**The Pennsylvania State University**
**The Graduate School**

# RETHINKING THE MEMORY HIERARCHY DESIGN WITH NONVOLATILE MEMORY TECHNOLOGIES

A Dissertation in
Computer Science and Engineering
by
Jishen Zhao

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2014

The dissertation of Jishen Zhao was reviewed and approved* by the following:

Yuan Xie
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Mary Jane Irwin
Professor of Computer Science and Engineering

Vijaykrishnan Narayanan
Professor of Computer Science and Engineering

Zhiwen Liu
Associate Professor of Electrical Engineering

Onur Mutlu
Associate Professor of Electrical and Computer Engineering
Carnegie Mellon University
Special Member

Lee Coraor
Associate Professor of Computer Science and Engineering
Director of Academic Affairs

*Signatures are on file in the Graduate School.

# Abstract

The memory hierarchy, including processor caches and the main memory, is an important component of various computer systems. The memory hierarchy is becoming a fundamental performance and energy bottleneck, due to the widening gap between the increasing bandwidth and energy demands of modern applications and the limited performance and energy efficiency provided by traditional memory technologies. As a result, computer architects are facing significant challenges in developing high-performance, energy-efficient, and reliable memory hierarchies. New byte-addressable nonvolatile memories (NVRAMs) are emerging with unique properties that are likely to open doors to novel memory hierarchy designs to tackle the challenges. However, substantial advancements in redesigning the existing memory hierarchy organizations are needed to realize their full potential. This dissertation focuses on re-architecting the current memory hierarchy design with NVRAMs, producing high-performance, energy-efficient memory designs for both CPU and graphics processor (GPU) systems.

The first contribution of this dissertation is to devise a novel bandwidth-aware reconfigurable cache hierarchy with hybrid memory technologies to enhance system performance of chip-multiprocessors (CMPs). In CMP designs, limited memory bandwidth is a potential bottleneck of the system performance. NVRAMs promise high-bandwidth cache solutions for CMPs. We propose a bandwidth-aware reconfigurable cache hierarchy with hybrid memory technologies. With different memory technologies, our hybrid cache hierarchy design optimizes the peak bandwidth at each level of caches. Furthermore, we develop a reconfiguration mechanism to dynamically adapt the cache capacity of each level based on the predicted bandwidth demands of different applications.

This dissertation also explores energy-efficient graphics memory design for GPU systems. We develop a hybrid graphics memory architecture, employing NVRAMs and the traditional memory technology used in graphics memories, to improve the overall memory bandwidth and reduce the power dissipation of GPU systems. In addition, we design an adaptive data migration mechanism to further reduce graphics memory power dissipation without hurting GPU system performance. The data migration mechanism exploits various memory access patterns of workloads running on GPUs.

Finally, this dissertation discusses how to re-architect the current memory/storage stack with a persistent memory system to achieve efficient and reliable data movements. First, we propose a hardware-based, high-performance persistent memory design. Persistent memory allows in-memory persistent data objects to be updated at much higher throughput than using disks as persistent storage. Most previous persistent memory designs root from software perspective, and unfortunately reduce the system performance to roughly half that of a traditional memory system with no persistence support. One of the great challenges in this application class is therefore how to efficiently enable data persistence in memory. This dissertation proposes a persistent memory design that roots from hardware perspective, offering numerous practical advantages: a simple and intuitive abstract interface, microarchitecture-level optimizations, fast recovery from failures, and eliminating redundant writes to nonvolatile storage media. In addition, this dissertation presents a fair and high-performance memory scheduling for persistent memory systems. This dissertation tackles the problem raised by shared memory interface between memory accesses with and without the persistence requirement. This dissertation proposes a memory scheduling scheme that achieves both fair memory accesses and high system throughput for the co-running applications. Our key observation is that the write operations are also on the critical execution path for persistent applications. This dissertation introduces a new scheduling policy to balance the service of memory requests from various workloads, and a strided logging mechanism to accelerate the writes to persistent memory by augmenting their bank-level parallelism.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

First of all, I would like to express my sincere gratitude to my advisor, Professor Yuan Xie, for his guidance throughout my Ph.D. study. His helpful suggestions, important advice, and constant encouragement are of the most important factors that helped me grow as a researcher. I would also like to thank my committee members, Professor Mary Jane Irwin, Professor Vijaykrishnan Narayanan, Professor Onur Mutlu, and Professor Zhiwen Liu for their support, insightful comments, and valuable feedback.

I would like to thank my managers, Dr. Norm P. Jouppi, Dr. Partha Ranganathan, Dr. Jichuan Chang, and Mr. Cullen E. Bash, as well as my mentor, Dr. Sheng Li, for their great support and advice while I was an intern at Hewlett-Packard Labs during the past two years. I would also like to thank my collaborators at AMD and Intel, Dr. Gabrial H. Loh, Dr. Yen-Kuang Chen, Dr. Christopher Hughes, and Dr. Changkyu Kim, for their insightful discussions on various research projects.

Special thanks to my husband, Yu Sheng. His endless love, care, and encouragement have been and will always be my motivation for improvement. Yu has always pushed me to strive for the highest-quality research and papers, the best internships, and the best dissertation. I would like to thank my parents. Their persistent support and belief in me have been invaluable wealth.

I am also greatly indebted to past and present MDL members for providing a supportive and productive environment. Especially, I thank Xiaoxia Wu, Guangyu Sun, Yibo Chen, Xiangyu Dong, Jin Ouyang, Tao Zhang, Dimin Niu, Cong Xu, Jing Xie, Matt Poremba, Qiaosha Zou, Hsiang-Yun Cheng, Jue Wang, Jia Zhan, and Ping Chi for their help. Many thanks are to my friends and colleagues at AMD, Carnegie Mellon University and Hewlett-Packard Labs, including but not limited to Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Samira Khan, Yoongu Kim, Donghyuk Lee, Yixin Luo, Justin Meza, Gennady Pekhimenko, Vivek Seshadri, Lavanya Subramanian, Hongyi Xin, Hans J. Boehm, John Byrne, Dhruva Chakrabarti, Terence Kelly, Kevin Lim, Laura Ramirez, Rob Schreiber, Joseph

Tucek, Alistair Veitch, and and Doe Hyun Yoon.

Finally, I would like to thank Penn State, Hewlett-Packard, NSF, DOE, and Semiconductor Research Corporation for their generous funding supports during my Ph.D. study.

Chapter 1

# Introduction

The field of computer architecture has seen tremendous achievements. Processor designs have moved from the era of single core to multi-/many-core, preserving continuous system performance growth without significantly increased power budget and design complexity. Special-purpose hardware accelerators, e.g., GPUs and sensor arrays, were developed to support a single or narrow class of applications with much higher energy efficiency than general-purpose computers. However, two major challenges remain: the complexity and richness of application demands continue to overwhelm hardware capabilities; new device and circuit technologies are developed with properties that disrupt previous design assumptions, making current computer architecture designs suboptimal.

Among various components of computer systems, the memory hierarchy – including processor caches and the main memory – is a critical component and fundamental performance/energy bottleneck in addressing these two challenges. The memory hierarchy severs as the bridge between processors and the storage components (disks/flash). It stores applications' data sets being frequently accessed and supplies data to the processors. Modern applications rely on processing data from infinite sea of books, maps, photos, audios, videos, references, facts, and conversations. Consequently, their data sets can be gigabytes, terabytes, or even larger in size. Storing and processing such a large amount of data raises significant challenges in designing high-performance, energy-efficient memory hierarchy. Commodity memory technologies, such as SRAM and DRAM, are facing scalability challenges constrained by their device cell size and power consumption. Recently, various nonvolatile memories (NVRAMs), such as spin-transfer torque memory (STT-MRAM), phase-change memory (PCRAM), and resistive memory (ReRAM), have been studied as the replacement of traditional memory technologies used in the memory hierarchy [1, 2, 3, 4]. NVRAMs and traditional memory technologies trade off density, speed, power, and reliability. Furthermore, NVRAMs promise the game-changing feature – having both traditional memory (fast) and storage (nonvolatile, i.e., retaining data without power supply) properties in one device. Consequently, they yield abundant opportunities and challenges in memory hierarchy design.

This dissertation discusses the challenges and solutions of redesigning the memory hierarchy to achieve efficient and reliable data storage and movement, by leveraging NVRAMs. In particular, this dissertation presents (1) how to use NVRAMs as cache and main memory replacement to accommodate high-performance, energy-efficient data access in CPU and GPU systems and (2) how to re-architect the memory/storage stack to fully leverage NVRAMs' game-changing feature to achieve efficient and reliable data movement.

## 1.1 Challenges with Memory Hierarchy Design

This section describes the challenges in designing high-performance, energy-efficient memory hierarchies in CPU and GPU systems, respectively.

### 1.1.1 Performance Challenges with CPU Memory Hierarchy Design

One critical bottleneck for CPU performance scaling is the widening gap between the increasing bandwidth demand created by processor cores and the limited bandwidth provided by off-chip memories [5, 6, 7]. Due to such limitation, memory-demanding applications with a large working set spends additional cycles on off-chip memory accesses, and thus decreases the parallelism. In addition, even moderately memory-demanding applications will reach the bandwidth limitation as the number of cores scales up [8]. Consequently, memory bandwidth becomes one of the most important factors that influence high performance system design.

Various techniques can be found in recent computer systems and research work to address this issue. High performance computing systems such as NVIDIA's Tesla [9] rely on extremely high main memory bandwidth provided by graphics DDR (GDDR) memory to satisfy the demand of large number of processor cores. However, GDDR memory runs at high clock rates and consumes more power than conventional DRAM modules. It is undesirable for either general purpose or high-performance computing systems to improve their computing performance by simply sacrificing power efficiency.

Caching is known to be the most effective approach to reduce memory access latency. Proper cache hierarchy design can also help mitigate the increasing pressure to off-chip memory bandwidth. With an extensive study on limited pin bandwidth in multiprocessor systems, Burger *et al.* concluded that on-chip cache with more levels would improve the system performance [6]. Rogers *et al.* explored the requirements of on-chip cache hierarchy and optimization techniques due to scaling of processor core numbers [7]. Both studies show that exploring on-chip memory hierarchy in a manner focusing on bandwidth optimization will benefit future computing systems in terms of performance scaling. Recently, various studies have been performed on addressing the bandwidth problem by optimizing cache hierarchy designs. Yu *et al.* proposed a last level cache (LLC) partitioning algorithm to minimize bandwidth requirement to off-chip main memory [10]. Cache resources were allocated for the target workload in a way to reduce the overall system bandwidth requirement by considering memory bandwidth demand for each task. One key insight in their work was that cache miss rate information might severely misrepresent the actual bandwidth demand of a workload. Thus the overall system performance and power consumption might be inappropriately estimated. However, they only focused on LLC (L2 cache). In this dissertation, our design target is the overall on-chip memory hierarchy, which provides more design dimensions and flexibility. Furthermore, the cache partition in their study [10] was determined offline, and remained fixed during run-time. This dissertation proposes a reconfiguration mechanism to dynamically adapt the space of each cache level to the demand of different applications.

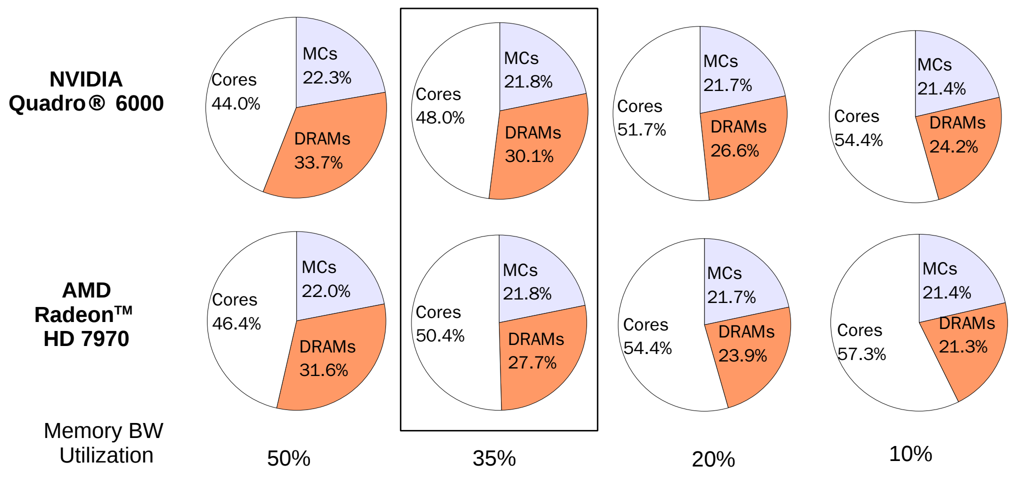


**Figure 1.1.** Power breakdown of both NVIDIA and AMD GPUs. Power consumptions of GPU cores and caches, memory controllers, and DRAMs are examined to show that off-chip DRAM accesses consume a significant portion of the total system power.

### 1.1.2 Energy Efficiency Challenge with GPU Memory Hierarchy Design

Modern GPU systems have become an attractive solution for both graphics and general purpose workloads that demand high computational performance. The graphics processing unit (GPU) exploits extreme multithreading to target high-throughput [11, 12]. For example, AMD Radeon$^{\text{TM}}$HD 7970 employs 20,480 threads interleaved across 32 compute units [11]. To accommodate such high-throughput demands, the power consumption of GPU systems continues to increase. As existing and future integrated systems become power limited, reducing system power consumption while maintaining high energy efficiency is a critical challenge for GPU system design.

To satisfy the demands of high-throughput computing, the GPUs require substantial amounts of memory (from hundreds of megabytes to gigabytes, which are usually off-chip memory) that can support a very large number of read and write accesses. Consequently, the off-chip memory consumes a significant portion of power in a GPU system. We evaluated the maximum power consumption of two GPU systems, AMD Radeon$^{\text{TM}}$HD 7970 [11], and NVIDIA Quadro®6000 [12], with the memory power model described in Section 6.7. Figure 1.1 presents the evaluated power distributions of GPU cores and caches, memory controllers, and off-chip memory. This dissertation consideres the memory bandwidth utilization (the fraction of all cycles when the data-bus is busy transferring data for reads or writes) from 10% to 50%. For both studied GPU systems, the off-chip memory consumes from 20% to over 30% of the total GPU system power. Note that for the workloads evaluated in this dissertation, the highest average bandwidth utilization observed was 35%. At this bandwidth level, the memory power consumption is 30.1% and 27.7% for the two GPU systems, respectively. If we can reduce the memory power by half, 12.5% of system power can be saved; this may seem like a relatively small amount, but it is in fact quite significant. For example, the maximum power consumption of AMD Radeon$^{\text{TM}}$HD 7970 [11] is 230W; therefore a 12.5% power reduction saves 29W. Therefore, techniques that reduce the graphics memory power requirements can be very effective at reducing the total system power.

Conventional graphics memories (GDDR) have employed several techniques to reduce memory power consumption. Using the pseudo-open drain (POD) signaling scheme [13] with on-die termination (ODT), static power is only consumed when driving a "low" signal and thus reduces the power of the memory interface. GDDR5, the latest generation of commercial graphics memory, provides further power savings compared to its predecessors with lower supply voltages, dynamic voltage and frequency (VF) scaling, and independent ODT strength control of address, command and data [13]. VF scaling techniques employed by conventional GDDRs reduce power by adapting the memory interface to the memory bandwidth requirements of an application. However, the power reduction comes at the expense of memory bandwidth degradation. For example, Elpida's GDDR5 memories are specified to operate over a large contiguous VF range to support data rates starting from as low as 800MB/s per channel to the maximum rate of 20GB/s. While 1.6GB/s per channel may be sufficient for displaying static images, a data rate of 6GB/s is required for playing high-definition (HD) video, and the maximum data rate of 20GB/s may be fully utilized by high-end gaming applications. For future high-performance GPGPU and advanced video processing (e.g., 3D HD multi-screen video games), existing power saving techniques may not suffice.

## 1.2 Design Opportunities Offered by NVRAMs

NVRAMs have several promising characteristics. They have much higher density than SRAM, as dense as DRAM, and need no refresh (which is required by DRAMs). In addition, they incorporate both memory and storage properties in a single device: can accommodate byte-addressable, fast memory accesses like memories; offer permanent data storage without power supply like disk and flash. These unique properties are likely to create an inflection point, opening doors to novel high-performance, energy-efficient memory hierarchy designs to unlock their full potential.

### 1.2.1 Opportunities in Replacing Traditional Memory Technologies

NVRAMs promise much lower leakage power than SRAM, which is the traditional memory technology used in the cache hierarchy. Therefore, we can save substantial portion of cache power by adopting NVRAM-based caches in CPUs [2]. Furthermore, we find that NVRAMs can improve cache performance at large capacities. When the cache capacity is small, SRAM has much lower latency, and therefore higher memory bandwidth, than NVRAMs. However, we find that NVRAMs can provide higher memory bandwidth than SRAM at large cache capacities [14]. As a result, using NVRAMs as the memory technology of larger caches can offer much higher system performance than employing pure SRAM-based caches.

NVRAMs also bring in opportunities in designing energy-efficient graphics memories in GPUs. First, NVRAMs do not require refresh operations, which can consume a large portion of memory power [15]. Furthermore, we can shut down NVRAMs, when they are not being accessed. In particular, we can effectively reduce graphics memory power consumption without hurting GPU system performance, by leveraging unique the data access patterns of the workloads running on GPUs [16].

### 1.2.2 Opportunities in Re-architecting the Memory/Storage Stack

For decades, computer systems adopt a two-level storage model to manipulate data access: a fast, volatile memory updated by loads and stores, with data being lost when system halts or reboots; a slow, nonvolatile storage device managed by databases or file systems, while data can survive across system boots. With the game-changing feature, NVRAMs can enrich such two-level memory/storage stack with the capability of accommodating fast accesses to permanent data storage in a unified nonvolatile memory. This feature brings new opportunities to address the massive online data storage and processing requirements of “big data” applications, allowing them to directly access permanent data storage in memory without the performance and energy overheads of transferring data from/to storage. Unfortunately, current hardware and hardware/software interface are optimized for the two-level memory/storage stack with vastly discrepant speed (fast memory and slow storage), interfaces (memory buses and storage I/Os), and functions (hardware-accelerated memory access and software managed permanent data storage). Consequently, computer architects need to redesign the memory/storage stack to unify the two functions through a memory interface with optimized system performance and reliability.

## 1.3 Solutions

The goal of this dissertation is to design high-performance, energy-efficient memory hierarchies for CPU and GPU systems, by fully leveraging NVRAM’s properties and benefits. This dissertation proposes the following three solutions to achieve this goal.

- The first solution is a novel bandwidth-aware reconfigurable cache hierarchy [14] to enhance system performance of chip-multiprocessors (CMPs) with hybrid memory technologies, by leveraging NVRAM’s performance benefits. The hybrid cache hierarchy maximizes the provided bandwidth of processor caches and minimizes the bandwidth demand to the off-chip main memory.

We also dynamically reconfigure the hybrid cache hierarchy to accommodate the actively changing bandwidth demands of various applications.

- The second solution is an energy-efficient graphics memory design [16], leveraging NVRAM's energy benefits to replace traditional pure DRAM-based graphics memories. We propose a hybrid graphics memory, mixing DRAM and various types of NVRAMs. The hybrid graphics memory can provide higher memory bandwidth and consumes less power than the traditional graphics memory designs. Although NVRAMs have longer latency than DRAM, our design can hide such longer latency with an adaptive data migration mechanism, by leveraging the memory access patterns of various workloads running on GPUs.

