that tracks outstanding `ASYNC.FETCH` instructions. When the `ASYNC.FETCH` instruction commits AET captures the instruction operands as a unique table entry. The micro-architecture also captures the application-specific ID (ASID) of the process for each table entry. Associated with each table entry is a state field to track the operation status.

Upon a successful update to the AET an NBLD “control unit” performs the data fetch. The control unit first performs a virtual to physical address translation of the effective virtual address. It then dispatches the data fetch request via the cache hierarchy and sets the state to pending read.

By issuing a request via the cache hierarchy, the hardware provides the guarantee that reads are cache coherent and consistent. The NBLD control unit defers data fetch exceptions to the load instruction that accesses the data in the callback function. To satisfy R4, we envisage a system where the control unit interface to the cache controller is identical to that of a processor core except that there is no data transfer (only address). The control unit can re-order, batch or coalesce the `ASYNC.FETCH` requests to the cache controller to ensure consistent memory system performance.

Once the memory system services the fetch request, the state of entry becomes ready and the NBLD control unit performs the following updates. It sets the handle register (rH) and return address register (rNPC) and then signals the processor’s front end to jump to f. We refer to this operation of the micro-architecture as *f-jump* mechanism.

Vendors can limit the size of the AET by supporting overflow exceptions. When the number of outstanding `ASYNC.FETCH` operations exceeds the size of the table, the AET triggers an interrupt which the OS will handle. The pseudo-code below describes
the sequence of operations:

Upon data availability
rH ← AET.h
rNPC ← nextPC
PC ← AET.f
Callback instruction sequence

The front-end performs the *f-jump* operation only when the system satisfies following two conditions. First, ASID of the AET entry matches the ASID of the current application. In the case of an ASID mismatch, the hardware control unit sets the state of entry as ready and processes the next request. The programmer can force an f-jump for these request using *ASYNC.B* instruction. The second requirement to perform f-jump is that the processor should not be executing another callback instruction sequence.

*ASYNC.HNDL*

The application invokes this instruction to read the value from rH register and store it in the application accessible general purpose register rD. Execution of this instruction outside of the callback sequence returns undefined data.

*ASYNC.RET*

This instruction returns from *ASYNC.FETCH* callback function. The instruction also notifies the NBLD control unit that the processor can now service new *f-jump* operations.

Once the frontend executes the *f-jump*, the hardware control unit clears the AET entry associated with the f-jump, however only after the invocation of *ASYNC.HNDL* and *ASYNC.RET* instructions can the registers rH and rNPC be cleared.
Table 5.1. NBLD instruction set execution model: Pseudo-code for the instruction set extension

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYNC.FETCH</td>
<td>AET.enqueue(va)</td>
</tr>
<tr>
<td></td>
<td>AET.enqueue(f)</td>
</tr>
<tr>
<td></td>
<td>AET.enqueue(h)</td>
</tr>
<tr>
<td></td>
<td>Normal instruction flow</td>
</tr>
<tr>
<td>ASYNC.HNDL</td>
<td>rD ← rH</td>
</tr>
<tr>
<td>ASYNC.RET</td>
<td>PC ← rNPC</td>
</tr>
<tr>
<td>ASYNC.B</td>
<td>rH ← AET.h</td>
</tr>
<tr>
<td>ASYNC.B.FORCE</td>
<td>rNPC ← PC iff AET.size ≠ 0</td>
</tr>
<tr>
<td></td>
<td>nextPC iff AET.size = 0</td>
</tr>
<tr>
<td></td>
<td>PC ← AET.f</td>
</tr>
<tr>
<td></td>
<td>Callback instruction sequence</td>
</tr>
</tbody>
</table>

By enforcing acknowledgment of f-jump operation from the frontend, the micro-architecture provides guarantee that the function-f executes once and only once for each committed ASYNC.FETCH instruction, satisfying requirement **R2**.