- The third solution of this dissertation leverages the disruptive property of NVRAMs to develop a persistent memory [17], incorporating the functions of memory and storage. Our design allows a persistent memory system to directly update the real in-memory data structures at high throughput.

- The final solution of this dissertation designs a memory scheduling scheme that achieves both fair memory access and high system throughput in persistent memory systems.

## 1.4 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 describes the background and the related work in NVRAM technologies, persistent memory, and high-performance, energy-efficient memory hierarchy designs. Chapter 3 presents a CPU cache hierarchy design with optimized memory bandwidth, utilizing hybrid memory technologies. Chapter 4 presents graphics designs with NVRAMs, optimized for system throughput and energy efficiency. Chapter 5 and Chapter 6 present our persistent memory designs, re-architecting the memory/storage stack by incorporating storage functions on maintaining data persistence into the memory system. In particular, Chapter 5 describes our hardware-based persistent memory design, which achieves both persistence and high performance in memory systems. Chapter 6 identifies the performance issue of resource competition at the shared memory interface of persistent memory, and tackles the problem by redesigning memory scheduling mechanisms. Finally, Chapter 7 concludes the dissertation by summarizing the key results and insights that have been presented, and presenting suggestions on future research directions concerned with NVRAM's implications on novel memory hierarchy design.

Chapter 2

# Background and Related Work

This chapter first describes the background of emerging technologies, including 3D/2.5D integration and NVRAMs. This is followed by the description of background of the emerging persistent memory technique that leverages the nontraditional property of NVRAMs. Finally, this chapter describes the relevant approaches classified into two categories: replacement of traditional memory hierarchies and the persistent memory by re-architecting the memory/storage stack.

## 2.1 Background of NVRAM Technologies

This dissertation explores memory hierarchy designs with three types of NVRAMs – STT-MRAM, ReRAM, and PCRAM [18, 19, 20, 21]. Note that some studies refer nonvolatile memories as flash memories [22], which are block addressable. This dissertation investigates the use of byte-addressable nonvolatile memories. Therefore, flash memories are not discussed as NVRAMs in our work.

**Various types of NVRAM technologies:** STT-MRAM is the latest generation of magnetic RAM (MRAM) [19, 20]. STT-MRAM employs Magnetic Tunnel Junction (MTJ), which contains two ferromagnetic layers and one tunnel barrier layer, as its binary storage. The relative magnetization direction of two ferromagnetic layers determines the resistance of MTJ. If the two ferromagnetic layers appear at the same directions, the resistance of MTJ is low, indicating a "0" state; otherwise, the resistance of MTJ is high, indicating a "1" state. In ReRAM [23, 24, 25], a normally insulating dielectric is conducted through a filament or conduction path generated by applying a sufficiently high voltage. The conduction path can be generated by different mechanisms, including defects, metal migration, etc. The filament may be reset (broken, resulting in high resistance) or set (re-formed, resulting in lower resistance) by applying an appropriate voltage. PCRAM [21, 26] uses chalcogenide-based material to storage information. The chalcogenide-based material can be switched between a crystalline phase (SET or "1" state) and an amorphous phase (RESET or "0" state) with the application of heat.

**Benefits:** NVRAMs promise density, performance, and energy benefits. They have much higher density than SRAM, the memory technology widely used in commodity processor caches. Compared to DRAM, NVRAMs show significant power benefits. Due to the non-volatility, NVMs do not require refresh operations, and have near-zero standby power. In addition, NVRAMs can accommodate both byte-addressable, fast data accesses and nonvolatile data storage. Therefore, they incorporate both memory (fast) and storage (retaining data without power supply) properties in one device. This feature can disrupt current two-level memory/storage stack with the capability of accommodating fast accesses to permanent data storage in a unified nonvolatile memory.

**Drawbacks:** The drawbacks of NVRAMs include long write latency, high write energy, and low write endurance. Write endurance is the number of times that a memory cell can be overwritten before the cell fails. Among the three NVRAMs, only STT-MRAM is free from the low write endurance issue. The endurance of STT-MRAM is larger than $10^{15}$ [1], which is close to that of SRAM. The endurance of ReRAM is in the range of $10^5$ to $10^{11}$ [27, 28, 29]. That of PCRAM is in the range of $10^5$ to $10^9$ [30]. Therefore, STT-MRAM is a practical solution for cache design [31, 32, 33], while ReRAM is feasible for last-level cache (LLC) with low write intensity. The low endurance of PCRAM makes it less feasible to be used as processor caches. All the three NVRAMs are feasible for designing the main memory, especially with architecture-level error protection techniques, such as ECP [34], dynamically replicated memory [35], SAFER [36], start-gap [3], security refresh [37], and FREE-p [38]. Furthermore, a projected plan by ITRS [39] highlighted that the endurance of PCRAM and ReRAM will be at the order of $10^{15}$ or higher by 2024.

**On-chip and off-chip memory implementations:** In principle, we can implement both on-chip and off-chip components of the memory hierarchy with various types of NVRAMs. For example, we can implement the off-chip NVRAM main memory as dual in-line memory modules (DIMMs), which is compatible to the commodity off-chip DRAM implementations [40]. Most NVRAM technologies are not compatible CMOS technology, which is the traditional technology used to implement the processor cores and caches. Consequently, with most types of NVRAMs, we need to implement on-chip NVRAM caches and memories by leveraging silicon interposer [41] or 3D stacking [42, 43, 44] technologies. With the silicon interposer technology, we can package CMOS components and NVRAM components side-by-side in a single chip. This technology has been widely explored by academia and industry to develop high-performance system-in-package designs [41, 45]. With 3D stacking technology, we can implement these components on separate dies and vertically stack the dies on top of each other [46, 47, 48, 44, 49, 50, 51, 52, 53, 54, 55]. Samsung recently announced a 3D-stacked wide-I/O DRAM targeting mobile systems [51]. The presented two-layer DRAM with four 128-bit wide buses has 12.8GB/s peak bandwidth, 2Gb of capacity, and only 330.6mW of power consumption. Woo *et al.* re-architected the memory hierarchy, including the L2 cache and DRAM interface, and take full advantage of the massive bandwidth provided by stacking the DRAMs on top of processor cores. Tezzaron corporation has implemented true 3D DRAMs, where the individual bitcell arrays are stacked in a 3D fashion [53]. The peripheral control logic and circuitry are placed on a separate, dedicated layer, incorporated with different process technologies. Micron developed a Hybrid Memory Cube (HMC) [52] that combines high-speed logic process technology with a stack of through-silicon via (TSV) bonded memory die. HMC increases density per bit and reduces the overall package form factor. Recently, AMD and SK hynix announced joint development of high bandwidth memory (HBM) stacks [56], which leverages TSV and wide I/O technology and conforms JEDEC HBM standardization [57]. Recently, Lin et al. demonstrated a CMOS-compatible STT-MRAM implementation [58], which allows STT-MRAM based memory components to be directly integrated with processor cores on the same die.

## 2.2 Background of Persistent Memory

Persistence has been well investigated in databases and file systems. We borrow the concept of atomicity, consistency, isolation, and durability (ACID) [59] from the database community to study the properties of persistent memory. These four properties can be separately maintained in different manners in a system. For example, transactional memories (TMs) [60] maintain A, C, and I, separated from D, while a recent study on failure-atomic msync [61] focuses on A and D.

In particular, a persistent memory system needs to ensure atomicity, consistency, and durability. First of all, a persistent memory system contains non-volatile devices so each data update is retained during power loss, crashes, or errors. This is referred to as the **durability** property. Second, because the granularity of programmer-defined data updates can be larger than the interface width of the persistent memory, a single update is typically serviced as multiple requests. Therefore, sudden power losses or crashes can leave an update partially completed, corrupting the persistent data structures. To address this issue, each single update must be "all or nothing", i.e., either successfully completes or fails completely with the data in persistent memory intact. This property is **atomicity**. Third, **consistency** requires each update to convert persistent data from one consistent state to another. Taking an example where an application inserts a node to a linked list stored in persistent memory, a system (including software programs and hardware) needs to ensure that the initial values of the node are written into the persistent memory before updating the pointers in the list. Otherwise, the persistent data structure can lose consistency with dangling pointers in a sudden crash, leading to a permanent corruption not recoverable by restarting the application or the system. Typically, programmers are responsible for defining consistent data updates, because only the programmers know what it means for application data to be in harmony with itself. Of course, programmers can leverage runtime API to do this. While executing the software programs, hardware and system software need to preserve the demanded consistency.

The fourth property, **isolation**, ensures that concurrent data updates are invisible to each other. Today, a programmer writing portable code atop a POSIX-compliant OS and hardware has two separate families of mechanisms for solving two isolation problems. One family of mechanisms is used to ensure orderly race-free access to data in multithreaded or multiprocess concurrent programs. This set of mechanisms includes mutexes, semaphores, TMs, and lock-free/wait-free data structures and algorithms. The other family of mechanisms is used to update data in durable media. This set of mechanisms includes system calls such as `write()`, `fsync()`, and `mmap()/msync()`. Commodity systems use separate and orthogonal mechanisms for handling isolation in the face of concurrency and durable updates. Our persistent memory design permits the same kind of orthogonal separation of concerns. Various concurrency control mechanisms can be integrated with our design.

Specifically, our persistent memory design maintains A and D, preserves C that is defined by programmers, and relies on concurrency control mechanisms to support isolation.

## 2.3 Related Research on Replacing Traditional Memory Hierarchies

A large body of recent studies focused on exploring the emerging technologies as the replacement of traditional memory design technologies in existing memory hierarchies [2, 4, 1, 62, 63, 64, 65, 66, 67]. These studies endeavor to balance between latency, bandwidth, and cost with emerging technologies. Various NVRAM technologies, including STT-MRAM, PCRAM, and ReRAM, were explored as the memory technologies of processor caches and main memory to improve system performance and energy efficiency [2, 4, 1]. NVRAMs typically impose longer write latency and higher write energy than traditional SRAM and DRAM. Therefore, it is unlikely that NVRAMs will completely replace traditional memory technologies in the near future. Hybrid memory technologies [1] will remain as a promising memory system solution. Most previous studies on hybrid memories focused on reducing the latency gap between the last level cache and the main memory [2, 1]. However, almost none of these work addressed the bandwidth bottleneck issue.

**Table 2.1.** Comparison of Kiln with previous work. ($\star$ means In-place updates are only performed for memory stores to a single variable or at the granularity of the bus width. $\diamond$ means ordering is maintained among the writes to the disk or flash by flush or checkpointing.)

| **Designs** | **Mechanisms** | | | | | **Persistence Support** | |
|---|---|---|---|---|---|---|---|
| | In-place | Logging | COW | clflush/ msync/ fsync | mfence/ barrier | Atomicity | Ordering |
| BPFS [68] | $\star$ | **No** | Yes | **No** | Yes | $\surd$ | $\surd$ |
| Mnemosyne [69] | $\star$ | Yes | Yes | Yes | Yes | $\surd$ | $\surd$ |
| NV-heaps [70] | No | Yes | **No** | **No** | Yes | $\surd$ | $\surd$ |
| CDDS [71] | No | **No** | Yes | Yes | Yes | $\surd$ | $\surd$ |
| UBJ [72] | **Yes** | Yes | Yes | $\diamond$ | $\diamond$ | $\surd$ | $\surd$ |
| eNVy [73] | No | **No** | Yes | $\diamond$ | $\diamond$ | $\surd$ | $\surd$ |
| Native System | **Yes** | **No** | **No** | **No** | **No** | $\times$ | $\times$ |
| Kiln | **Yes** | **No** | **No** | **No** | **No** | $\surd$ | $\surd$ |

## 2.4 Related Research on Persistent Memory

Protecting data against system failures and crashes forces a trade-off between performance and reliability. This section investigates the persistence mechanisms in previous work.

### 2.4.1 Maintaining Atomicity by Multiversioning

Multiversioning is a common method to ensure atomicity. With multiversioning, multiple copies of data exist. When performing updates to one copy of data, another copy is left intact. If one copy of data is corrupted by a partial update, another copy is still valid and available for recovery.

Most previous work on persistence, e.g., persistent object systems [74, 75, 76, 77, 78, 79, 70], the Java persistence API [80, 81], RVM [82], Rio file cache [83], Stasis [84], Mnemosyne [69], eNVy [73], and UBJ [72], employ one of two techniques to maintain multiversioning: write-ahead logging (or journaling) [85, 86, 69, 70, 82, 87, 88] or COW [89, 73, 83, 68, 71]. Several previous studies investigated the use of battery-backed RAMs as persistent storage [90, 91, 92]. Although battery-backed RAMs are byte-addressable, these designs inefficiently access the RAMs through a driver like disks and adopt database management systems (DBMS) or file systems to implement logging or COW to manage the persistent memory. NV-heaps [70] and Mnemosyne [69] adopt durable software transactional memory (STM) to support persistence for in-memory data objects. Both designs enforce atomic transactional updates by maintaining a redo log.

Both logging and COW mechanisms impose significant performance overhead by explicitly executing logging or data copying instructions. While the software overhead is tolerable with traditional disk-based persistent memories where the I/O delay dominates the performance overhead, the fraction of software overhead increases dramatically when the persistent memory can be accessed at a much faster speed [93]. Furthermore, duplicated data (logs or data copies) traverse the cache hierarchy to the memory, contaminating caches with non-reusable cache lines. Therefore, the key reason that the native system runs fast is that it performs in-place updates to the real in-memory data, without explicitly duplicating the data like logging or COW does. However, in-place updates are hard to implement in most previous NVRAM-based persistent memory designs [69, 70, 68, 71], which maintain persistence in a single-level memory. In such systems, at least one more copy of data needs to be stored in addition to the real data, to maintain multiversioning.

An exception of ensuring atomicity without multiversioning is when an update can be completed instantaneously, typically with very small granularity of memory stores. Examples of such cases are updating a single variable [69] or a memory store of the granularity the same as the bus width [68, 94]. Unfortunately, these studies do not provide any mechanisms that can be applied to in-place updates of larger granularities.

### 2.4.2 Preserving Consistency by Ordering

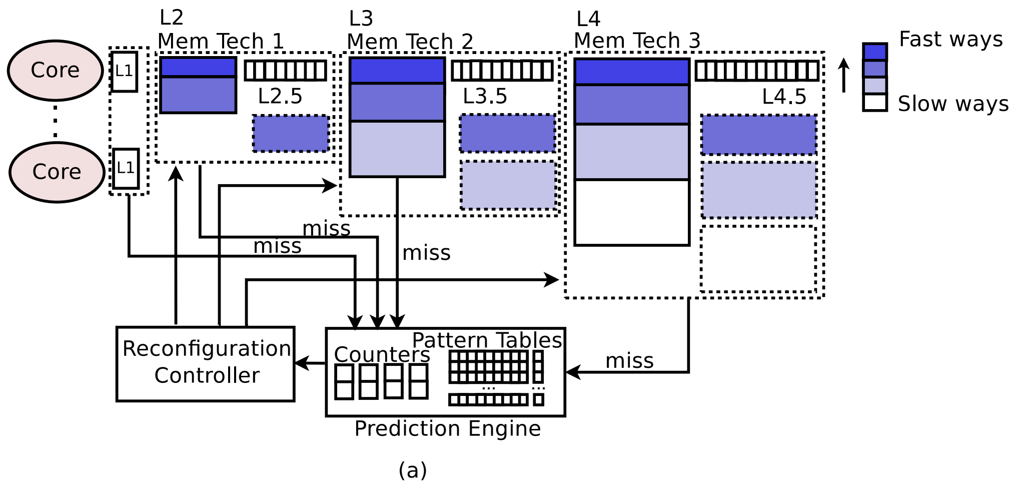
Controlling write ordering is a primary mechanism to preserve consistency in application programs. Ordering means that the order that updates become permanent must match the order in which they are issued. A mismatch can happen when processor caches and memory controllers reorder memory requests to optimize performance. A persistent memory employs ordering control mechanisms to prevent mismatch. Most previous persistent memory designs ensure the ordering by write-through caching [69] or bypassing the processor caches entirely, flush, memory fence [69, 71, 94], and msync operations, each imposing high performance costs. With write-through caching, each memory store needs to wait until reaching the main memory. Flush and memory fence mechanisms can cause a burst of memory traffic and block subsequent memory stores. Furthermore, most previous designs [69, 71, 94] employ instructions such as `clflush`, which flushes dirty cache lines to ensure ordering, with a latency that can be up to several milliseconds. Besides the long latency, flushing an entire cache can also evict the working sets of other applications from the cache. BPFS [68] adopted an epoch barrier mechanism to minimize the flush traffic, however at the cost of reduced durability strength that leads to potential data loss.

Chapter 3

# BARCH: Bandwidth-aware Hybrid Cache Hierarchy Design for CMPs

While many design methods involved with new memory technologies endeavor to reduce the off-chip memory access latency, our work focuses on reducing off-chip bandwidth demand by employing hybrid on-chip memory hierarchy and reconfiguration. Figure 3.1(a) depicts an overview of our BARCH design. We examine different memory technologies in terms of read and write access latencies, dynamical energy, and bandwidth under dynamic energy constraint. The bandwidth-capacity curves of different memory technologies are shown in Figure 3.1(b). In a given range of capacities, one memory technology may provide the highest bandwidth. However, we cannot find a single memory technology that always maintain the highest bandwidth over the whole range of capacities. Based on this observation, we employ hybrid memory technologies in BARCH. At each level of cache, we select the memory technology which provides the highest bandwidth within a specific capacity range. The overall bandwidth-capacity curve of BARCH is maintained to be the highest across the whole capacity range, shown in the solid curve in Figure 3.1(b).

In addition, we dynamically adapt the cache hierarchy according to the bandwidth demand of different applications. The total cache space at each level is partitioned to a set of fast ways and slow ways. During run-time, we examine the bandwidth demand of individual application at each execution time interval. Cache space at each level is tailored according to the bandwidth demand. In order



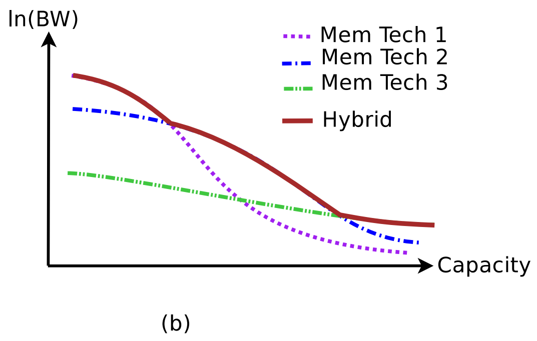


**Figure 3.1.** Overview of the hardware configuration. (a) Configuration of reconfigurable hybrid cache hierarchy. (b) The overall bandwidth-capacity curve of the hybrid cache hierarchy (a generic case of Figure 3.3).

to facilitate the reconfiguration, we design a statistical prediction engine to collect the bandwidth demands of applications at the end of each execution time interval, and predict the bandwidth demands for the next time interval. Rather than conventional last value or history table based predictors, we present a probability-based statistical predictor which can achieve high accuracy with small performance and area overhead.

## 3.1 Latency, Energy, and Bandwidth of Various Memory Technologies

First of all, we examine the latency, dynamic energy, and bandwidth of different memory technologies, including SRAM, STT-MRAM, ReRAM, and eDRAM. Be-

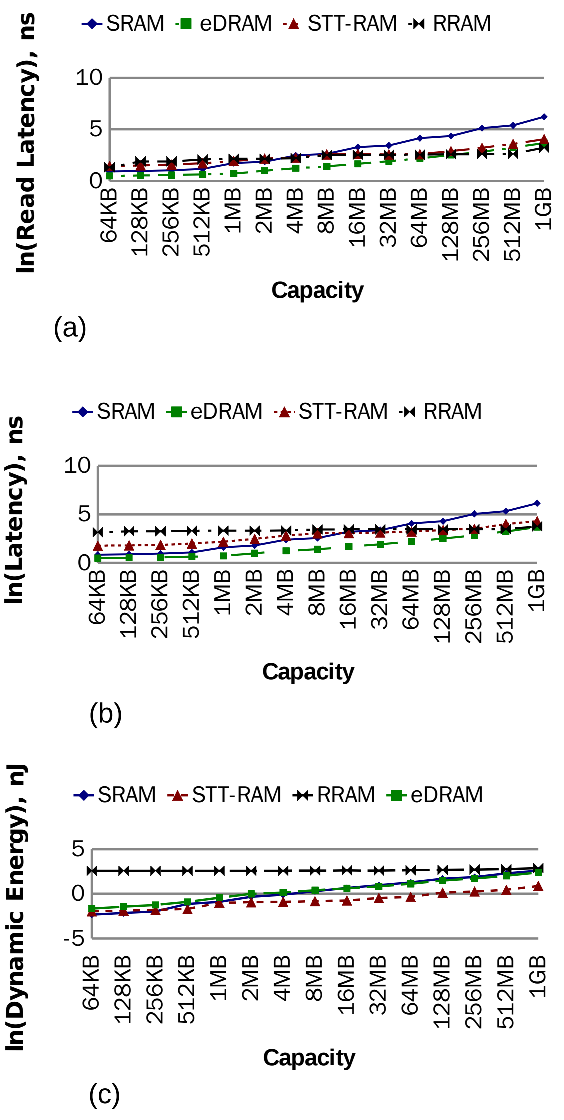


**Figure 3.2.** Latency, bandwidth, and dynamic energy of different memory technologies. (a) Read latency. (b) Latency with 40% of write intensity. (c) Dynamic energy with 40% of write intensity.

cause PCRAM has serious endurance issues, we do not consider it as an on-chip memory candidate. We use NVSim [95], a circuit level performance, energy, and area estimation tool, to evaluate different memory technologies.

**Latency:** The read and write latencies of the two NVRAMs, STT-MRAM and ReRAM, are asymmetric. The write latency is much higher than read. Therefore, we consider the read and write latencies separately. The read latency ($d_r$) is evaluated using the following equation:

$$d_r = d_{Hti} + d_{wl} + d_{bl} + d_{comp} + d_{Hto} \tag{3.1}$$

where $d_{Hti}$ and $d_{Hto}$ are H-tree input and output delays that determined by the RC delay of global wires, $d_{wl}$ is decoder and word-line delay, $d_{bl}$ is bit-line and sense amplifier delay, $d_{comp}$ is comparator delay related to the read noise margin of memory cell that is affected by off/on resistance ratio, and $d_{Hto}$ is H-tree output delay. Figure 3.2(a) illustrates the read latency of different memories as a function of memory capacity with both x- and y-values in *log* scale. Sensing delay dominates the read latency of the two NVRAMs at small capacities. Therefore, STT-MRAM and ReRAM do not show any advantages in read latency. When H-tree delay unveils at large capacities, ReRAM (with the smallest cell size) becomes faster than other memory technologies. The read latency of SRAM will increase rapidly after 128MB due to the large area. The write latency of NVRAMs is dominated by the write pulse width. We evaluate the write pulse of 10ns, 20ns, and 100ns for STT-MRAM and ReRAM. When the cache size is small (less than 4MB), the write latency of the two NVRAMs are much higher than SRAM and eDRAM. As the capacity grows to larger than 128MB, the write latency of SRAM becomes higher than the NVRAMs again due to the large area. Fortunately, the write intensity of most applications is lower than 40%. We inject 40% of write intensity, and obtain the latency curves of different memory technologies as shown in Figure 3.2(b). The curves meet each other at different capacities. The key observation is that the latency benefit of STT-MRAM, eDRAM and ReRAM starts to show at large capacities. We examine the latency curves with other write intensities, and observe similar pattern.

**Dynamic energy:** Figure 3.2(c) demonstrates the dynamic energy of different memory technologies with 40% write intensity. The first crossing point locates between SRAM and STT-MRAM at the capacity around 2MB. STT-MRAM consumes lower dynamic energy than SRAM after this cross point. The curves of SRAM and eDRAM cross at around the capacity of 16MB. The dynamic energy of ReRAM keeps high until hits the curve of the SRAM at the capacity of 1GB.

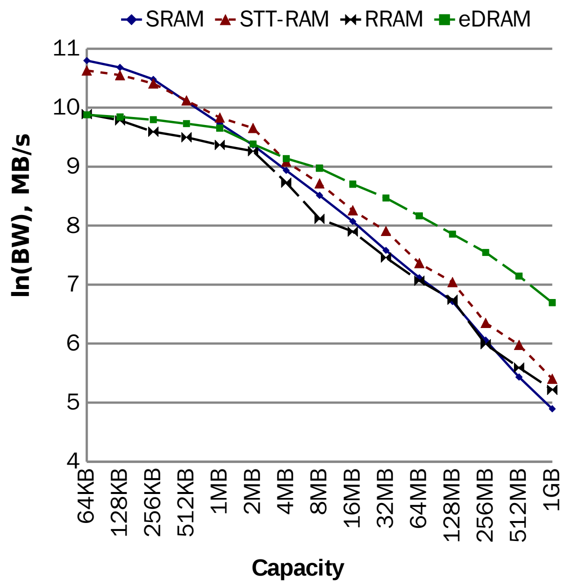


**Figure 3.3.** Bandwidth-capacity curves of different memory technologies under dynamic energy constraint (with 40% of write intensity).

**Bandwidth under energy constraint** We estimate the read and write bandwidths that can be provided by different memory technologies based on our latency and dynamic energy evaluations. The access power of a cache is approximately proportional to $bandwidth \times \sqrt{capacity}$ [96]. Figure 3.3 shows the bandwidth curves estimated under the energy constraint based on this relationship. In this figure, the curves meet each other at different memory capacities. For example, the curves of SRAM and STT-MRAM cross each other at around 2MB, and eDRAM provides the highest bandwidth after 16MB. Therefore, a single technology can provide the highest bandwidth within a given range of memory capacity. Based on our evaluation of latency, energy, and bandwidth of different memory technologies, we select SRAM, STT-MRAM, and eDRAM to construct our hybrid cache hierarchy. We discard ReRAM due to its high dynamic energy and low endurance.

## 3.2 Hybrid Cache Hierarchy

The goal of our hybrid cache hierarchy design is to leverage different memory technologies to configure an on-chip memory system with optimal available bandwidth at each level with various capacities. The on-chip memory system will therefore always keep high bandwidth over the whole range of capacities.

As shown in Figure 3.1(a), the baseline CMP system consists of multiple cores, where the L1 caches are private to each core and the lower level caches are shared by the cores. With the bandwidth-capacity curves of various memory technologies, we can optimize the bandwidth provided by the cache hierarchy with hybrid memory technologies. To achieve this goal, we configure the cache hierarchy based on the following factors.

**Number of levels:** Figure 3.1(b) is a sketch of Figure 3.3. In our case, "Mem Tech" 1, 2, and 3 are SRAM, STT-MRAM, and eDRAM respectively. We can observe two crossing points (SRAM and STT-MRAM, STT-MRAM and eDRAM) of the bandwidth-capacity curves, dividing the capacity range into three regions. Based on this observation, we configure the shared cache hierarchy as three levels. Each level of cache is implemented with the memory technology that provides the highest bandwidth in a specific capacity range. As a result, the overall bandwidth-capacity curve of the shared cache hierarchy is the solid curve ("Hybrid") in Figure 3.1(b).

**Memory technology of each level:** At each cache level, we select the memory technology providing the highest bandwidth within the range of capacities between the two crossing points in bandwidth-capacity curve. In our case, SRAM, STT-MRAM, and eDRAM are selected as the L2, L3, and L4 caches respectively.

**Capacity of each level:** The total capacity of each level is determined by the crossing point of the bandwidth curve of two memory technologies. For example, the overall capacity of SRAM/L2 is 2MB, since the curves of SRAM and STT-MRAM meet between the capacities of 2MB and 4MB. The total capacity of STT-MRAM/L3 is 16MB. We limit the capacity of eDRAM/L4 to be 64MB to avoid high area and energy overhead. Each level of cache is configured to be multiple banks the same way as the conventional cache design.

## 3.3 Reconfiguration

Although the above hybrid memory configuration maintains the optimal provided bandwidth over the whole range of capacities, it does not guarantee the best performance of different applications with a variety of bandwidth demands. Smaller caches provide higher bandwidth. However, the smaller the capacity, the less proportion of the working set can be fit into such limited cache space. As a result, the application may create very high bandwidth demand to the next level of cache. Consequently, we reconfigure the each level of caches at run-time adaptive to the bandwidth demands and the working set sizes of different applications, and balance the available and demanded bandwidth at each cache level.

In order to reconfigure the cache spaces, we further divide the overall cache space at each level into a set of fast ways and slow ways, which are defined as "partitions". The faster partitions will provide higher bandwidth, but smaller capacities. During system initialization, we configure the cache system to provide the highest available bandwidth. Only the fastest partitions are activated. The rest of the cache space is sent into drowsy state [97]. During run-time, we re-adjust the cache capacities, and activate the slower partitions according to the demand bandwidth of specific applications. The bandwidth-capacity curve of the hybrid cache hierarchy appears to be monotonically decreasing as depicted in Figure 3.1(b). At a specific time point, the demanding bandwidth of an application at each cache level can be mapped to a single point on the curve. Accordingly, we can reconfigure each level of cache to the available size that is the closest to the capacity point corresponding to the demanded bandwidth.

Reconfiguration is applied at the end of each evaluation time interval. The length of the time interval can be fixed, or depends on the operating system context switch. At the end of a time interval, we determine the upper bound of the capacity at cache level-i ($s_i^u$) by mapping the demand bandwidth (DBW) of a specific application to the hybrid cache hierarchy's bandwidth-capacity curve, i.e., $s_i^u = f^{-1}(\text{DBW}_i)$ where $f(x)$ represents the bandwidth-capacity relationship of the hybrid cache hierarchy. DBW is measured by miss per second at previous level of cache, i.e., $C_m B_l / t$ where $C_m$ and $B_l$ are number of cache miss and cache line size respectively. $\text{DBW}_i$ is generated using the prediction engine, which will be presented in section 3.4. In theory, higher bandwidth provided by the memory system increases both the throughput and power consumption of computing systems. Therefore, we define a lower bound to the capacity of cache level-i as $s_i^l = f^{-1}(\mathrm{DBW}_i * (1 + \sigma))$, where $\sigma$ is a pre-defined threshold to constrain the provided bandwidth with limited power overhead. The capacity of cache level-i ($s_i$) is thus selected in the range of $s_i^l \leq s_i \leq s_i^u$. In addition, one or more partitions at cache level-i can be configured to become level-i.5 as shown in Figure 3.1(a). This may happen when the demand capacity at cache level-(i+1) is detected to be smaller than the available free space at level-i. In this case, the primary miss path at level-i is re-directed to level-i.5.

Our reconfigurable design exploits set associativity in conventional cache organizations. One merit of such design is the trivial modification to existing cache architecture, since the division of ways already presents in a conventional cache organization. An n-way set associative cache consists of $n$ data and tag arrays. We divide the each level of cache into partitions of fast ways and slow ways at the granularity of the $k$-ways, where $k$ is determined by the available capacity range of the cache level. Reconfiguration will not affect the bits of the address fields that are used as tag, index, and block offset bits. Modifications to the conventional cache architecture include:

- **Memory status vector** A set of memory status vectors are stored in each level of cache. A single 2-bit entry in the vector represents the current status of the corresponding partition (active, drowsy, or configured as level-i.5).
- **Input and output paths** The input and output data paths are duplicated to accommodate multiple active partitions in a single cache level.
- **Additional multiplexers** The reconfiguration also requires additional wiring and multiplexers at address decoders and tag comparators.

## 3.4 Prediction Engine

Memory reconfiguration relies on accurate predictions of bandwidth demand of a workload to address the dynamically varying application characteristics. Conven-

**ALGORITHM 1:** Statistical prediction algorithm

```
Require: The new demand bandwidth sample $\tilde{w}_i$ in time interval $i$.
Ensure: The prediction of demand bandwidth $w_{i+1}$ in the next time interval $i+1$.
 1: Normalize the new sample to one of quantization bins as $w_i$;
 2: Update the two counters $c(w_{i-2}^{i})$ and $c(w_{i-2}^{i-1})$ with $w_i$;
 3: if !hitPattern($s \leftarrow w_{i-l+1}..w_i$) then
 4:     Add an new entry $s$ into pattern table;
 5:     $p(s) \leftarrow calcProbability(s, c(w_{i-2}^{i}), c(w_{i-2}^{i-1});$
 6: else
 7:     $p(s) \leftarrow calcProbability(s, c(w_{i-2}^{i}), c(w_{i-2}^{i-1});$
 8: end if
 9: $k \leftarrow indexOfMaxProbability(p)$
10: $w_{i+1} \leftarrow patternTable[k][l]$
```

tional last value or table based predictors can model neither long range patterns of an application nor patterns with variable lengths. In this work, we employ a statistical predictor to support the BARCH design. The basic idea of the predictor is similar to the n-Gram models, which are typically used by speech and natural language processing [98]. The n-gram models are usually formulated as a probability distribution $p(s)$ over a sequence of strings $s$, and attempt to reflect how frequently $s$ occurs as a sentence. Prediction on a reasonable combination of strings is then generated based on the probability distribution. Our predictor employs the same basic idea, but with significantly different implementation details. While language modeling is built from a set of previous collected training sentences with finite lengths, our statistical model needs to be able to dynamically generate predictions based on a continuous sequence of metrics. In addition, we can only implement limited resolution for the metrics. Therefore, we need to normalize the demanded bandwidth values with a limited number of quantization bins. In our model, each demanded bandwidth (DBW) sample obtained in a time interval is analogous to a string in language, and a given length (the “order”) of samples is stored in a table as a pattern. At each time interval, the prediction engine will update the pattern table with the new DBW sample, and calculate probability of the updated pattern. Using the chain rule, the probability of a pattern $s$ of length $l$ can be calculated without loss as the product of conditional probabilities as the equation shown below.

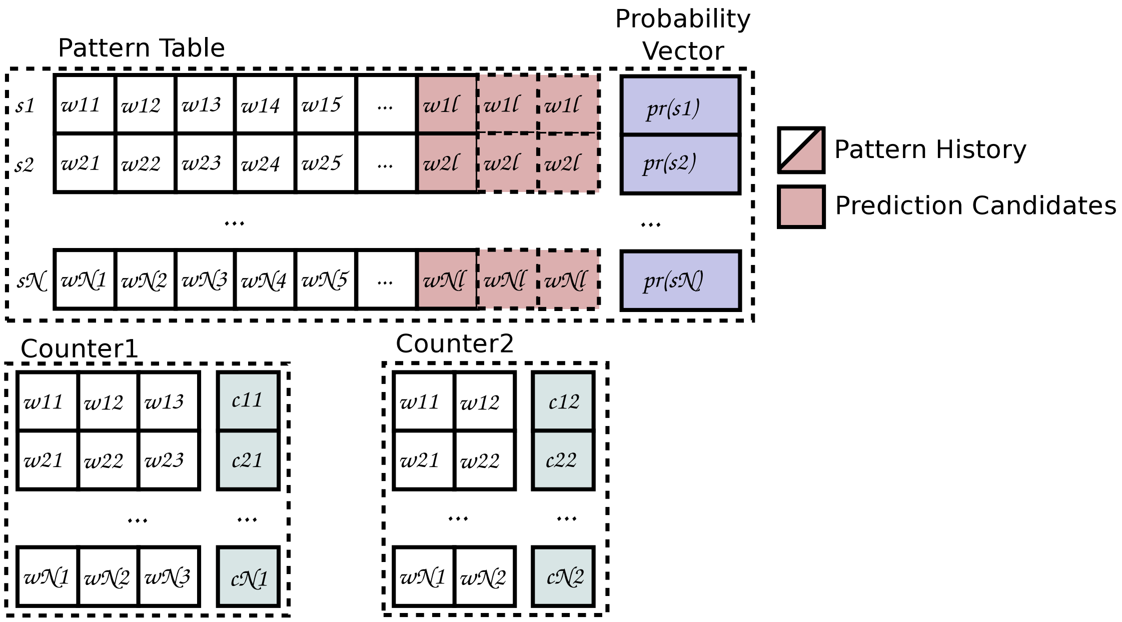


**Figure 3.4.** Components of the prediction engine include the pattern table, the probability vector, and an array of counters.

$$p(s) = p(w_1)p(w_2|w_1)...p(w_l|w_1...w_{l-1}) \tag{3.2}$$

where $w_l$ is the DBW sample obtained in current time interval, and $w_1$ through $w_{l-1}$ are the preceding $l-1$ DBW samples. Equation 3.2 can be represented as

$$p(s) = \prod_{i=1}^{l} p(w_i|w_1...w_{i-1}) \tag{3.3}$$

In n-gram models, we make the approximation that each conditional probability only depends on the preceding $n-1$ samples and obtain the following equation.

$$p(s) = \prod_{i=1}^{l} p(w_i|w_{i-n+1}^{i-1}) \tag{3.4}$$

in which $w_{i-n+1}^{i-1}$ denotes the sequence of $w_{i-n+1}...w_{i-1}$. In order to compute the result of Equation 3.4, an estimation of of $p(w_i|w_{i-n+1}^{i-1})$ can be generated using maximum likelihood estimation (MLE). In n-gram models, the widely-used largest n is n=3, and induces a trigram model. We evaluate the prediction accuracy with different n values, and demonstrate that the trigram model can achieve reason-

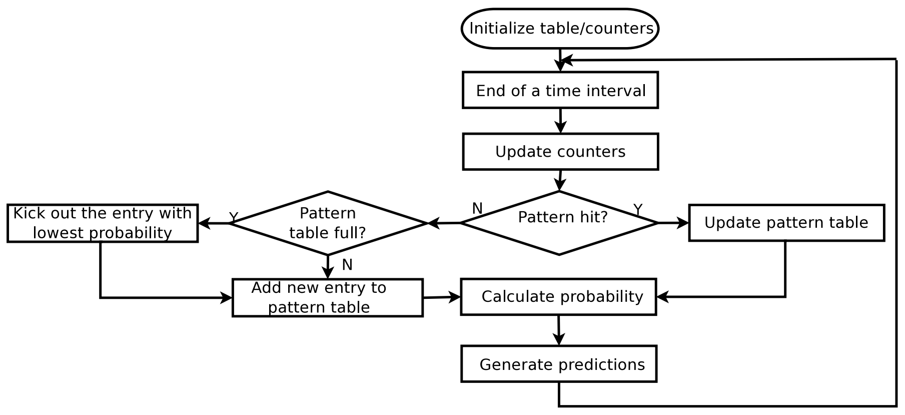


**Figure 3.5.** The control flow of the prediction engine.

**Table 3.1.** Prediction accuracy for different widths of the pattern table

| **Benchmark** | **Width of Pattern Table** | | | | |
|---|---|---|---|---|---|
| | 12 | 10 | 9 | 8 | 7 |
| canneal | 100% | 34% | 34% | 33% | 31% |
| facesim | 98% | 30% | 21% | 19% | 15% |
| streamcluster | 100% | 44% | 42% | 40% | 33% |
| astar | 100% | 100% | 100% | 37% | 31% |
| bwaves | 100% | 100% | 100% | 31% | 35% |
| gamess | 100% | 100% | 100% | 100% | 100% |
| GemsFDTD | 100% | 100% | 100% | 100% | 100% |
| lbm | 100% | 100% | 100% | 56% | 62% |
| mcf | 100% | 100% | 100% | 97% | 49% |
| perlbench | 100% | 100% | 100% | 100% | 100% |
| wrf | 100% | 100% | 100% | 100% | 100% |
| zeusmp | 100% | 100% | 100% | 100% | 100% |

able high accuracy. Therefore, we adopt n=3 in our predictor. Each conditional probability is calculated using MLE in Equation 3.5.

$$\hat{p}(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-2}^{i})}{c(w_{i-2}^{i-1})} \tag{3.5}$$

where $c(w_a^b)$ is the number of times that the sequence $w_a..w_b$ appears in preceding samples. The bandwidth demand of the next time interval is predicted to be the last value in the pattern with the highest probability.