**ASYNC.B and ASYNC.B.FORCE** These instructions allow the application to enforce barriers on several outstanding ASYNC.FETCH instruction. ASYNC.B instruction forces the NBLD control unit to fetch the data from the NVM and immediately perform an f-jump operation for an AET entry. If there are no more outstanding ASYNC.FETCH instruction the f-jump mechanism sets the rNPC as the “next PC” as defined by the frontend. However, if there are more entries in the table, the f-jump sets rNPC to “current PC” (address of ASYNC.B instruction). The pipeline control unit of the CPU treats ASYNC.B instruction as a conditional branch instruction to itself and the branch is taken as long as AET has valid entries. The ASYNC.B.FORCE is similar to ASYNC.B, except that it does not wait for the data fetch from NVM to complete.

**Memory Ordering, Consistency, and Protection**

The data accessed by the f adheres to the memory consistency model of the
processor. Any changes to the permissions or translation enforced after the control unit fetches the data are visible to the function. In other words, we identify the consistency and permission model of NBLD is similar to parallel execution of callback function and the code from \texttt{ASYNC.FETCH} to \texttt{ASYNC.B} instruction. As a consequence, similar to conventional asynchronous or multi-threaded program, there is no fixed ordering between \texttt{ASYNC.FETCH} and writes to data. Programs that require ordering can enforce by acquiring locks before invoking \texttt{ASYNC.FETCH} instruction and releasing them after accessing the data within \( f \).

### 5.1.4 Software Interaction

Applications can use non-blocking loads to reduce pipeline stalls and improve the CPU utilization while accessing NVMMs. For example, asynchronous graph processing frameworks can use \texttt{ASYNC.FETCH} instruction to read the vertex and edge values asynchronously and perform compute operations without having to stall the processor owing to random dependent reads. However, using non-blocking loads introduces two main challenges:

- **Synchronization:** NBLD makes no guarantees about the ordering of function-\( f \) execution. Applications that need ordering can use the \texttt{ASYNC.B} instruction to force the execution of outstanding \texttt{ASYNC.FETCH} instructions.

- **Context Switch and interrupts:** First, issuing non-blocking loads in the interrupt context results in undefined behavior of the system. Second, in the event of a context switch the OS is responsible for saving and restoring the RH and NPC states along with AET. To reduce the overhead of context switch, the OS can delay saving the AET as each entry in the table has an ASID associated with it allowing the micro-architecture to support requirement \textbf{R2}.
Table 5.2. Zynq SoC features and peripherals

<table>
<thead>
<tr>
<th></th>
<th>Zynq Processing System</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Dual Core ARM Cortex-A9</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32kB I-Cache/32kB D-Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512kB Unified (8-way)</td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>256kB</td>
</tr>
<tr>
<td>Coherence Unit</td>
<td>Snoopy Coherency Controller</td>
</tr>
<tr>
<td>Memory</td>
<td>1GB DDR3 533MHz</td>
</tr>
<tr>
<td>Master Interconnects</td>
<td>2xAXI Full</td>
</tr>
<tr>
<td>Slave Interconnects</td>
<td>2xAXI Full</td>
</tr>
<tr>
<td></td>
<td>4xAXI HP (only to DDR)</td>
</tr>
<tr>
<td></td>
<td>1x Accel. Coherency port (ACP)</td>
</tr>
</tbody>
</table>

5.2 Methodology

In order to evaluate the application level impact of NBLD we develop a prototype implementation using an FPGA platform. This section describes the hardware and software components used to emulate the asynchronous load operation and asynchronous function execution. We also describe the limitations imposed by the hardware platform, in particular, the limited architectural states exposed by the processor.

Prototype Hardware

We built the prototype on a Xilinx Zynq SoC ZC706 [100] evaluation platform. The Zynq SoC consist of a dual-core ARM Cortex A9 processor (PS) and Kintex-7 based FPGA fabric (PL). Table 5.2 summarizes the SoC features. The FPGA fabric implements the AET and the hardware control unit. The memory controller for the emulated NVM is shown in Figure 5.1. We emulate the `ASYNC.FETCH` instruction via the Zynq’s cache coherent interconnect while relying on a software library to perform `f-jump` and `ASYNC.HNDL` operations. The software library also provides a mechanism for enforcing `ASYNC.B` but does not require `ASYNC.RET` implementation.
Figure 5.1. Zynq SoC based evaluation architecture: Xilinx Zynq SoC’s ARM Cortex-A9 processor (Processing system) implements the software stack and the FPGA fabric implements the non-blocking load hardware logic.