Figure 3.4 shows the hardware components of our prediction engine, which includes a pattern look-up-table, a probability vector, and a set of counters. Figure 3.5 describes the control flow of our predictor. The prediction algorithm is described in Algorithm 1. The predictor can catch a pattern of different lengths.

As shown in Figure 3.4, all the shaded entries in the pattern table can be prediction candidates. If the predictor cannot match a pattern of the maximum available length, it will try to match the patterns with lower orders.

### 3.4.1 Prediction Accuracy

Table 3.1 lists the accuracy of the prediction engine applied to both multithreaded and single-threaded benchmarks. The DBW values are normalized using 20 quantization bins. Based on our evaluation, the primary parameter that affect the prediction accuracy is the width (order) of the pattern table. By storing longer patterns, the predictor is less likely perturbed by a single deviated sample. As illustrated in Table 3.1, our predictor achieves almost 100% accuracy with the order of 12. Multithreaded applications, such as canneal, facesim, and streamcluster, tend to favor higher orders than single-threaded benchmarks. Even with the order of 9, the predictor is still 100% accurate with 9 out of 12 benchmarks.

### 3.4.2 Storage Overhead

Wider pattern tables lead to higher prediction accuracy, but also incur more storage and performance overhead. To balance between the prediction accuracy and the overhead, we configure the width of the pattern table at each cache level to be 12, which guarantees almost 100% prediction accuracy with most of the applications. Each probability entry is a 64-bit floating point value. The counter array is 3-byte wide. The lengths of the pattern table and counter array are fixed to 240. The storage overhead at each cache level is listed in Table 3.2. The prediction engine requires only 6KB of storage at each cache level.

**Table 3.2.** Storage overhead of the prediction engine.

| **Component** | **Width** | **Length** | **Storage** |
|---|---|---|---|
| Pattern Table | 12-byte | 240 | 3KB |
| Probability Vector | 8-byte | 240 | 2KB |
| Counter Vector | 3-byte | 240 | 1KB |

### 3.4.3 Computational overhead

The computational complexity of our prediction algorithm is $O(ql)$, where $q$ is the number of quantization bins. The computational time of generating a prediction is bounded by the size of the pattern table and the limited quantization bins. The overall computational overhead is constrained to be on the order of microseconds. Therefore, the prediction algorithm can be implemented by operating system during context switch without explicit performance overhead.

## 3.5 Experiments

This section shows experimental results for system performance improvement with the our novel BARCH design.

### 3.5.1 Experimental Setup

We use Simics [99] as the simulator to run our experiments. It is configured to model a four-core CMP. Each core is in-order, and is similar to UltraSPARC III architecture. Since our design focuses on shared on-chip memories, we fix the private L1 caches to be 16KB and SRAM-based. Table 3.3 lists the detailed parameters.

We simulate both multithreaded and multiprogrammed workloads. We selected the multithreaded applications with large working sets from PARSEC benchmark suite [100], which consists of emerging workloads designed to represent next-generation shared-memory programs for CMPs. Multithreaded benchmarks from SPEC OMP2001 [101] are also evaluated. The multiprogrammed workloads are selected from SPEC CPU2006 benchmark suite [102]. Since the performance of

**Table 3.3.** Baseline CMP configuration.

| | |
|---|---|
| No. of cores | 4 |
| Configuration | 1GHz, in-order, 14-stage pipeline |
| Private L1 | SRAM, 64B line, size 64KB |
| Shared caches | SRAM/STT-MRAM/eDRAM/ReRAM, 64B line, 1 to 3 levels, size of 512KB to 64MB |
| Main memory | 4GB |

Table 3.4. Characteristics of selected benchmarks. I'06 and F'06 represent the SPEC CPU2006 integer and floating point benchmarks respectively.

| Benchmarks | Benchmark Suite | Multithreaded | Write% | PDBW |
|---|---|---|---|---|
| canneal (CL) | PARSEC | Y | 31.4% | 791 MB/s |
| facesim (FS) | PARSEC | Y | 30% | 572 MB/s |
| streamcluster (SC) | PARSEC | Y | 0.6% | 552 MB/s |
| mgrid (MG) | SPEC OMP2001 | Y | 3.6% | 562 MB/s |
| swim (SW) | SPEC OMP2001 | Y | 3.6% | 643 MB/s |
| wupwise (WW) | SPEC OMP2001 | Y | 4% | 536 MB/s |
| astar | I'06 | N | 38% | 4.1 GB/s |
| bwaves | F'06 | N | 24.5% | 2.5 GB/s |
| gamess | I'06 | N | 28.4% | 1.1 GB/s |
| GemsFDTD | F'06 | N | 30.5% | 2.6 GB/s |
| lbm | F'06 | N | 42.2% | 3.9 GB/s |
| mcf | I'06 | N | 26.2% | 1.8 GB/s |
| wrf | F'06 | N | 25.1% | 2.6 GB/s |
| zeusmp | F'06 | N | 5.5% | 3 GB/s |

different memory technologies are closely related to read and write intensities, we selected some workloads that vary in the L2 cache write intensity (Write%) and peak demand bandwidth (PDBW), which are listed in Table 4.3.

Table 3.5. Multithreaded and multiprogrammed workload sets.

| | Abbreviation | Workload Sets |
|---|---|---|
| Multithreaded | CL | canneal |
| | FS | facesim |
| | SC | streamcluster |
| | MG | mgrid |
| | SW | swim |
| | WW | wupwise |
| Multiprogrammed | M1 | sphinx3+astar+lbm+zeusmp |
| | M2 | wrf+GemsFDTD+bwaves+mcf |
| | M3 | perlbench+milc+gamess+sphinx3 |
| | M4 | sphinx3+wrf+perlbench+astar |
| | M5 | gamess+milc+perlbench+mcf |
| | M6 | mcf+milc+lbm+gamess |
| | M7 | perlbench+lbm+astar+milc |
| | M8 | zeusmp+bwaves+wrf+mcf |

We evaluate the shared cache hierarchy in four different cases: pure SRAM-based L2 cache with fixed capacity (SRAM.fix), hybrid L2/L3/L4 caches with fixed maximum available capacity at each level (hybrid.fix), hybrid reconfigurable caches (hybrid.rfg), and hybrid reconfigurable caches with workload partition (hybrid.par). SRAM.fix is the baseline. The case of hybrid.par is only applied to mul-

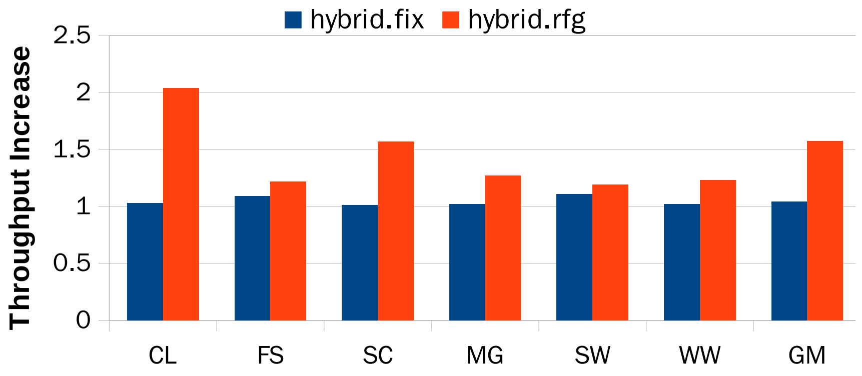


**Figure 3.6.** Performance improvement of multithreaded workloads, evaluated in terms of throughput, i.e., the number of executed instructions per second.

tiprogrammed workloads. With hybrid.par, we partition the shared cache space according to the specific demanding bandwidth of each individual application in a workload set. The metric we evaluate is the throughput, which is the executed instructions per second. With PARSEC multithreaded workloads, we evaluate the result obtained within the region of interest (ROI) defined in each benchmark source code. With SPEC OMP2001 and SPEC CPU2006 applications, we warm up the caches with 500 million instructions and then evaluate the next 1 billion cycles.

### 3.5.2 Results

This section shows our experimental results and explain the reasons leading to these results.

**Multithreaded workloads:** Figure 3.6 shows the results of throughput improvement with multithreaded benchmarks, where throughput is the number of executed instructions per second. Throughput of each configuration is compared to the case when only SRAM-based L2 cache is present in CMP system. As illustrated in Figure 3.6, hybrid.fix does not help much to improve the performance of most multithreaded workloads. With large capacity at each level, the provided bandwidth of the cache hierarchy is also fixed in a low level. Many multithreaded workloads do not require large caches. Smaller cache sizes are sufficient to accommodate their working sets. With hybrid.rfg, we tailor the cache capacities

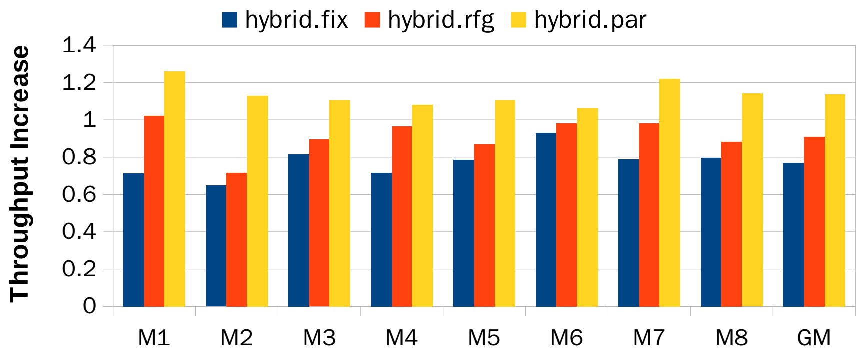


**Figure 3.7.** Performance improvement with multiprogrammed workloads, evaluated in terms of throughput, i.e., the number of executed instructions per second.

according to the demand of each workload. The results show that hybrid.rfg improves the throughput of all the evaluated benchmarks. The geometric mean of the performance improvement achieves 58%.

**Multiprogrammed workloads:** With multiprogrammed workloads, each processor core executes one benchmark workload. Figure 3.7 illustrates the performance improvement of different cache hierarchy configurations. The configurations of hybrid.fix and hybrid.rfg do not improve the throughput. In fact, both configurations result in performance degradation with most of workload sets. One possible reason leading to the performance degradation of hybrid.fix is that the multiprogrammed workloads have high bandwidth demand. The provided bandwidth of hybrid.fix is maintained in a relatively low level. With hybrid.rfg, a factor to affect the performance is the reconfiguration time. The reconfiguration controller consumes additional cycles at each time interval. Another reason that leads to the performance degradation of both hybrid.fix and hybrid.rfg is contention between the working sets of different workloads. The only configuration that improves throughput is hybrid.par. The primary benefit of hybrid.par is to minimize contention. At each time interval, each workload is partitioned to a separate cache space according to individual requirement. If all the workloads in a multiprogrammed workload have stable working sets, the partition is also stable. Different workloads are less likely to compete with each other for cache space. Another benefit is that the cache hierarchy is reconfigured to fit each individual workload rather than the whole workload set. Different from multithreaded workloads, which have relatively balanced requirement with each thread, different workloads in a multiprogrammed workload set have different bandwidth demand. It is unfair to tune the cache hierarchy according to the overall bandwidth demand. Rather than global tuning the cache hierarchy, reconfiguring each partition with individual workloads is much more flexible. Overall, the geometric mean of throughput improvement with hybrid.par is 14%. Based on the experimental results, hybrid.par can be selected as reconfiguration scheme with multiprogrammed workloads.

## 3.6 Summary

This chapter proposed a bandwidth-aware reconfigurable cache hierarchy design with hybrid memory technologies, including traditional SRAM, eDRAM, and emerging NVRAMs. The design consists of a hybrid cache hierarchy, reconfiguration mechanisms, and a prediction engine. The hybrid cache hierarchy leverages different memory technologies to provide an optimized bandwidth-capacity curve to the on-chip portion of the memory hierarchy, which effectively reduces the bandwidth demand to the off-chip main memory. On top of the hardware substrate of such a hybrid cache hierarchy, we dynamically reconfigure the cache capacity at each level adaptive to the bandwidth demands of different applications. We also present an accurate statistical prediction engine to facilitate such reconfiguration. We evaluate the proposed design method with both multithreaded and multiprogrammed workloads. Experimental results show that reconfigurable hybrid cache leads to 58% and 14% performance improvements to multithreaded and multiprogrammed workloads, respectively.

Chapter 4

# Energy-Efficient Graphics Memory Design

The increasing computational power of modern GPUs makes it a commonly used solution for high-performance computing by employing hundreds of processing units and thousands of in-flight threads [11, 12]. Although the bandwidth of graphics memories continues to increase in recent years, the energy efficiency of graphics memories is decreasing due to an increasingly significant portion of power in GPU systems.

Various previous work explored how to address the GPU power challenge [103, 104, 105, 106, 107, 108]. Gebhart *et al.* investigated register file caching and multi-level thread scheduling to reduce the number of accesses to large register files to reduce power [103]. SRAM-DRAM hybrid memory technology was exploited by Yu *et al.* to reduce the area and power consumption of GPU register files [104]. In their work, embedded DRAM (eDRAM) with a higher density than SRAM is used to store multiple copies of register file data. Wang *et al.* proposed the predictive shader shutdown technique to exploit workload variations across frames for leakage reduction of GPU shader processors [108]. Software optimization was studied by Ren *et al.* to improve the GPU power efficiency by modifying matrix multiplication algorithms [105]. Most of the existing studies explore either GPU shader cores and caches architecture, or software optimization, and require both hardware and software modifications to current GPU processor design. In our work, we explore power reduction techniques by limiting the architectural modifications to the graphics memory interface with only minor changes to GPU compute-unit architecture.

Most existing mechanisms seeking to save power consumption of GPU systems focused on leveraging idle states of GPU cores [109, 110] and graphics memory [111]. How to actively tune the VF states of GPU systems remains an open question. A large body of previous work has studied CPU system power management with DVFS techniques. Most of these studies focused on DVFS of only CPU cores [112, 113, 114, 115]. Recent work [116, 117] showed that DVFS on memory provides substantial energy savings. Very few work studied coordinated DVFS for power management of the entire system including CPU processor and memory subsystem. CoScale [118] explored power management mechanisms by coordinating DVFS of CPU and memory subsystem under performance constraints. However, CoScale [118] implemented its DVFS algorithm in operating system (OS) with a typical reconfiguration interval corresponding to an OS time quantum (5 milliseconds). Directly adopting the CoScale [118] method in GPU systems will require even longer reconfiguration intervals, because the DVFS algorithm that executes on the host CPU will need to communicate with GPU through PCIe interface with a very long turn-around latency.

The emergence of various NVRAM technologies provides promising memory system solutions with non-volatility and low power consumption. Our study of various GPU workloads shows that only a portion of data (less than 50% for 10 out of 20 studied application) is frequently accessed during run-time. Therefore, the data that is infrequently accessed and of low write intensity can be stored in NVRAM and managed in standby mode with near-zero power consumption. In this chapter, we propose a hybrid graphics memory design, mixing DRAM, STT-MRAM, and ReRAM. It can provide higher memory bandwidth and consumes less power than the traditional GDDR5 memory. Replacing part of the DRAM with a NVRAM partition, the hybrid graphics memory can run at a higher frequency and thus provide higher peak memory bandwidth. By migrating the read-only and infrequently-accessed data in the NVRAM partition, the hybrid memory system also consumes less power than conventional GDDR5 memory. Although NVRAM has a longer write latency than DRAM, our study indicates that the memory access patterns of GPU workloads can naturally hide such latency. In order to save the

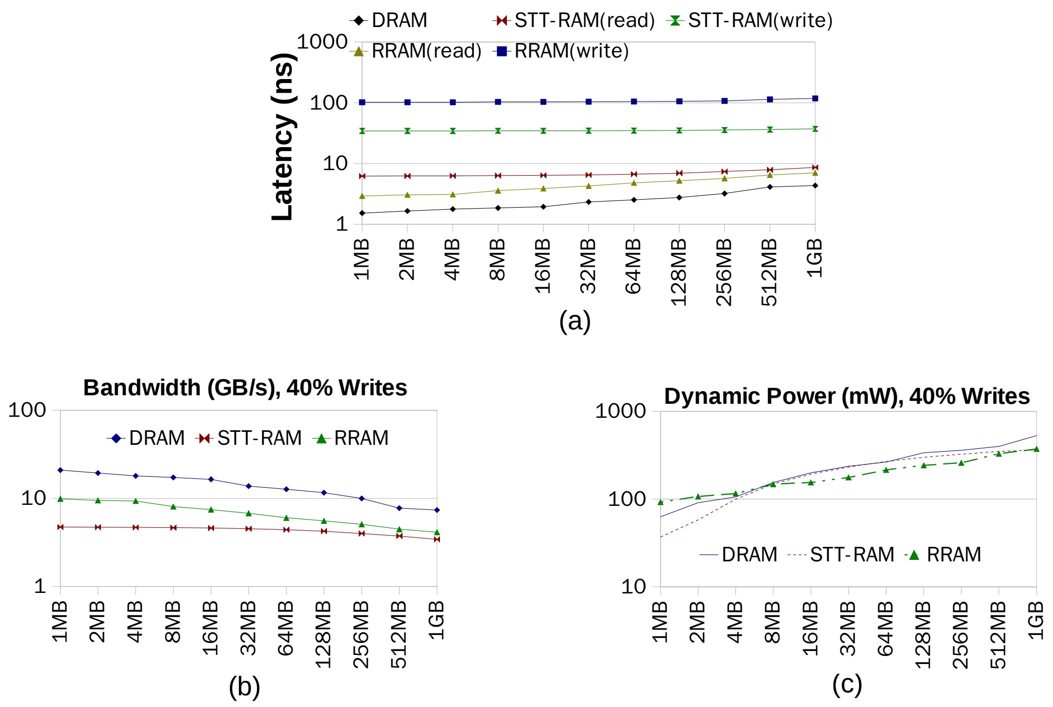


**Figure 4.1.** (a) Latency, (b) provided bandwidth (PBW), and (c) dynamic power of different memory technologies with respect to different capacities.

memory power without much performance degradation, we propose an adaptive data migration mechanism, leveraging different memory access patterns of different GPGPU workloads.

## 4.1 Motivation

We evaluate the provided bandwidth (PBW) and power consumption of DRAM, STT-MRAM, and ReRAM. NVRAMs appear to have significant power benefits, but lower PBW than DRAM. Our hybrid memory hardware configuration is built based on these bandwidth and power characteristics. We also study the memory access patterns of various GPU applications, based on which we develop our energy-efficient adaptive data migration mechanism.

### 4.1.1 Characteristics of Various Memory Technologies

We use NVSim [95], a circuit level memory model, to evaluate the performance, bandwidth, and power of the three memory technologies of DRAM, STT-MRAM, and ReRAM. Figure 4.1 and 4.2 illustrate the results. At each memory capacity, the memory bank and mat organizations are optimized for read latency to minimize the dynamic power consumption. We only show bandwidth and dynamic power at 40% write intensity, the maximum write intensity observed from the GPU applications we studied.

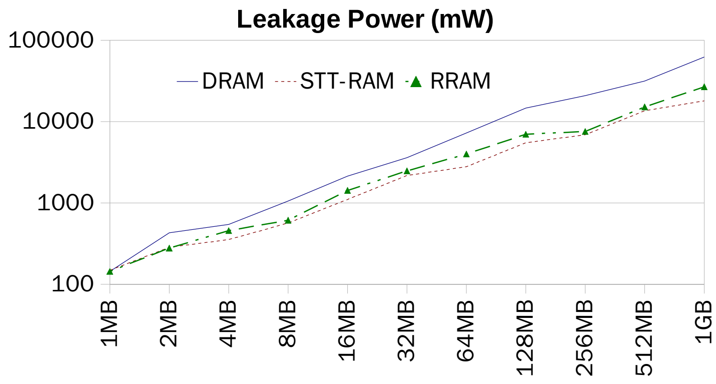


**Figure 4.2.** Leakage power of different memory technologies with respect to various capacities.

**Latency and bandwidth:** Figure 4.1(a) shows that DRAM has the lowest read and write latencies among the three memory technologies. The read latency of ReRAM is lower than STT-MRAM, and is comparable with DRAM at large capacities. For example, ReRAM only incurs 2ns additional read latency than DRAM at 128MB. Although with higher read latency, STT-MRAM has lower write latency than ReRAM across all capacities. Figure 4.1(b) shows the PBW curves. Due to long write latency, the average PBW of ReRAM and STT-MRAM is only 50% and 30% that of DRAM.

**Power:** Due to the low access current, the dynamic power of ReRAM is lower than DRAM. As shown in Figure 4.1(c), ReRAM consumes 17% dynamic power less than that of DRAM at the capacities larger than 128MB. Figure 4.2 illustrates the leakage power of different memory technologies. On average, the leakage power of STT-MRAM and ReRAM is only 37% and 48% that of DRAM. In addition,

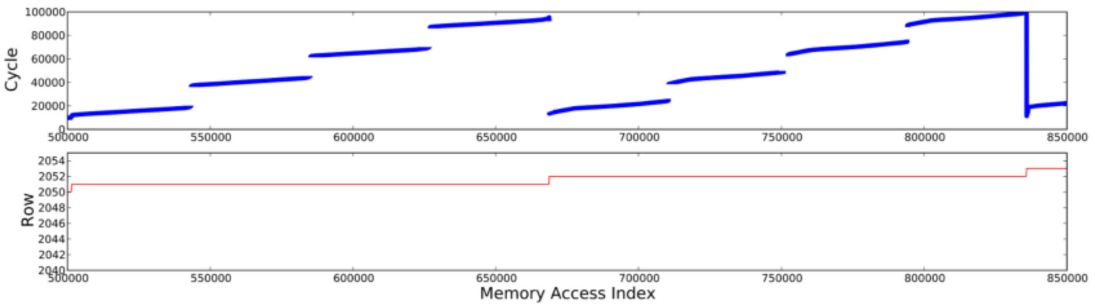


**Figure 4.3.** The pattern of “interleaved access”.

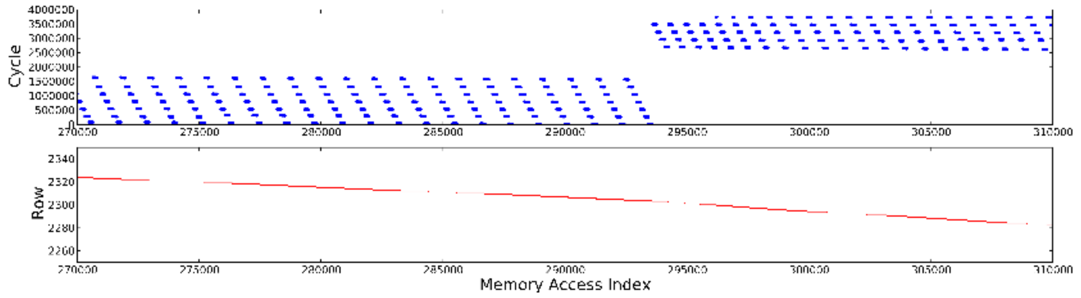


**Figure 4.4.** The pattern of “access then idle”. Can also observe the “burst” pattern during the access period.

due to the non-volatile nature, we can obtain near-zero standby power by power gating the idle portions STT-MRAM and ReRAM. Overall, we can not find a single winner from the perspective of both performance and power. DRAM has the best PBW among the three memory technologies. However, the two NVRAMs have significant power benefits.

Based on such observation, we adopt hybrid memory design to combine the benefits of high PBW of DRAM and low power consumption of NVRAMs.

### 4.1.2 Memory Access Patterns of GPGPU Workloads

We examined various GPGPU workloads on the baseline GPU system with GDDR5 graphics memory (Section 6.7), and observe three memory access patterns, namely “interleaved access”, “access then idle”, and “burst”.

**Interleaved access:** Figure 4.3 demonstrates the pattern of “interleaved access”. Here we sort the memory accesses based on ascendant order of DRAM row address. The x-axis is the index of memory accesses. The figure on the first row represents the cycle of each memory access. The figure on the second row represents the row that is accessed by each memory request. For example, the 550000th memory access is from the *row 2051* at *cycle 40000*. Figure 4.3 shows that row 2051 is accessed during the entire application execution. However, we can observe three idle periods that are twice as long as the time when the row being accessed. Although not shown in the figure, we also observe that a significant $\frac{5}{6}$ of memory access is read-only. This portion of data may be corresponding to constant or texture data in GPU applications. Since ReRAM only incur small read latency penalty, we can maintain this portion of data into ReRAM. We can reduce the memory power by powering off the memory space during the idle periods.

**Access then idle:** Figure 4.4 illustrates the pattern of "access then idle". We can observe that the memory rows between 2300 and 2320 are only accessed during the initial time to the cycle 1,800,000. Afterward, this portion of memory becomes idle, and never accessed again. Potentially, we can turn off this portion of memory to save power. This is impossible with pure DRAM based memory. With NVRAMs, however, it is feasible to standby some portions of memory space.

**Burst:** The "burst" pattern represents frequently accesses during the entire execution time. It can be observed during the access period of both "interleaved access" and "access then idle" patterns. For example, in Figure 4.4 row 2300 is in "burst" state before cycle 2,400,000 is achieved. This portion of data needs to be maintained in DRAM to minimize performance degradation.

## 4.2 Hybrid Graphics Memory Architecture

Figure 4.5 depicts an overview of our hybrid graphics memory design. We replace half of the DRAM capacity with ReRAM and STT-MRAM. With half the capacity, DRAM can provide up to 25% higher memory bandwidth by scaling up the clock frequency. We migrate the read-only and infrequently accessed data to NVRAMs. Due to the non-volatility, we can significantly reduce the memory power consumption by powering off the idle NVRAM space. The hardware modification is limited to the memory interface and controllers and no modification is required to the internal structures of the GPU processors and the memory arrays.

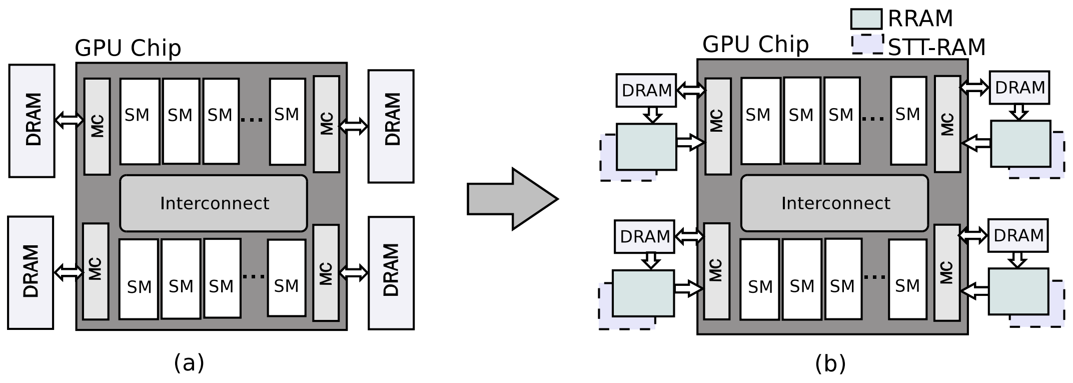


**Figure 4.5.** Overview of GPU system with hybrid memory. (a) Conventional GPU system with off-chip GDDRs. (b) GPU system with hybrid memory.

## 4.2.1 Hardware Configuration

**Partitions of different memory technologies:** Our hybrid graphics memory consists of DRAM, ReRAM, and STT-MRAM partitions. Frequently accessed data is maintained in the DRAM that has the highest PBW among the three memory technologies. The two NVRAM partitions are used to store the data with low access frequency. The idle rows in the NVRAM partitions are powered off. Between the two NVRAMs, ReRAM is used to store data that is read-only or with extremely low write intensity. Since STT-MRAM has better write performance than ReRAM, it is used to store the infrequently accessed data with higher write intensity.

**Capacity of each partition:** Our baseline GPU system (Section 6.7) employs 256MB DRAM per channel. In our hybrid graphics memory, we reduce the DRAM capacity to 128MB per channel and replace the rest 128MB with ReRAM. This is based on the observation that the dynamic power of DRAM is higher than ReRAM at the capacities of higher than 128MB. In addition to power benefits, reducing the DRAM capacity can also improve PBW. In Figure 4.1(b), we observe that the PBW of DRAM at 128MB is 1.25× that of 256MB. We adopt 8MB STT-MRAM per channel. This is based on the fact that the dynamic power of STT-MRAM is the lowest among the three memory technologies at small capacities. The row buffer size of different memory partitions are configured to be 256-bit, so that data migration incurs minimum overhead in address mapping. The hybrid memory may incur a small increased area ($9mm^2$) compared to the baseline (45nm technology).

However, the increased memory area is only limited to the off-chip memory and we do not expect high cost increase to the GPU system.

**Memory interface:** Figure 4.6 shows our memory interface configuration. As shown in Figure 4.6, an additional bus of 32 bits is adopted to accommodate data migration and NVRAM reads (data will be read out directly from the NVRAMs, once it is migrated to the NVRAM partitions). GPU can read and write from the DRAM partition. GPU can only read from the NVRAM partition. In our experiments, we consider the additional I/O termination power incurred by this modification and show the total system power is still reduced with such I/O overhead. Furthermore, multiplexing is required to switch the reads between DRAM and NVRAMs.

**Memory controller:** A NVRAM controller is integrated in the memory controller to accommodate the data migration and manage the power mode of NVRAMs. Components of the NVRAM controller include data migration buffers, read buffers, and registers to store the memory idle states. We modify the memory controller to facilitate data migration. Timers are employed to manage the idle state of each DRAM row. Each row also has counters to collect memory access data. We evaluated the area and power overhead of these timers and counters. For 32-bit timers (counters), the total storage size will be 32.3KB (assume 2KB page size (row-buffer size) as same as the baseline GDDR5). This is negligible in a memory system of 256MB. An address mapping table is used to store the new address after a row is migrated from DRAM to NVRAMs.

### 4.2.2 Data Migration Mechanism

Our data migration algorithm is implemented in the memory controllers. The goal of our data migration mechanism is to improve the system energy efficiency, i.e., to reduce the memory power consumption with low performance degradation. Figure 4.6 illustrates the data flow between memory controller and hybrid memory. The overall idea is to store the idle data in NVRAMs in powered-off state, and maintain the frequently accessed data in DRAM. Based on our GPU workload characterization, we find the opportunity of data migration in both “access and idle” and “interleaved access” patterns. With “access then idle”, it is straight-

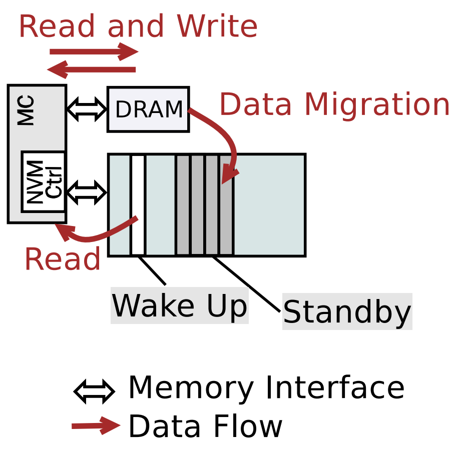


**Figure 4.6.** Memory interface configuration and data flow between memory controller and hybrid memory.

forward to start migration once a row becomes idle. With "interleaved access", however, the start point of data migration needs to be carefully determined. To obtain sufficient energy benefit from data migration, the idle time need to be sufficiently long. Otherwise, the GPU system may suffer significant performance loss due to the low PBW of NVRAMs and NVRAM lifetime will be reduced. Our data migration mechanism is illustrated in Figure 4.7 and 4.8.

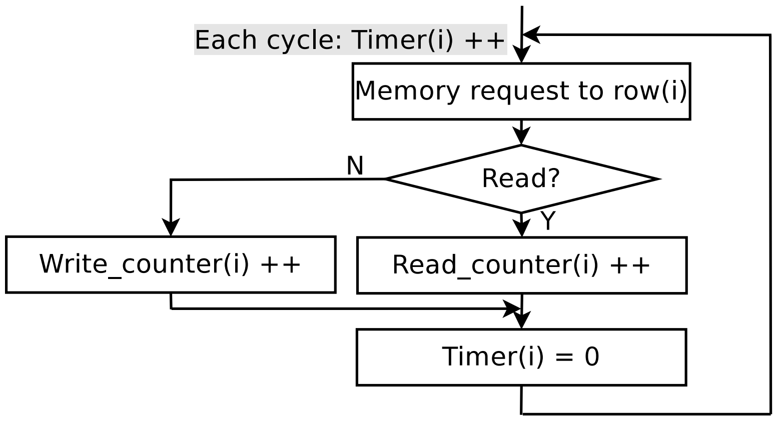


**Figure 4.7.** The loop of DRAM access management.

Figure 4.7 is the loop of DRAM access management. At each DRAM access, we update the access counters and the row timer, based on the types of accesses. Figure 4.8 shows the mechanism to determine the start point of data migration. We do not immediately initialize a data migration operation at a row time-out

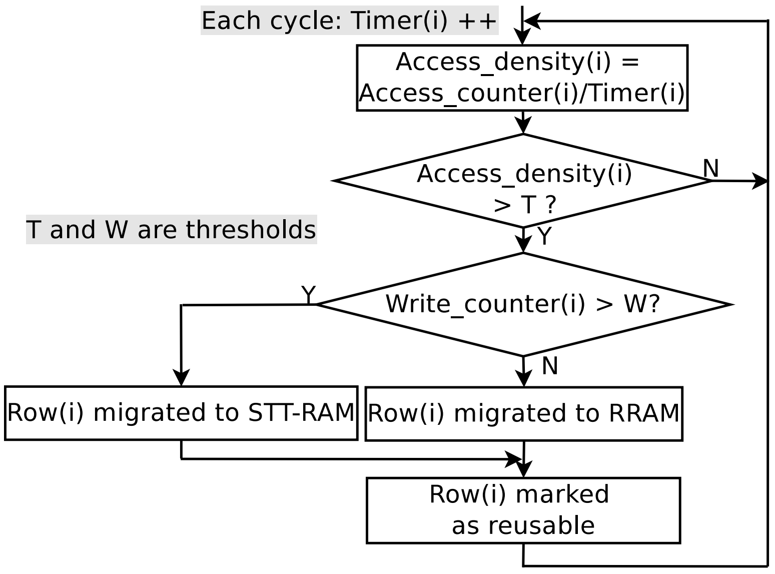


**Figure 4.8.** Control of data migration.

event. Instead, data migration is determined by the memory access density within a period of execution time. In this way, we avoid the GPU system performance degradation incurred by unnecessary data migration and extend the NVRAM lifetime. Furthermore, we evaluate the write intensity to determine which NVRAM space will be used. STT-MRAM is used when the write intensity is higher than a pre-defined threshold to avoid high performance degradation incurred by frequent data migration. Some applications may have extremely low memory intensity. The power consumed by data migration may be higher than the DRAM dynamic power. In this case, we disable data migration when the memory intensity is lower than a threshold. To determine the power modes (standby or wake-up), we maintain a timer for each row in NVRAMs. Once a time-out event is detected, we power off the row. A read or migration request to the row will wake it up.

## 4.3 Experimental Setup

This section describes our simulation framework, the evaluated workloads, and our GPU system power model.

**Table 4.1.** Baseline GPU configuration. The parameters of streaming multiprocessors (SMs), caches, and memory controllers.

| | | | |
|---|---|---|---|
| Number of SMs | 16 (574MHz Clock) | Warp Size | 32 |
| CTAs per SM | 8 | Threads per SM | 1536 |
| Shared Memory per SM | 48KB | Registers per SM | 32768 |
| L1 Caches per SM | Texture: 12KB; Data: 16KB | DRAM Request Queue Size | 16 16 |
| Shared L2 Cache | 768KB | Memory Controllers | 6 (FR-FCFS) |

**Table 4.2.** DRAM configurations. Baseline is off-chip GDDR5 memory with 32-bit bus width per chip. The maximum bus width of 3D die-stacked DRAM is 256-bit per chip. The maximum clock frequency is 1.5GHz. The peak memory bandwidth of 3D die-stacked DRAM can be changed by scaling down the bus width and clock frequency.

| Memory Interface | Baseline GDDR5 | 3D die-stacked DRAM |
|---|---|---|
| Clock Frequency | 1.5GHz | 375MHz - 1.5GHz |
| Bus Width per Chip | 32 bits | Maximum 256 bits |
| Bandwidth | 144GB/s | Maximum 720GB/s |
| Memory Timing | $t_{RAS}$=28ns, $t_{RP}$=12ns, $t_{RC}$=40ns, $t_{RCD}$=12ns, $t_{RRD}$=5.5ns | $t_{RAS}$=22.4ns, $t_{RP}$=9.6ns, $t_{RC}$=32ns, $t_{RCD}$=9.6ns, $t_{RRD}$=4.4ns |

### 4.3.1 Simulation Framework

We use GPGPU-sim v.3.0 [119], a cycle accurate PTX-ISA simulator, to run our experiments. The simulator models shader cores, texture caches, constant caches, L1 caches, interconnection network, memory controllers, and DRAM memories. Table 4.1 and Table 4.2 specify the configurations and parameters used in our simulation. We assume the system is implemented with 40nm process technology. We evaluate a GPU processor with 16 streaming multiprocessors (SMs). The SMs, caches, and memories are configured based on NVIDIA Quadro®6000 [12]. We model a perfect crossbar interconnection network in the GPU processor with one-cycle latency, so that the bandwidth demand at the processor-memory bus is not limited by the network bandwidth. We modify the simulator to implement our reconfiguration mechanisms.