Figure 5.2. Zynq SoC based software stack for NBLD: The software stack consists of user-level libraries to access the emulated NVM and Linux device drivers to manage the hardware resources.
5.2.1 **Hardware Components**

We implement two hardware components in the PL: a modified memory controller to emulate NVM latencies and Emulated Entry Unit to perform data pre-fetch operations. In this section, we describe the two hardware components.

*Emulated NVM*

To emulate long-latency, byte-addressable memory, we use the 1GB DDR3 memory connected to the FPGA fabric of the ZC706 kit. The Xilinx DDR3 memory controller IP connects the memory to the Zynq processing system via a general purpose AXI slave port. To modify access latency, we introduce a delay IP in-between which delays the AXI read and write response signals for configured duration. The delay IP allows the load and store latency to be configured from 145 ns - 5.2 $\mu$s and 70 ns - 5.2 $\mu$s respectively with a minimum resolution of 5 ns. The delay unit increases the latency by postponing the AXI Write and Read response signals.

*Emulated Entry Unit*

Implemented in the FPGA fabric is Emulated Entry Unit as shown in Figure 5.1. The hardware consists of a slave and master interface which connects to the Zynq PS AXI master and ACP slave ports respectively.

The ACP port allows the Emulated Entry Unit to perform cache coherent read and write operation. The FPGA logic in the Emulated Entry Unit dispatches and arbitrates read and write request using multiple control units as shown in Figure 5.1.

The ARM core communicates with the Emulated Entry Unit via AXI slave interface for configuring and issuing NBLD request via 2 sets of registers:

- **Operand registers**: Applications can issue a request to the Emulated Entry Unit by writing the callback function address, load address and callback handle to the operand registers. In addition, the emulation hardware requires the application to
specify a memory address for posting request completion.

- **Address Translation**: The Emulated Entry Unit uses the address translation registers to perform virtual to physical address translations. Address translation registers are accessible only by the operating system.

The implementation of Emulated Entry Unit differs from the AET described in Section 5.1 in two aspects. First, it needs to perform address translation within the table logic. Second, it relies on software polling mechanism to trigger the function execution. The following steps describe how our prototype implements `ASYNC.FETCH` operation. When an application writes to the operand register, the hardware first performs virtual to physical address translation of the NVM address and notification address. Second, it enqueues the translated addresses in a FIFO. A control logic called *dispatch unit*, then dequeues the request from the FIFO and performs the necessary fetch operation via ACP port. Once the ACP read request completes, it posts the completion message at the notification address. A user library is responsible for reading the status by polling the address and perform the necessary function call when the notification is available. Table 5.3 shows the resource utilization for the different hardware components implemented in the FPGA fabric. All user-logic in the FPGA runs at 200 MHz.

### 5.2.2 Software Components

The prototype software system consists of both kernel and user space components that help emulate non-blocking loads. The kernel components consist of a driver to manage the Emulated Entry Unit and configure the latency of the non-volatile memory. The system uses a user-space library, called *event-library* to interact with the Emulated Entry Unit and an NVM aware allocator to allocate data in the emulated NVM. Figure 5.2 shows the two software libraries and kernel modules that we describe in detail below.
**OS Driver**

The operating system is responsible for configuring the hardware resources - Emulated Entry Unit and emulated NVM controller. The driver exposes the memory via a simple character device that allows the application to map the 1 GB of emulated NVM directly into the user space. In practice, DAX [26] enabled systems can allow applications to map NVM to user-space via the file system interface.

Currently, our system supports a single linear address space mapping for the NVM region. When an application maps the character device, the driver updates the hardware address translation registers in the Emulated Entry Unit.

**User Space Libraries**

Easy programmer utilization of non-blocking load requires library/compiler support. For our prototype system we implement two libraries that support the instructions described in Section 5.1.