Table 4.2 lists the DRAM configuration parameters used in our simulation. Each memory controller has two DRAM channels. Therefore, we model a system with in total 12 DRAM channels. The baseline is off-chip GDDR5 graphics memory with 32-bit bus width per chip and 1.5GHz clock frequency. This is in-line with the GDDR5 memory used by NVIDIA Quadro®6000 [12]. The low level memory timing of the baseline is obtained from datasheet [120]. 3D die-stacked memory latency is the sum of DRAM core access latency, silicon interposer pin delay, intra-package wiring delay, and memory controller traversal delay. Our 3D die-stacked memory employs 3D stacked DRAM dies. As reported in Tezzaron's datasheets [53], the access latency (tRC) of a five-layer DRAM is only 67.5% of that of conventional 2D GDDR5 memories. Furthermore, the latency of signals passing through silicon interposer can be reduced to 1/5 of that with standard I/Os [41]. Therefore with 3D die-stacked DRAMs, we conservatively assume 20% memory latency reduction compared to off-chip GDDR5 memory. The refresh period of off-chip DRAM is 64ms. To account for higher leakage rates due to higher temperature operation, we assume a 32ms refresh period with 3D die-stacked DRAM. The maximum bus width of each 3D die-stacked DRAM chip is 256 bits. We evaluated the system energy efficiency with various peak memory bandwidths by varying the configuration of memory interface (bus width, clock frequency, and supply voltage). With the maximum clock frequency of 1.5GHz, 3D die-stacked graphics memory can provide 1152GB/s peak memory bandwidth. In this dissertation, we only evaluated up to 720GB/s peak memory bandwidth. This is sufficient to show the benefit of our design.

### 4.3.2 Workloads

We evaluated various available GPU workloads from the NVIDIA CUDA SDK [121] and Rodinia Benchmarks [122]. Table 4.3 lists the characteristics of our 26 workloads. The memory intensity of some applications, such as *MC*, *MS*, and *BN*, is lower than 1.0. The three most memory-intensive benchmarks are *KM*, *NW*, and *BFS*. We profiled our benchmarks by sweeping the instruction interval $N$ from one thousand to ten million. We found that we can catch the changes of memory intensity of all the benchmarks with one million instruction intervals. Further reducing the instruction interval can incur performance overhead by frequently invoking the reconfiguration algorithms without really reconfiguring the memory interface. Therefore, we show the results with one million instruction interval in Section 4.4.

**Table 4.3.** Characteristics of selected GPGPU benchmarks. (IC represents instruction count. MI represents memory intensity.)

| Abbrev. | Benchmarks | IC | MI | IPC | Power |
|---|---|---|---|---|---|
| MC | Monte Carlo [121] | 1G | 0.1 | 526 | 158W |
| MS | Merge Sort [121] | 2G | 0.5 | 522 | 191W |
| BN | Binomial Options [121] | 11.8G | 0.6 | 549 | 196W |
| CT | Texture Convolution [121] | 5.4G | 1.0 | 565 | 217W |
| MM | Matrix Multiplication [121] | 836M | 1.4 | 397 | 169W |
| SN | Sorting Networks [121] | 5.6G | 1.5 | 484 | 158W |
| HS | Hot Spot [122] | 297M | 1.7 | 375 | 127W |
| NE | Nearest Neighbor [122] | 47M | 1.8 | 109 | 148W |
| PF | Path Finder [122] | 72M | 2.3 | 196 | 183W |
| SLA | Scan of Large Arrays [121] | 15.4G | 3.3 | 437 | 174W |
| DWT | Discrete Wavelet Transform [122] | 20M | 3.4 | 411 | 152W |
| GS | Gaussian Elimination [122] | 9M | 3.6 | 143 | 149W |
| HG | Histogram [121] | 5G | 3.7 | 491 | 217W |
| 64H | 64bin Histogram [121] | 25.8G | 3.9 | 423 | 204W |
| CS | Separable Convolution [121] | 8G | 4.1 | 428 | 212W |
| LUD | LU Decomposition [122] | 5.6G | 4.8 | 150 | 161W |
| SD | Speckle Reducing Anisotropic Diffusion [122] | 2.4G | 5.7 | 379 | 186W |
| BP | Back Propagation [122] | 190M | 6.2 | 388 | 213W |
| CFD | CFD Solver [122] | 4.9G | 8.5 | 187 | 197W |
| BLK | BlackScholes Option Pricing [122] | 9.5G | 9.0 | 203 | 154W |
| FWT | Fast Walsh Transform [121] | 4.4G | 12.2 | 209 | 129W |
| SC | Streamcluster [122] | 2.8G | 15.2 | 191 | 199W |
| SP | Scalar Product [121] | 24M | 19.0 | 259 | 163W |
| KM | K-means [122] | 6.1G | 23.9 | 260 | 117W |
| NW | Needleman Wunsch [122] | 211M | 27.6 | 29 | 173W |
| BFS | Breadth First Search [122] | 454M | 81.9 | 62 | 140W |

### 4.3.3 Power Model

We model system power consumption of three sub-components, including GPU cores and caches, memory controllers, and DRAMs. We calculate the power of GPU cores, caches and memory controllers based on the power model from McPAT [123]. We modify the power model to adapt to the configuration of GPU processor. We add GPU-specific power components to the power model, including warp schedulers and instruction buffers, the large register file, different types of caches, and the shared memory. The dynamic instruction execution and memory access information is fed into the power model to calculate the run-time power consumption of each component. We calculate the DRAM power based on the power model from Micron [15]. First of all, we calculate the maximum DRAM power with different interface configurations offline, assuming 100% memory bandwidth utility. The values are stored in the central controller. During the execution of an application, we obtain its run-time memory access statistics, and calculate the real-time power consumption. With 3D die-stacked DRAM, on-die termination resistors can be eliminated [124]. Therefore, we only model the power consumption of I/O drivers.

## 4.4 Results

This section shows evaluation results on system performance, power, and energy efficiency with the our energy-efficient graphics memory design and explain the reasons leading to these results.

Figure 4.9 to 4.12 show the power, performance, and energy efficiency with our hybrid graphics memory design.

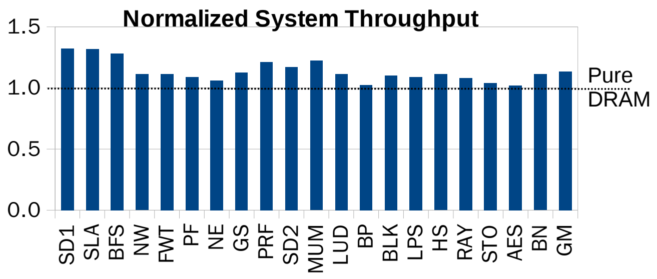


**Figure 4.9.** System throughput with hybrid graphics memory, normalized to baseline.

### 4.4.1 Throughput Improvement

We have shown that with half the capacity of the baseline DRAM, we can obtain up to 25% PBW improvement by scaling up the DRAM clock frequency (Section 6.2). We evaluate the system throughput, which is the number of executed instructions per second, with our hybrid GPU memory architecture. As shown in Figure 4.9, the most significant throughput improvement is obtained by applications with high DBWs, such as *SD1*, *SLA*, and *BFS*. Although the average DBW of *PRF* and *MUM* is lower than 10 GB/s, the two applications suffer very heavy memory accesses with high DBW during a period of execution time. Therefore, increasing the PBW can also improve the throughput with these two applications. The mean throughput improvement with all the applications is 12%.

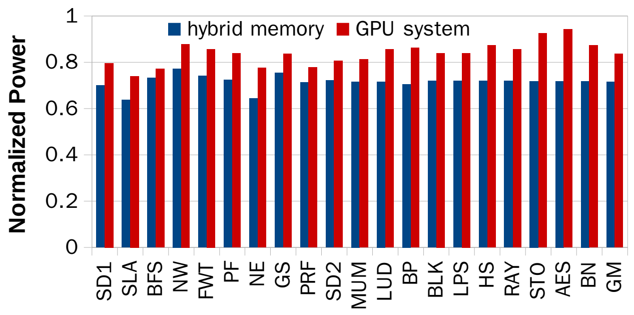


**Figure 4.10.** Memory and system power consumption with hybrid graphics memory, normalized to the baseline pure DRAM based graphics memory.

### 4.4.2 Power Savings

Increasing the DRAM clock incurs power overhead, since the supply voltage is scaled up as well. However, the reduced DRAM capacity results in significant leakage power reduction. The NVRAM partitions also reduces the memory power consumption with low dynamic power and near-zero standby power. We observe that both memory and system power consumption is reduced with all evaluated applications. Figure 4.10 shows the power consumption of hybrid graphics memory and GPU system, normalized to the baseline with pure DRAM based graphics memory. Our proposed design is more effective on savings system power with applications that have higher bandwidth demands, such as *SD1*, *SLA*, and *BFS*, because memory consumes a large portion of total system power for these applications. The mean savings of memory and system power consumptions are 31% and 16%, respectively.

### 4.4.3 Power Breakdown

We also studied the power breakdown of our hybrid graphics memory design. The results are demonstrated in Figure 4.11. Here we do not show DRAM refresh power, since the maximum of two refreshes are observed with various applications and incur negligible (less than 1% of total memory power) power consumption.

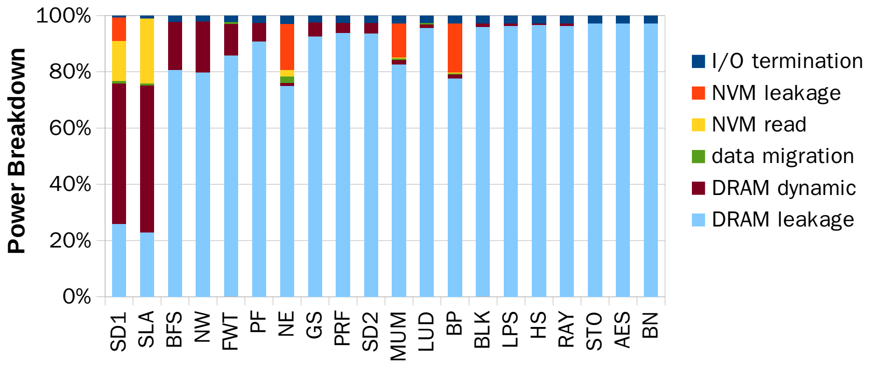


**Figure 4.11.** Power breakdown of hybrid graphics memory.

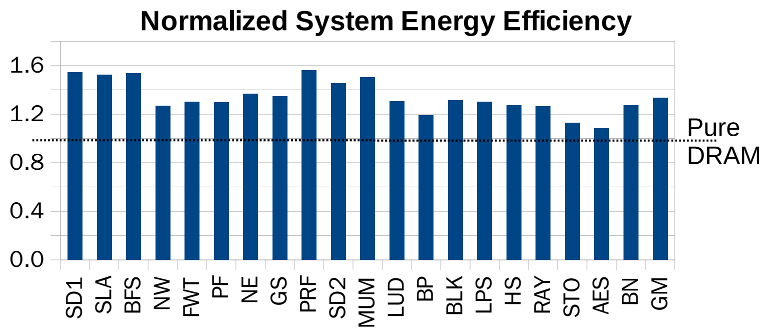


**Figure 4.12.** System energy efficiency with hybrid graphics memory, normalized to baseline.

With applications that have high DBWs, such as *SD1* and *SLA*, the dynamic power covers a significant portion of total memory power consumption due to the high memory access intensity of these workloads. In contrast, DRAM leakage power dominates the total memory power consumption of applications with low DBWs, such as *STO*, *AES*, and *BN*. In this case, we cannot afford the dynamic power consumption of data migration and it is therefore disabled with these three applications to avoid the power overhead incurred by data migration.

## 4.4.4 Energy Efficiency

Figure 4.12 illustrates the results of system energy efficiency, defined as the executed instructions per second per Watt. The system energy efficiency is improved with all the evaluated applications. The most significant improvement can be observed with the applications having high DBWs. The mean improvement of system energy efficiency is 33%.

## 4.5 Summary

This chapter described a hybrid graphics memory design, which improves both memory bandwidth and GPU system energy efficiency. The key insight in our work is that hybrid graphics memory design is especially suitable for GPU applications. The memory access patterns of these applications are naturally used to hide the latency issue of NVRAMs. Our initial results are very promising for future GPU systems, improving 33% in system energy efficiency. Our migration mechanism limits the frequency of write operations, and therefore we do not expect significantly degradation of the lifetime.

Chapter 5

# Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support

Traditional computing systems have adopted a two-level storage model with separated volatile memory and nonvolatile storage systems. Recently, this traditional storage model is enriched by the new persistent memory technology, which blurs the boundary between the two levels by incorporating the properties of both main memory and storage. An application can directly access persistent data through a memory interface with loads and stores, without paging data blocks from/to a storage device or context switching while servicing page faults.

Applications that require high reliability, such as databases and file systems, need to periodically store critical data in nonvolatile devices so the data can survive system failures or program crashes. Commodity computing systems employ slow block-addressable storage media, such as spinning disks or flash, to store this critical data. Due to hardware (PCIe or SATA I/O delay) and software (legacy block-oriented file system interfaces) costs, applications suffer from significant throughput degradation.

**Persistent memory** is a new technology incorporating the properties of both main memory and storage. An application can directly access persistent data through a memory interface with loads and stores, without paging data blocks from/to a storage device or context switching while servicing page faults. Recent work [69, 70] has demonstrated much higher program throughput (up to 32×) by utilizing byte-addressable nonvolatile memory technologies (NVRAM) such as spin-transfer torque RAM (STT-MRAM) or phase-change memory (PCM) to build persistent memory. These studies operate directly on nonvolatile data that is accessible through the processor-memory bus, eliminate the overhead of PCIe or SATA accesses and legacy block-oriented file-system interfaces, and update the persistent data structures at cache line granularity without the need for batching. Neither memory (SRAM, DRAM, and flash) nor storage media (hard drives and optical discs) in current commercial systems are both nonvolatile and byte-addressable. Hence, NVRAM-based persistent memory enables a new class of applications that can store pointer-rich, user-defined data structures directly in a nonvolatile memory and process a large amount of data at low latency and high bandwidth.

A caveat for persistent memory design is that system failures or program crashes may corrupt the state of data structures. For instance, a power outage may occur while an application is inserting a node in a doubly-linked list. If only one pointer is written out to nonvolatile devices (NVRAM) and the other is still in volatile devices (processor caches or DRAM), the doubly-linked list will be broken and not usable after the crash. Ideally, a persistent memory system (hardware, software, or a combination of both) must ensure safe data updates so that data integrity is maintained in the presence of system failures or program crashes. Borrowing the ACID (atomicity, consistency, isolation, and durability) [59] concept from the database community, persistent memory systems must update a set of programmer-defined nonvolatile locations in an atomic, consistent, and durable way to enforce crash consistency (i.e., **persistence**).

Unfortunately, supporting persistence in memory still incurs significant performance cost, even with the latest proposals. Existing persistent memory designs employ logging or copy-on-write (COW) to manage persistent data updates. Logging mechanisms track the changes to critical data by maintaining a set of journals, which store old data values (undo logging) or new updates (redo logging). COW stores new updates in a temporary data copy, while the real data is unchanged. However, these mechanisms increase the demand of storage space and reduce system performance by increasing memory traffic with extra data transfers. Furthermore, previous persistent memory designs use instructions such as flush (`clflush`)

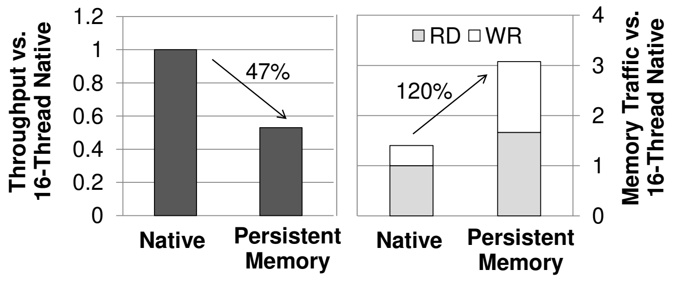


**Figure 5.1.** Comparison between a native system with no persistence support (Native) and log-based persistent memory (Persistent Memory). Speedups of transaction throughput (higher is better) and memory traffic (lower is better), including reads and writes, are averaged across benchmarks.

and memory fence (`mfence`) to ensure consistency by flushing the dirty lines in caches at the barrier of each persistent memory update. As a result, we observe a large performance gap between a system with a persistent memory and a "native system" (i.e., with no persistence support). Persistent memory implementations using off-chip NVRAM and logging incur a 120% increase in memory traffic (60% in reads and 180% in writes) and only achieve 53% of the throughput of a native system (Figure 5.1). Therefore, our goal is to design a persistent memory with performance close to that of the native system.

We propose Kiln[1], a persistent memory design that employs a nonvolatile last level cache and a nonvolatile memory to construct a persistent memory hierarchy. Our design allows a persistent memory system to directly update the real in-memory data structures, rather than performing logging or COW. We refer to these direct updates to the real in-memory data structures as **in-place** updates. We also develop a set of light-weight software and hardware extensions to facilitate atomicity and consistency support. With in-place updates, Kiln can achieve 91% of native system performance, which is about a 2× improvement over log-based persistent memory designs using NVRAM.

[1]"Kiln" was once used by ancient Mesopotamians to bake the clay tablets with temporary scripts and turn them into permanent records. We name our persistent memory design Kiln, because it is analogous to the persistent memory that turns volatile data into permanent records.

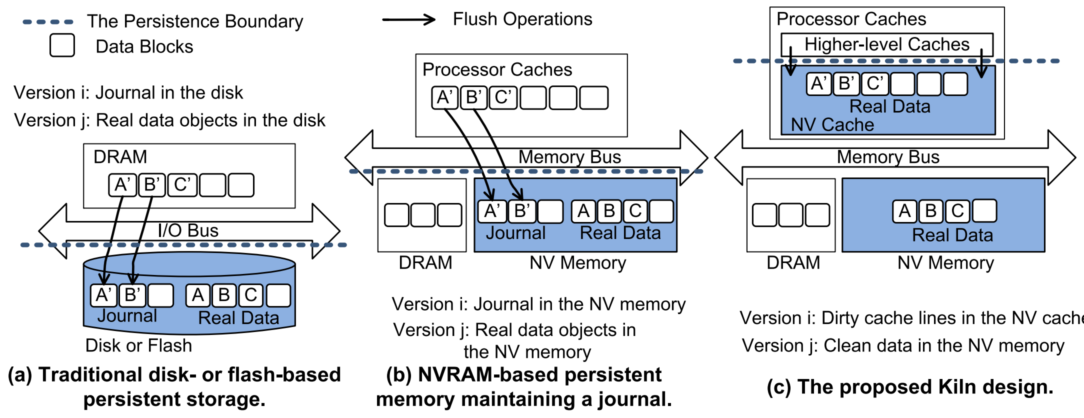

The Persistence Boundary
Data Blocks
Flush Operations
Version i: Journal in the disk
Version j: Real data objects in the disk
DRAM
A' B' C'
I/O Bus
A' B' A B C
Journal Real Data
Disk or Flash
(a) Traditional disk- or flash-based persistent storage.
Processor Caches
A' B' C'
Memory Bus
A' B' A B C
Journal Real Data
DRAM NV Memory
Version i: Journal in the NV memory
Version j: Real data objects in the NV memory
(b) NVRAM-based persistent memory maintaining a journal.
Processor Caches
Higher-level Caches
A' B' C'
Real Data
NV Cache
Memory Bus
A B C
Real Data
DRAM NV Memory
Version i: Dirty cache lines in the NV cache
Version j: Clean data in the NV memory
(c) The proposed Kiln design.

**Figure 5.2.** Overview of Kiln persistent memory design and previous work. Most previous studies ((a) and (b)) employ logging or COW to maintain multiversioning, explicitly duplicating data in a separate journal or temporary buffer data structures. In these studies, ordering is enforced by write-through caching, cache flush and memory fence instructions, or fsync operations. (c) shows an overview of the proposed Kiln design. With the multiversioned memory hierarchy consisting of the NV cache and the NV memory, Kiln allows in-place updates to the real in-memory data structures without logging or COW.