*LibNVM:* This is a generic user space library that allows the application to allocate NVM memory on the heap. The library extends the Intel PMEM [73] persistent memory programming library that provides support for mapping NVM regions into an application’s address space and provide a persistent memory allocator. We modify the Intel library to allocate data structures such as locks and allocation tables in the low latency ARM DRAM and only reference secondary data such as memory pool and per allocation header information in NVM. Our implementation of LibNVM uses conventional load/store instruction to access the meta-data stored in the memory and is independent of the instructions (conventional load or NBLD) the application uses to access the data.

*Event Library:* The event library is our prototype specific library which provides a simple interface for applications to issue and complete the instructions from Section 5.1.
Figure 5.3. Latency breakdown for 32 B fetch via ACP port: Our emulation setup adds a minimum of 300ns overhead for each request.

Table 5.3. FPGA resource utilization for Zynq SoC based architecture: LUT and BRAM utilization for Emulated Entry Unit and AXI delay unit

<table>
<thead>
<tr>
<th>Component</th>
<th>slice registers</th>
<th>LUT</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emulated Entry Unit</td>
<td>7257</td>
<td>2551</td>
<td>6</td>
</tr>
<tr>
<td>AXI Delay</td>
<td>1697</td>
<td>2837</td>
<td>-</td>
</tr>
</tbody>
</table>

The library provides 3 functionalities. First, it maps the Emulated Entry Unit registers for application level access. Second, it provides an interface for applications to issue \texttt{ASYNC.FETCH} instruction by writing to the AXI registers. Finally, it uses a per-cpu thread to poll the notification address for completion information and invoke the callback function specified by the application. A processor that implements NBLD does not require this library as the microarchitecture implements the logic to write to the hardware registers and execute call back routines.

5.3 Results

In this section, we evaluate the performance of NBLD operations for graph processing application.

First, we measure the overhead introduced by our prototype hardware to perform a single NBLD operation. We measure the hardware and software overheads in bare-metal mode. We set the emulated NVM read latency to $2 \mu s$ and configure the AET for
32 B fetch operations. Figure 5.3 provides the breakdown of latencies for an aligned request that misses the L2 cache. With the limited hardware access provided by the Zynq processor, we find that the asynchronous fetch incurs 300 ns overhead. Of which, nearly 100 ns is for issuing the request on the AXI port. However, this would be negligible for processors that integrate the AET alongside the CPU dispatch unit. Further, our prototype pays the unavoidable penalty to transfer the actual data via the ACP port even though the purpose of the request is to only fetch the data into the L2 cache. We expect the systems that implement NBLD to provide negligible latency overhead to fetch data compared to conventional access mechanisms. In order to understand the impact of NBLD on high-performance systems, we measure application level performance by scaling the latency and performance of the system to a processor and memory controller running at 3GHz.

**Application Performance - GraphChi**

We use GraphChi [51] a graph processing system and implement two graph applications: Pagerank and Alternating Least Squares (ALS) to measure application level performance. GraphChi is a disk-based graph processing system that extends the
GraphLab [57] system for a single machine. GraphChi system supports vertex-centric programming model where - the system iteratively executes a user-defined function over the vertices of the graph. GraphChi performs asynchronous and parallel updates on vertices where changes made are immediately visible to subsequent invocations of the user-defined function.

We modify the GraphChi system to load the vertex and the edge vectors from the input file onto the NVM memory. The PS DRAM contains all other data structures. We compare the application level metrics with three different configurations: baseline, software prefetch, and NBLD.

- **Baseline**: The baseline system corresponds to the algorithm that uses the GraphChi system as is without any optimizations.

- **Software prefetch**: In this mode, the algorithm issues a software prefetch instruction before accessing any data that resides in the NVM.

- **NBLD**: We modify the algorithm to utilize the Emulated Entry Unit to perform fetch operation.

We briefly describe the two graph algorithms and how we integrate NBLD operation in them.

*Pagerank* [71] extends the in-memory vertices model described in [51]. However, in our design, the in-memory vertices are stored in NVM and the update function at each iteration loads the edge and vertex values from NVM to compute the sum. The software prefetch design prefetches the vertex value from NVM for each incident edge of the vertex before computing the sum. NBLD based design uses the software interface to fetch the vertex value. The function (f) to compute sum gets invoked when the vertex
value is available in the L2 cache. The vertex update function waits for all calls to ’f’ to complete before updating the Pagerank for the vertex.

*Alternative Least Squares (ALS)* [103] algorithm is an extension of the collaborative algorithm described in [51] but uses the NVM memory to store not only the edges and vertices but also the 5-dimension latent vector. During each vertex update operation, the algorithm performs a regularized least square computation with the neighbors’ latent vectors and neighbors weight. For our software prefetch based design, the vertex update function first prefetches the neighbor’s weight and uses the weight value to perform the computation on the prefetched latent vector. In the NBLD based design, the vertex update functions issues `ASYNC.FETCH` to first read the neighbor’s weight. The callback function-’f’ then issues another `ASYNC.FETCH` operation to prefetch the latent vectors for the edge. Ordering `ASYNC.FETCH` operations in this way ensures that the regularized least square computation on the latent vectors uses the current neighbor’s weight value. Once all invocations to function-’f’ completes the vertex update function returns control to the GraphChi engine.

We use the web-BerkStan [54] dataset to evaluate the performance of Pagerank algorithm and run the GraphChi system for 6 iterations. For ALS we use a smaller subset of the Netflix dataset [67] and run the system for 3 iterations. For the two algorithms, we set the number of OpenMP threads to 2 and measure the following two metrics

- **Runtime**: This is the total time taken to perform the update operation on the vertices.

- **Stalls**: This is the percentage of CPU cycles for which the application stalled. This is a conservative metric as it includes the numbers of cycles the CPU stalled due to AXI transfers initiated by the event library.

Figure 5.4 shows the runtime for Pagerank and ALS algorithm when we increase
Figure 5.5. Emulated system CPU stall cycles for different operation mode: Shows the percentage of CPU cycles the system is stalled for Pagerank and ALS.

The NVM latency from 960 to 3800 CPU clock cycles. As expected, the runtime of baseline increases as the latency increases for both Pagerank and ALS. For Pagerank software prefetching does not provide any significant reduction in runtime. With NBLD however, the runtime is $2 \times$ to $6 \times$ times more than the baseline. We find this increase in runtime is due to two reasons. First, Pagerank issues NBLD operation for 4 bytes of data. The software and hardware overhead associated with emulation overshadows the benefits of background fetch operation. Second, the function-f only performs a simple add operation once the data becomes available resulting in the event-library’s polling thread keeping the CPU busy. For ALS however, we observe that software prefetch of latent vector data (40 B) reduces the runtime by 25%. Despite the overheads, NBLD reduces the runtime by nearly 40% for latencies 640 ns and above or for its target load latencies. We attribute this improvement to the availability of data in the L2 cache.

Figure 5.5 shows the percentage of CPU cycles for which the system is stalled. This is a conservative estimate as it also includes the cycles stalled due to NBLD AXI request. At lower read latencies (less than 800 ns) NBLD introduces significant stalls, due to the overheads in the emulation. However, at latencies above 800 ns we find that
for both Pagerank and ALS NBLD reduces the overall stalls - up to 19% for Pagerank and up to 17% for ALS.

5.4 Summary

This chapter described Non-Blocking Load, an instruction set and architecture supporting asynchronous memory access for high latency emerging memory technologies. NBLD triggers the execution of function when the data is available closer to the processor. It avoids the cost of software layers associated with conventional IO stack by allowing byte addressable, high latency NVMs to reside on the processor memory bus and at the same time reduces the CPU stalls associated with loads. We implemented a prototype system using the Zynq FPGA and evaluated the performance for graph processing applications. We found that despite the overheads of emulation, NBLD offers improvement in overall system performance while reducing the CPU stalls.