## 5.1 Design Overview

Kiln adopts a new persistent memory architecture consisting of a nonvolatile cache (NV cache) and a nonvolatile memory (NV memory), naturally forming a multiversioned persistent memory hierarchy (Figure 5.2 (c)). The newly updated versions are dirty NV cache lines. The old versions are clean data stored in the NV memory, which will be automatically updated when the dirty NV cache lines are evicted. With this multiversioned persistent memory hierarchy, Kiln simplifies persistent memory update operations by allowing memory stores to be performed in-place to the persistent data structures in the NV cache, without logging or COW. Therefore, Kiln's memory store operations are similar to those of the native system. As a result, Kiln's performance is also very close to that of the native system, yielding a significant performance improvement over previous NVRAM-based persistent memory designs. Table 2.1 qualitatively compares Kiln with the native system and related persistent memory designs in terms of memory update mechanisms and support of atomicity and ordering.

### 5.1.1 Assumptions and Definitions

**Mapping data to a hybrid memory address space:** We assume that DRAM and NVRAM are both deployed on the processor-memory bus and mapped to a single physical address space. Kiln stores the user-defined critical data in NVRAM. The DRAM is used to store data that is not required to be persistent and can be overwritten frequently. Examples of such data are stacks and data transfer buffers. Runtime systems such as the ones developed by prior studies [70, 69] can be employed to expose the NVRAM address space to persistent data objects.

**Program hints on persistent memory transactions:** Kiln adopts program hints to decide when and what data blocks need to be persistent. Recent NVRAM-based persistent memory designs [69, 70] obtain this information by allowing users to define durable STM transactions. Similarly, Kiln exposes to programmers an interface of "persistent memory transactions", which are groups of instructions performing persistent memory updates. Kiln reads users' input to define the beginning and end of each transaction.

**States of a persistent memory transaction:** Each persistent memory transaction will go through three states: *in-flight*, *committing*, and *committed*. After the first instruction of a persistent memory transaction starts execution, the transaction becomes an *in-flight* transaction. When the last instruction of the transaction completes execution, the transaction is in *committing* state. In this state, Kiln will perform the clean-on-commit operation (Section 5.1.3) and update the state of persistent data structures (Section 5.1.4). When these operations are completed, the transaction is *committed*. All data updated by this transaction is now persistent.

### 5.1.2 In-place Updates without Logging or COW

Previous persistent memory designs maintain multiversioning by software, using application or OS libraries. From the perspective of software, a memory system is a flat address space, consisting of a sequence of pages. Therefore, previous persistent memory designs need to explicitly create multiple regions, logs or temporary data copies, to maintain multiple versions of data. Different from software, hardware views a memory system as a hierarchy with multiple levels of processor caches and a main memory. This hierarchy naturally stores different versions of data in different levels.

Leveraging this hierarchy, we design a multiversioned persistent memory that includes a last-level NV cache and a NV memory (Figure 5.2(c)). The dirty NV cache lines are one version and the clean data in the NV memory are another. Both versions have the same address so this persistent memory hierarchy directly performs in-place updates to real data structures. We allow in-flight and committing persistent memory transactions to overwrite data values in processor caches (including the NV cache), but not in the NV memory. Therefore, the version stored in the NV memory is persistent if a system crashes when a persistent memory transaction is executing or committing. We allow NV cache lines of committed persistent memory transactions to be written back to the NV memory. However, we do not allow evictions from higher-level volatile caches to overwrite a NV cache line that is being written back. Therefore, the version stored in the NV cache is persistent if a system crashes when writing back a NV cache line.

Our work is different from previous work that use a disk cache or flash buffer to improve the persistence performance, such as eNVy [73], and UBJ [72]. The file cache and flash buffer in these designs are simply used as buffers of the journal or temporary copies of data which still serve for logging or COW, rather than as a way of enabling in-place updates.

### 5.1.3 Ordering Control by Clean-on-commit

We employ an optimized flushing operation called *clean-on-commit* to preserve the ordering of persistent memory updates, when a persistent memory transaction is committing. Unlike previous work, Kiln allows cache controllers to issue flush requests without explicitly executing instructions such as `clflush` or `mfence`. We allow out-of-order write-backs of any dirty cache lines in the volatile caches, including those being updated by in-flight persistent memory transactions. The cache controllers will track the dirty cache lines that are updated by an in-flight persistent memory transaction and still remain in the volatile caches. The architecture extension in the NV cache (Section 6.6) will track the dirty NV cache lines updated by an in-flight persistent memory transaction. When a persistent memory transaction commits, typically a large portion (demonstrated in Section 6.8) of its dirty cache lines have already been written to the NV cache. Therefore, only the remaining dirty cache lines updated by the transaction in volatile caches need to be flushed. After all the dirty cache lines that belong to the committing transaction are flushed to the NV cache, the state of the transaction transitions from committing to committed.

The clean-on-commit operation is improved over the ordering mechanisms of previous designs in four aspects. First, clean-on-commit only flushes the volatile dirty cache lines of the committing persistent memory transactions. Many previous designs [69, 71] employ flushing instructions (e.g., `clflush`) that unnecessarily flush the dirty cache lines out of the cache hierarchy. Second, the memory traffic to perform the flushes is significantly reduced because we only flush a small number of cache lines. Third, the bandwidth of processor-cache buses is much higher than that of the off-chip memory bus, and therefore the flush operations can be completed much faster. Finally, clean-on-commit will be issued in the same order as the commits of persistent memory transactions, and therefore does not employ memory fence or barrier instructions which block other memory accesses. However, clean-on-commit requires bookkeeping functionality to be added to the volatile cache controllers. We will discuss the mechanisms and the overhead in Section 6.6.

### 5.1.4 Timeline of a Transaction

With in-place updates and clean-on-commit, Kiln provides a way to reduce the latency of data persistence by committing the persistent memory transactions right after all the updates arrive at the NV cache, rather than waiting for the updates to be flushed to the NV memory. Figure 5.3 shows the execution timeline of Kiln compared to that of a persistent memory system with redo logging. We do not show an example with undo logging, because its performance is usually worse than that of redo logging.

Figure 5.3(a) shows the sequence of updating persistent memory that employs redo logging to a journal in the NV memory. An in-flight persistent memory transaction keeps adding new data values and their addresses to a journal. This is followed by flush and memory fence operations to ensure that all the journal updates reach the NV memory immediately after they are issued. A persistent

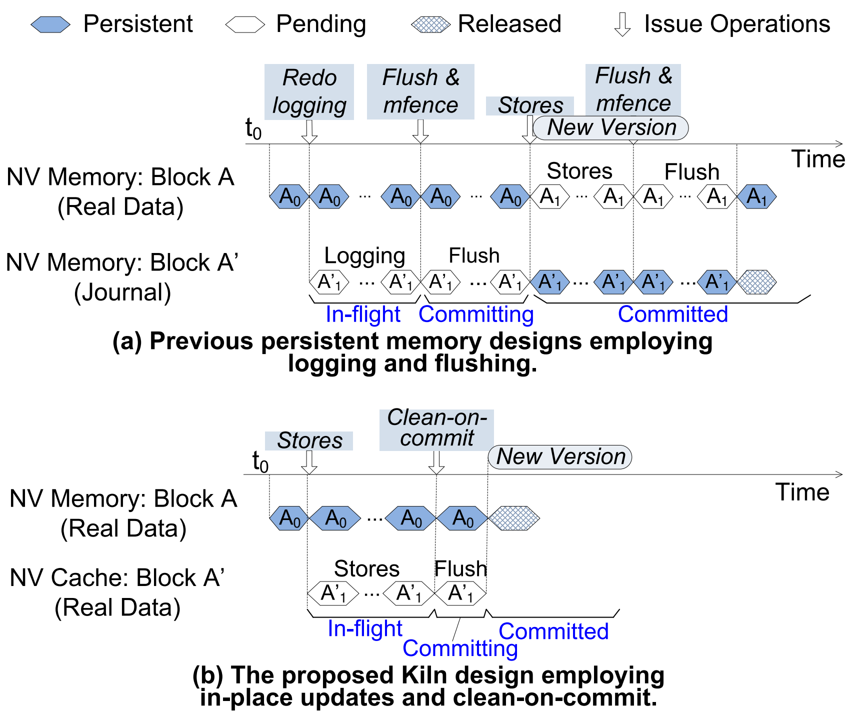

Persistent
Pending
Released
Issue Operations
Redo logging
Flush & mfence
Stores
Flush & mfence
New Version
$t_0$
Time
NV Memory: Block A (Real Data)
Stores
Flush
NV Memory: Block A' (Journal)
Logging
Flush
In-flight
Committing
Committed
(a) Previous persistent memory designs employing logging and flushing.
Stores
Clean-on-commit
New Version
NV Memory: Block A (Real Data)
NV Cache: Block A' (Real Data)
Stores
Flush
In-flight
Committing
Committed
(b) The proposed Kiln design employing in-place updates and clean-on-commit.

**Figure 5.3.** Comparison of the timeline of Kiln and previous persistent memory designs. Block $A$ represents the data block (with a size of multiple cache lines) of an old valid version. Block $A'$ represents the new version being updated.

memory transaction becomes committed after the last instruction in a transaction is executed and all the logs are flushed into the NV memory. Then, the system can overwrite the real data structures in the NV memory. Figure 5.3(b) shows the timeline of Kiln. After executing the last instruction in an in-flight transaction, the state of the transaction becomes committing. *Committing* a persistent memory transaction consists of two steps. First, Kiln performs clean-on-commit to flush all the corresponding dirty cache lines remaining in volatile caches. Then, Kiln updates the state of every corresponding NV cache line, from uncommitted to committed. After these two steps are completed, a persistent memory transaction becomes *committed*.

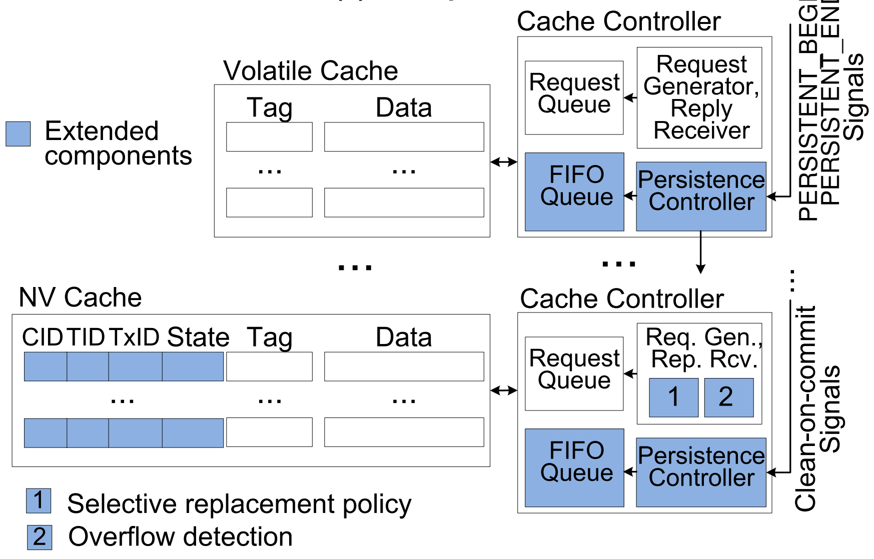
Compared with redo log based persistent memory, Kiln executes faster with both a single persistent memory transaction and a sequence of them. As discussed in Section 5.1.3, clean-on-commit is much more efficient than executing flush and memory fence instructions. Therefore, Kiln completes a single persistent memory transaction faster than the redo logging method, despite the longer last-level

```
persistent(inorder) {
   read x1, x2, x3;
   do some processing;
   write y1, y2;
}

persistent(inorder) {
   write py1, py2;
}
```

```
persistent {
   read x1, x2;
   do some processing;
   write z1, z2;
}
```

**(a) Example code.**



**(b) Cache architecture extensions.**

**Figure 5.4.** Software and architecture extensions developed to facilitate Kiln. (a) A code example with Kiln software interface. (b) Cache architecture extensions. The shaded blocks are the modifications required over conventional architecture. Note that the dirty bit, the invalid bit, and other cache coherence information are included in the original tag region.

cache (NV cache) access latency. Kiln also executes much faster than redo log based persistent memory when running a sequence of transactions. The redo logging mechanism only flushes log updates when a transaction is committing. The real data updates of a committed transaction can still remain in volatile caches. Therefore, the NV memory needs to keep the log updates after a transaction is committed, until all the real data updates arrive at the NV memory. As a result, a redo log based persistent memory needs to periodically perform a truncation operation, which flushes real data updates from caches to the NV memory and then releases (free of reclamation) the corresponding log entries. Instead, Kiln releases NV memory data blocks right after the corresponding transaction is committed.

| (Bits) | L1 | L2 | L3 |
|---|---|---|---|
| CID | 0 | 0 | 3 |
| TID | 1 | 1 | 1 |
| TxID | 5 | 5 | 8 |
| State | 2 | 2 | 2 |
| Way | 2 | 3 | 4 |
| Set | 8 | 9 | 16 |
| Entries | 16 | 16 | 128 |
| Total (bytes) | 288 | 320 | 544 |

**(a) Storage overhead of the FIFO queues.**

| Miss casued by | Cache set filled by | Detect |
|---|---|---|
| In-flight Txi | In-flight Txi | Overflow |
| In-flight Txi | In-flight Tx | Set timer |
| Non-persistent | In-flight Tx | Set timer |

**(b) Detection of NV cache overflow.**

**Figure 5.5.** The storage overhead of the FIFO queues added to the cache controllers. Note that the L1 and L2 caches are private so the total overheads are calculated as the sum of eight FIFO queues.

Therefore, Kiln reduces the total time of completing a sequence of persistent memory transactions by eliminating the truncation operations.

### 5.1.5 Discussion

**Durable TM transactions:** Persistent memory transactions are similar to file system and database transactions, which make atomic, consistent, and durable modifications to the storage system. TM, a concurrency control mechanism which also borrows the concept of “transaction” from the database community for controlling shared memory access, also supports atomic and consistent memory accesses. However, directly enabling durability with TM is suboptimal for persistent memory updates, if not impossible. STM records every speculative store in a log. Therefore, employing STM with durable memory transactions still requires maintenance of a journal. For example, recent studies employing STM for persistent memory updates, including Mnemosyne [69] and NV-heaps [70], both maintain a redo log in the persistent memory. Another type of TM implementation, hardware transactional memory (HTM), does not necessarily require logs. Commodity HTM implementations, such as the transactional synchronization extensions specified by the Intel Haswell processor [125] and the transactional memory processor instructions supported by the IBM zEC12 [126], buffer speculative stores at processors' private caches (in particular, the L1 caches) and overwrite the lower-level caches and memory when transactions commit. These HTM implementations need to support fast recovery from transaction aborts, and therefore ensure atomicity only at higher-level caches. Unless the entire cache hierarchy is made nonvolatile, it is impossible to ensure atomic updates crossing the persistence boundary by directly adopting these HTM implementations. Other HTM implementations, such as IBM Blue Gene/Q's hardware support for TM [127] and LogTM [128], allow the speculative stores to enter the lower-level caches. However, they have other downsides. The IBM Blue Gene/Q [127] requires write-through L1 caches or invalidating the entire L1 cache at the beginning of each transaction. LogTM [128] maintains a hardware-based undo log to buffer the speculative stores. Recovery with the persistent memory from system failures is performed off-line or off the critical path of program execution, and therefore can tolerate much longer recovery latency. Employing durable HTM transactions to update the persistent memory can be unnecessarily cumbersome and inflexible. With Kiln, race-free isolated data accesses in multi-threaded or multi-process programs can be guaranteed by TM or any other concurrency control mechanisms, such as mutexes, semaphores, or lock-free/wait-free data structures and algorithms.

**Critical-data persistence vs. whole-system persistence:** Kiln supports persistence for user-defined critical data structures typically used in databases or file systems, such as search trees, hash tables, and graphs. This is especially useful for servers running database and file system services. Another research direction focuses on the persistence of the entire system, called whole-system persistence (WSP) [94], supporting instant program restart or resuming after failures. This method makes a persistent copy of the entire memory upon failures, by employing flush-on-fail, i.e., flush all register and cache states to the NV memory. With sufficient backup power sources, a system employing Kiln can also provide high-performance WSP support by mapping all the data to the NV memory address space and performing the same flush-on-fail operation.

## 5.2 Implementations

This section addresses the implementation details. First, we provide a software interface for users to define the boundary of a persistent memory transaction. Second, we provide a finite-state machine for every NV cache line to ensure that the persistent memory is in a consistent valid state with only the committed transaction data. Third, we implement a set of cache architecture extensions, including the extended tags and the selective replacement policy at the NV cache, and track logic and FIFO queues in the cache controllers. Fourth, we provide a solution to detect the NV cache overflow and present a fall-back path to resolve the overflow. Finally, we will discuss the physical implementation choices, including the memory technologies used in the NV cache and the NV memory and integration technologies.

### 5.2.1 Software Interface and ISA Extension

To define the beginning and end of a persistent memory transaction, we provide the software interface,

```
persistent{...}
```

to define persistent memory transactions. Furthermore, we provide a software interface that allows the users to declare strong and relaxed ordering control. The strong ordering is denoted by

```
#pragma persistence_inorder
```

With the strong ordering control declared, Kiln applies clean-on-commit for each persistent memory transaction. Without this declaration, the users can specify the transactions that require ordering with an attribute called *inorder*, i.e., using

```
persistent(inorder){...}
```

Ordering is maintained within persistent memory transactions with the inorder attribute. Clean-on-commit operations on transactions without this attribute may be delayed. Figure 5.4(a) shows an example of using the software interface with relaxed ordering control. In this example, the pointers `py1` and `py2` will be updated after the updates to their data objects `y1` and `y2` are flushed to the NV cache. The updates to `z1` and `z2` may remain in volatile caches without being forced to the NV cache.

We also extend the ISA with a pair of new instructions, `PERSISTENT_BEGIN`

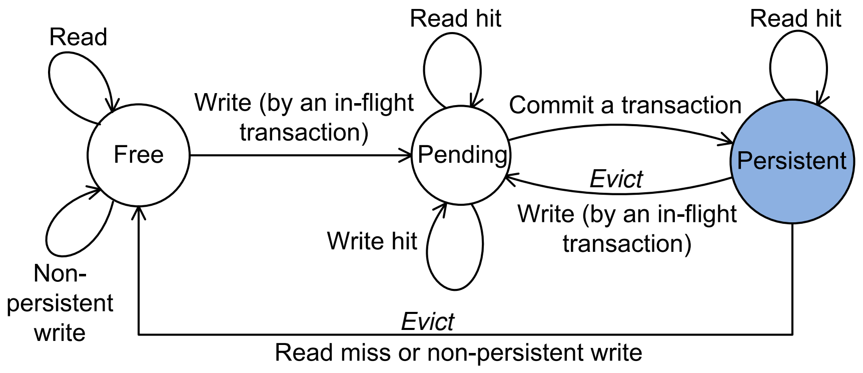

Read
Free
Write (by an in-flight transaction)
Read hit
Pending
Write hit
Commit a transaction
Evict
Write (by an in-flight transaction)
Read hit
Persistent
Non-persistent write
Evict
Read miss or non-persistent write

**Figure 5.6.** The state transition of NV cache lines.

and PERSISTENT_END. The software interface can be translated to ISA instructions with simple modifications to the compiler. Similar ISA and software interface extensions have been implemented to support HTM, such as those of Intel's Haswell processors [125] and IBM's zEC12 [126]. We provide a separate set of extensions with persistent memory transactions so that HTM can be simultaneously used as the concurrency control mechanism.