Acknowledgements

This chapter contains material from “Instruction Set and Micro-architecture Supporting Asynchronous Memory Access”, by Meenakshi Sundaram Bhaskaran, David Roberts, Elliot Mednick and Steven Swanson, which is currently being prepared for submission. The dissertation author was the first investigator and author of this paper.
Chapter 6

Summary

New memory technology such as NAND flash provides two orders of magnitude better performance than disk. Even faster-than-flash non-volatile memories are starting to become commercially available [60]. The emergence of new memory technologies provides the potential to vastly increase the IO subsystem performance. We began the dissertation by reviewing the characteristics of the emerging NVM and the potential for these memories to be available both in PCIe SSDs form factor and DIMM form factor. Fully utilizing the improved latency and bandwidth of emerging NVM requires re-visiting the hardware and software components.

We began the dissertation with a focus on immediate use case for emerging NVM SSDs — caches for a larger conventional disk-based storage system. We show that conventional caching systems designed for Flash-based SSDs introduce significant software overheads for servicing cache hits that overshadow all of the performance benefits from NVM SSDs. To mitigate the bottleneck, we presented Bankshot a system that bypassed OS and file systems when servicing cache hits. We also saw that a combination of hardware optimizations to Bankshot SSD and software optimizations to the kernel driver provided large gains in performance compared to a software-only optimization.

We then saw that for decades, computer systems have relied on the same block-
based interface to storage devices: reading and writing data from and to fixed-sized sectors. It was no accident that this interface was a perfect fit for hard disks, nor was it an accident that the interface has changed little since its creation. As other system components have gotten faster and more flexible, their interfaces have evolved to become more sophisticated and, in many cases, programmable. However, the poor performance of hard disks has meant that attempts to rethink how the system interacts with storage have had little overall impact on performance. In Chapter 4, we presented Willow, a system that makes programmability the central feature of the SSD.

We implemented two classes of application using the Willow prototype: Caching and Append Zero-Out that extended the interface to support caching and file system operations respectively. Willows RPC model of programming allowed us to implement the two applications in C within 1-month as compared to Bankshot’s year long hardware development efforts. Willow offers an incredibly flexible interface that application developers can make use of without much knowledge of hardware or SSD characteristics. Offloading the computation to the storage processor also enables Willow to free up the host CPU for other useful work.

While Bankshot and Willow provided low software overheads in accessing emerging NVM, the hardware overheads associated with PCIe based access overshadows the benefits of byte-addressable NVM. However, survey showed that as the capacity of the emerging NVM increased the access latencies also increased. In Chapter 5, we look at the micro-architectural challenges in attaching byte-addressable sub-μs NVM to the processor memory bus and accessing by load/store instructions. We focus on performance impact from long load latency and present Non-Blocking Load an instruction set extension to hide the latency. NBLD triggered the execution of application specific code once the data is resident in the cache.

Fully utilizing the improved latency and bandwidth characteristics of emerging
non-volatile memories required hardware, software and microarchitectural optimizations. We presented systems that address the challenges associated with three different use cases for emerging NVM.
Appendix A

Bankshot SSD Hardware

The PCIe SSD used for Bankshot emulation extends Moneta-D [13] by adding new hardware blocks for caching. One of the key hardware components highlighted in Chapter 3 was the cache controller. In this appendix we describe the design and construction of the cache controller.

A.1 Cache Controller

Moneta-D provides a virtualized interface for secure access of SSD contents directly from userspace. Figure 3.2 shows the internal construction of the SSD and the various components that enable the SSD to support direct user space access to cache. Of them, the cache controller implements the control logic necessary to implement the various commands that Bankshot introduces namely:

- **Cache Read**: Read the cached data from a given cache address.

- **Cache Clean Write**: Write data block to cache without setting the dirty bit. Only the cache manager in the operating system can use this command to perform a cache fill operation, i.e. transfer the data from backing store to cache upon a cache miss.
Figure A.1. Cache Controller: Each cache controller design stacks building blocks called *commandlet*, that allows the hardware to manage 8 GB of memory as block cache

- **Cache Dirty Write**: Write data block to cache and mark the corresponding dirty bit.

- **Cache Checkin and Cache Checkout**: The cache manager uses these commands to read, modify and write back the dirty status of cache contents.

Figure A.1 shows the internal architecture of cache controller. Each cache controller uses modular building blocks called *Memory Pipeline Unit* (MPU) each of which perform specific task. There are four MPUs and all MPU use a standard input and output interface enabling us to plug out modules at will and introduce new modules for hardware development and test. We first describe the design of generic FIFO interface followed by the description of each MPU.

### A.1.1 FIFO interface

The design of the FIFO interface enables decoupled control and data flow within the SSD and uses simple handshaking protocols to avoid introducing additional transfer latencies. In FPGA, we implement these interfaces as Xilinx FIFO IP for optimal placement and performance. Each FIFO interface consist of 4 sets of input and output interface: command interface, result interface, write interface and read interface.
Figure A.2. Memory Pipeline Unit Control Interface: The Memory Pipeline Unit uses the command and result FIFO for control flow

Command Interface  Figure A.2(a) defines the command interface. Each command stored in the command FIFO consist of 72bits of data. The lower 46 bits defines the 32-byte aligned memory address and the 8bit size determines the amount of 32-byte data to be accessed by the operation. Operations can be either read, write or custom defined by each MPU. In order to track status of each command issued through the command interface, the MPU uses an 8-bit tag field. One of the distinguishing property of the tag field is that the lower 3-bits defines the MPU ID. This allows each MPU to create its own sequence of tags for its own command.

Result Interface  Figure A.2(b) defines the result interface which allows a MPU to monitor the status of each command issued via the command interface. The fields Size and Tag are similar to those defined by the command interface while the Offset fields allows the sender of the result to process request in parts. One of the unique property of the result interface is that MPUs can terminate a command at point by setting the invalidate bit.

Read and Write Interface  Figure A.3 shows the data transfer interface used by the MPU. Besides supporting 128bit transfer, the interface allows read error propagation via the data bus. Errors in write are reported as a part of the result interface.
Figure A.3. Memory Pipeline Unit Data Interface: The data interface consists of read and write FIFOs that allow high bandwidth data transfer.

Figure A.4. Memory Pipeline Unit Ring Decode/Encode: Interface to the SSD ring network.
A.1.2 Ring MPU

We build basic modules for data transfer that allows us to test the cache controller independently. The ring is a special MPU as one side of its interface connects to the SSD’s internal ring network. Figure A.4 shows the construction of the ring MPU. For each incoming request on the RX interface, the MPU assigns a free tag and stores the packet headers in internal scoreboard. It then frames the command packet and sends it to the command FIFO interface and the data packet to the write FIFOs. Results are aggregated by the result parser and a single TX response is sent on the ring network for each RX command.

A.1.3 Merge/Split MPU

One of the challenges with the MPU design is that it needs to support each technology (DDR2, PCM etc.) read, write properties. In our prototype we used the Xilinx DDR2 ipcore that required the data transfer to be 32byte aligned and 32byte wide. To
Figure A.6. Memory Pipeline Unit Cache Control: The cache interface handles all cache specific commands.

perform such technology specific operations we use the Merge/Split MPU that allows us to break down the commands into smaller units and monitor their status. Further, the merge/split unit allows multiple down the pipe units processing request in parallel. In order to track the number of outstanding commands issued against a single tag, it uses a reference counter as shown in Figure A.5.

A.1.4 Cache MPU

The key component of the cache controller is the cache MPU which handles all the cache specific commands including dirty data tracking. The command parser in the cache MPU handles the previous described commands of the cache controller. In order to update the dirty bits and persist the dirty bit information by writing it back to a dedicated region in the memory the cache MPU implements a read/modify write engine. This engine issues special commands to read the meta-data from the DDR2 memory, apply the dirty bit changes and write back the data. Figure A.6 shows the design of the cache MPU and the different components.

A.2 Summary

Using the memory pipeline unit helped us design and test the cache controller and other components in a short span of time. One of the key characteristics of MPU is its ability to add new MPUs quickly. For example, we also use the same MPU framework.
in Willow SSD but replace the cache MPU with a simple read/modify write MPU to support unaligned read/write operations. MPU reduced both the development and test time required for building our SSDs.
Bibliography


[71] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.