## 5.2.2 Maintaining the State of NV Cache Lines

The NV cache is shared by *non-persistent* cache lines (mapped to the DRAM address space), the cache lines being updated by in-flight persistent transactions, and the cache lines with the committed transactions. Each NV cache line is assigned one of three states: free, pending, and persistent (Figure 5.6). A *free* cache line stores non-persistent data mapped to the DRAM address space. A *pending* cache line is updated by an in-flight persistent memory transaction, storing the new data value. A cache line with the latest version of a committed persistent memory transaction is called *persistent*. As shown in Figure 5.6, various access events at a NV cache line can trigger state transitions of the cache line. Note that read or write misses do not apply to a pending cache line, due to our selective replacement policy presented in Section 5.2.3. Although the state transition can be integrated with a cache coherency protocol, doing this can increase the complexity of maintaining coherence. Therefore, we maintain the state transition separately.

### 5.2.3 Cache Extensions

We develop a set of cache architecture extensions (Figure 5.4(b)) to facilitate Kiln, including additional regions in the NV cache tags, a selective replacement policy, and tracking logic and tables in the cache controllers.

**Additional regions in the NV cache tags:** We add four additional fields to each cache tag, including the core ID (CID), the hardware thread ID (TID), the persistent memory transaction ID (TxID), and the cache line state. The first three IDs are used to distinguish between different persistent memory transactions initiated by different processor cores. The cache line state is used to maintain the state transition among the states of *free*, *pending*, and *persistent*. The storage overhead of each tag entry is $log_2 N + log_2 T + log_2 M$, plus 2 bits for the cache line state. Here $N$ and $T$ are the number of cores and hardware threads per core, and $M$ is the number of maximum in-flight persistent memory transactions supported by Kiln. If strong ordering is enforced (i.e., `#pragma persistence_inorder` is declared), the number of in-flight persistent memory transactions is limited by the total number of hardware threads, i.e., $N \times T$. The TxID of a persistent memory transaction can be reused after it is committed. We can estimate the storage overhead in the NV cache tags with the following case. If we support 256 in-flight persistent memory transactions on a processor with eight cores and two hardware threads per core, we need an additional 14 bits in each NV cache tag, which only adds 2.7% to a 64-byte cache line. If strong ordering control is enforced, the maximum number of in-flight persistent memory transactions is far less than 256, 16 in this example.

**Selective NV cache replacement policy:** Existing cache replacement policies are not designed for data persistence. To prevent the in-flight persistent memory transactions from corrupting the data structures stored in the NV memory, we implement a simple selective NV cache replacement policy extension: we do not allow the evictions of pending cache lines. Read and write misses at pending cache lines are thus not allowed in Figure 5.6. Our extension can work with most existing cache replacement policies. In practice, we adopted LRU as the basic replacement policy in Section 6.8.We leave the exploration of more sophisticated optimizations of cache replacement policy as future work.

**Tracking in-flight persistent memory transactions in cache controllers:** We extend cache controllers with FIFO queues and persistence controllers, as illustrated in Figure 5.4. The FIFO queues are used to track all the dirty cache lines updated by in-flight persistent memory transactions. Each FIFO queue entry is a copy of the extended tag information (CID, TID, and TxID) and the location of a dirty cache line (its set and way number). We evaluated the storage overhead of FIFO queues in a cache hierarchy described in Table 6.4 (choose option (b) for L3 cache). We employ the number of FIFO entries that is sufficient to accommodate the workloads described in Table 6.5: the FIFO queues at each L1 and L2 cache have 16 entries; the one at the L3 cache has 128 entries. Figure 5.4(c) lists the storage overhead of the FIFO queues at each L1, L2, and L3 cache. Note that the storage device in cache controllers is volatile for fast access and easy fabrication. The information stored in the FIFO queues will be lost if the processor loses power. In this case, all the in-flight persistent memory transactions need to be re-executed after the system restarts. Persistence controllers are in charge of enqueuing the FIFO and issuing the clean-on-commit operations. They also allocate TxIDs to the new persistent memory transactions. The persistence controller at the L1 cache controllers are extended to detect the boundary of each persistent memory transaction, by receiving the `PERSISTENT_BEGIN` and `PERSISTENT_END` signals from the processor cores. The request generator in the NV cache controller is extended to implement the selective replacement policy and the overflow detection mechanisms.

### 5.2.4 NV Cache Overflow and Fall-back Path

NV cache overflow is the case when a miss at the NV cache can never be serviced because no victim can be found for replacement. In this case, the program cannot make forward progress without the NV cache overflow being resolved. Because we do not allow pending cache lines to be evicted from the NV cache, the overflow may be caused by one of two reasons: (1) the *capacity* is smaller than the total size of in-flight persistent memory transactions or (2) the *associativity* is insufficient to accommodate all in-flight persistent memory transactions that conflict at the same cache set.

**Table 5.1.** Parameters of the evaluated multi-core system.

| Processor/Technology | Intel Core i7 like/22 nm |
|---|---|
| Cores | 8 (2.5GHz), 16 threads |
| L1 Cache (Private) | Volatile (SRAM), 64KB, 4-way, 64B blocks, 1.6ns latency |
| L2 Cache (Private) | Volatile (SRAM), 256KB, 8-way, 64B blocks, 4.4ns latency |
| L3 Cache (Shared) | (a) Volatile (SRAM), 16MB, 16-way, 64B blocks, 10ns latency<br>(b) Nonvolatile (STT-MRAM), 64MB, 16-way, 64B blocks, 15ns (19ns) read (write) latency |
| Memory Controller | Two dual-channel memory controllers, FR-FCFS |
| Memory Technology | 30 nm |
| DRAM DIMM | DDR4-2133, 2GB |
| NV Memory DIMM | STT-MRAM, 2GB, 25ns row-hit latency,<br>65ns (76ns) read (write) row-conflict latency |
| Power and Energy | Processor (with L1 and L2): 149W (peak).<br>L3 (SRAM): read/write: 0.58nJ/access;<br>L3 (STT-MRAM): read (write): 0.61 (0.67) nJ/access.<br>NV memory : row buffer read (write): 0.93 (1.02) pJ/bit,<br>array read (write): 1.00 (2.89) pJ/bit |

**Detecting NV cache overflow:** We can detect an NV cache overflow when searching for an eviction victim at the NV cache. Figure 5.4(d) lists the scenarios which can lead to NV cache overflows. NV cache overflows are hard to detect if the cache set is filled by a mix of different in-flight persistent memory transactions. It is possible that the program can continue to make progress after one of the in-flight persistent memory transactions is committed and advance one of the cache lines in the set to the *persistent* state. Unfortunately, simply waiting for next available victim will incur performance overhead and even deadlocks. Instead, we stall memory requests when the request queue at the higher level cache is almost full (e.g., 80% filled) and then provide a fall-back path.

**Fall-back path:** We provide a fall-back path to resolve the issue of NV cache overflows, allowing the pending cache lines to be written back to the NV memory and maintain multiversioning in the NV memory with *hardware-controlled COW* similar to that used in eNVy [73]. When an NV cache overflow is detected, Kiln will notify the operating system by interrupt to allocate new pages to buffer the pending cache lines evicted from the NV cache. A mapping table will be created in the NV memory and updated with the physical addresses of buffered pending cache lines. When a persistent memory transaction is committed, the page table will be updated to invalidate the old data values and enable the new data values according to the mapping table. Then the corresponding mapping table entries can be discarded.

Commodity processors typically employ several megabytes of last-level cache with high associativity (e.g., 16-way). The density of NVRAM is much higher than SRAM, so the capacity of the NV cache can be as large as tens of or over one hundred megabytes. The associativity of the NV cache can also be higher than SRAM-based caches. Therefore, Kiln can support in-flight persistent transactions with memory footprints up to tens of megabytes. The memory footprints of the in-flight persistent memory transactions are determined by the granularity of modifications performed to the persistent data structures and upper-bounded by the size of data structure elements (e.g., tree nodes, table entries, graph edges, etc.). Furthermore, small-granularity data updates may dominate some commercial and future real-world workloads. For example, several key-value workload characteristics published recently by Facebook [129] showed that most queries employ keys of less than 32 bytes and values of no more than a few hundred bytes. For this type of workload, NV cache overflow will be less of an issue.

### 5.2.5 Recovery

Kiln allows easy and fast system recovery mechanisms, because most of the persistent updates are applied in-place to the real in-memory data structures. Upon restart from an abnormal termination, the system can go through the following steps for recovery. First, we scan the NV cache tags and invalidate the cache lines in the pending state because they are partially updated data structures in process by in-flight persistent memory transactions before failure. Next, we scan the page table in the NV memory to identify the temporary data copies (if any) due to NV cache overflows. These data copies were updated by in-flight memory transactions as well, and hence can be invalidated. These recovery steps can be performed by hardware, reusing the tracking logic and FIFO queues in cache controllers.

### 5.2.6 Physical Implementation

In principle, our persistent memory architecture design does not rely on any specific physical implementation of processors and memories. For example, all components of the processor and memory can be packaged in a single package with silicon interposer technology, which has been widely explored by academia and industry to develop high-performance system-in-package designs [41, 45]. The NV cache and the NV memory can both be implemented by STT-MRAM, which provides the best latency and endurance among NVRAM technologies. Everspin [40] recently launched the DDR3 compatible STT-MRAM components, which is projected to be able to scale to Gb densities (close to NAND flash). Existing work has demonstrated the feasibility of STT-MRAM used in lower-level caches [1] in multi-core processors. The NV cache can be stacked on top of the CPU die for large capacity and high bandwidth, or packaged with the NV memory, sitting beside the processor with higher-level caches. In this case, the processor can be fabricated without the effort of integrating different memory technologies. We can also implement the NV memory with resistive RAM (ReRAM) or PCM, because they are byte-addressable and nonvolatile just like STT-MRAM. The main memory, including the NV memory and DRAMs, can be implemented with an off-chip DIMM interface or wide I/O interface [130, 131]. The wide I/O implementation can achieve higher memory bandwidth between the processor and the main memory for better performance, however it incurs complexity and higher cost.

## 5.3 Experimental Setup

We evaluated the performance and power of our persistent memory design on a multi-core system. This section describes our simulation framework, processor and memory configurations, and benchmarks.

### 5.3.1 Simulation Framework

Our experiments are conducted using McSim [132], a Pin-based multi- and many-core cycle-accurate simulation infrastructure [133]. McSim models out-of-order cores, caches, directories, on-chip networks, and memory channels. Table 6.4 lists the detailed parameters and architecture configurations of the processor and memory system in our simulation. The multi-core processor consists of eight out-of-order cores, each of which is similar to one of the Intel Core i7 cores [134]. Each

**Table 5.2.** Benchmarks used in our experiments.

| **Benchmarks** | **Description** |
|---|---|
| BTree [137] | Inserts/deletes nodes in a B-tree. |
| Hash [70] | Inserts/deletes entries in a hash table. |
| RBTree [70] | Inserts/deletes nodes in a red-black tree. |
| SDG [138] | Inserts/deletes edges in a scalable large graph. |
| SPS [70] | Random swaps between entries in an array. |
| SSCA2 [139] | A scalable large graph analysis benchmark. |

processor core incorporates SRAM-based volatile private L1 and L2 caches. Kiln employs an STT-MRAM based L3 cache (the NV cache) (option (b) in Table 6.4). Option (a) in Table 6.4 lists the parameters of a system with SRAM as L3 cache, which is used to validate the performance of Option (b). Note that the parameters of the two systems are calculated based on the same silicon area, i.e., a 16MB SRAM-based cache occupies the same silicon area of 64MB STT-MRAM based cache. Both L3 caches are 16-way set-associative and multi-banked. The processor cores and L3 cache banks communicate with each other through a crossbar interconnect. A two-level hierarchical directory-based MESI protocol is employed to maintain cache coherence at the private caches and the L3 cache. The DRAM and the NV memory are modeled as off-chip DIMMs. Memory requests to DRAM and the NV memory are managed by two dual-channel memory controllers. The timing and energy parameters of the NV cache and NV memory are calculated with NVSim [135], a performance, power, and area estimation tool for NVRAM.

Our simulation framework models Kiln's in-place updates, clean-on-commit functionality, and architecture extensions. We also model HTM based on Hammond et al.'s work [136] as one of our two concurrency control mechanisms used in the experiments. The implementations of most commodity HTM, e.g., Intel's Haswell processor [125] and IBM's zEC12 processor [126], are similar to Hammond et al.'s work. The memory footprint of transactions is limited up to the capacity of private caches. Overflow at the private caches will result in transaction abort (re-execution) or transferring the control to software.

### 5.3.2 Benchmarks

The persistence interface of most existing software applications are optimized for accesses to disk-based storage devices. Currently, no existing public benchmark suites can be used to evaluate the Kiln design. Therefore, we constructed a set of benchmarks as described in Table 6.5. The data structures and functionality of these benchmarks are similar to those in the benchmark suite used by NV-heaps [70]. The benchmarks perform search, insert, and delete to data structures used in databases and file systems, including a search tree, hash table, sparse graph, and array. Two sets of experiments are conducted to insert and delete the data elements (tree nodes, table entries, graph edges, etc.) with small (512 bytes) and large (512 kilobytes) granularity, respectively. They will be referred to as workloads of small and large footprints in the rest of the dissertation. Each persistent memory transaction inserts or deletes a single data element. The benchmarks are written with the strong ordering control interface (Section 6.6) to force all the transactions to commit inorder. HTM is used as the concurrency control mechanism for workloads of small footprint, while mutex lock is used for workloads of large footprint. We also implemented another version of the benchmarks, which perform undo and redo logging at word granularity to provide persistence support. We only evaluate the hardware performance of various persistent memory designs, so we do not count the latency of executing the logging instructions. We collect the performance and power results of the running phase of the benchmarks, skipping the initialization phase.

## 5.4 Results

This section presents our evaluation results and analyze the reasons for these results.

### 5.4.1 Volatile Vs. Nonvolatile Last-level Cache

We first compare throughput (in terms of the executed insert/delete operations per second) of two systems with the L3 cache implemented by SRAM and STT-MRAM, without providing persistence support (Figure 5.7). Despite its lower latency, the

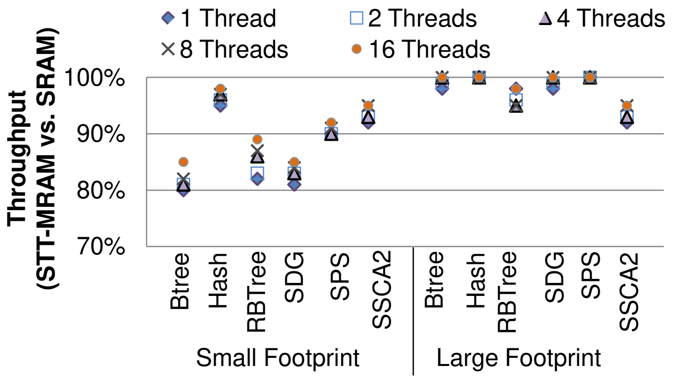


**Figure 5.7.** Performance comparison between two native systems adopting STT-MRAM and SRAM as L3 cache respectively. Results show that the two systems have similar performance.

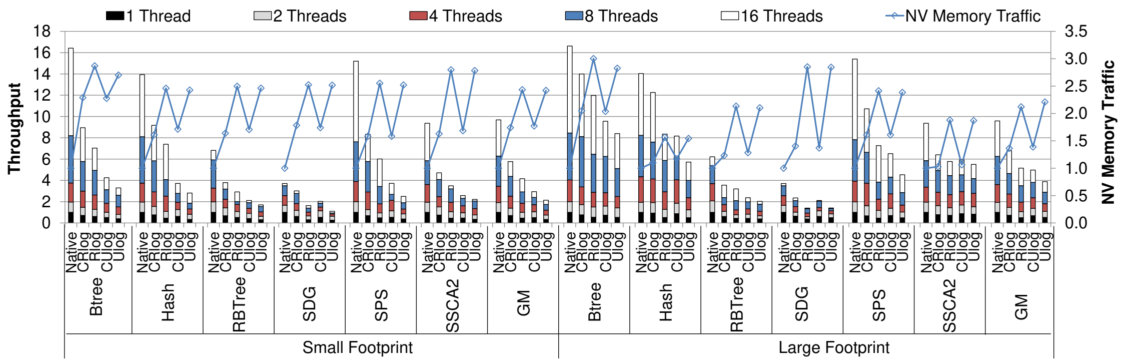


**Figure 5.8.** Performance of systems that adopt a NV L3 cache, but with logging for atomicity and flush and memory fence for ordering. We evaluate the throughput (bars) and NV memory traffic (broken lines). All the throughputs are normalized against the native system running 1 thread. For NV memory traffic, we only show the normalized results running 16 threads.

SRAM-based last-level cache is only a quarter the capacity of STT-MRAM based cache on the same silicon area. Our results show that using an STT-MRAM based L3 cache can achieve on average 91% and 99% of the performance using SRAM-based L3 cache on workloads of small and large footprints, respectively. These results show that employing NV cache in a non-persistent manner as the last-level cache does not remarkably change the system performance due to the latency and capacity trade-offs of SRAM and STT-MRAM technologies. In the following experiments, we use the configuration of STT-MRAM based L3 cache as the baseline native system.

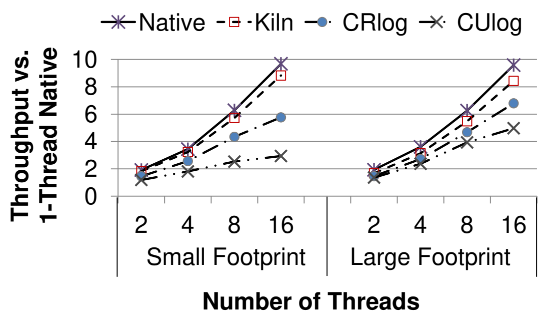


**Figure 5.9.** Performance gap vs. number of threads.

## 5.4.2 Log-based Persistent Memory Performance

A log-based persistent memory system can adopt a NV L3 cache, with logging to ensure atomicity and flush and memory fence to ensure ordering. In this system, the logs become persistent once they arrive at the NV cache. We want to demonstrate that the performance of such an optimized log-based system is not scalable as the number of threads increases.

A log-based system can adopt two types of logs, redo and undo logs. We denote the resultant systems as CRlog and CUlog, respectively. Rlog and Ulog denote the systems where the logs are only stored in the NV memory. CXlog uses Kiln's cache controller extensions to track the dirty cache lines of logs and flush them into the NV cache. Xlog uses `clflush` and `mfence` to write logs in to the NV memory. However, the latency of executing these two instructions is not counted as discussed in Section 6.7. Figure 5.8 shows the comparisons between CXlog (CRlog and CUlog) and Xlog (Rlog and Ulog) for throughput of insert/delete operations and NV memory traffic. The results show that the throughput of CRlog and CUlog increases by an average of 38% and 33% compared with Rlog and Ulog, with workloads of small footprints running 16 threads. The corresponding NV memory traffic is reduced by 28% (CRlog) and 26% (CUlog) on average. With workloads of large footprints, the average improvement of throughput is 31% (CRlog) and 28% (CUlog). The corresponding NV memory traffic reductions are 35% and 37%.

While CXlog significantly reduces the number of accesses to the NV memory, it can still incur an over 50% increase in the memory traffic compared with the native system (denoted as the Native). In addition, the throughput of CXlog does not scale well when the number of threads increases from two to 16 (Figure 5.9).

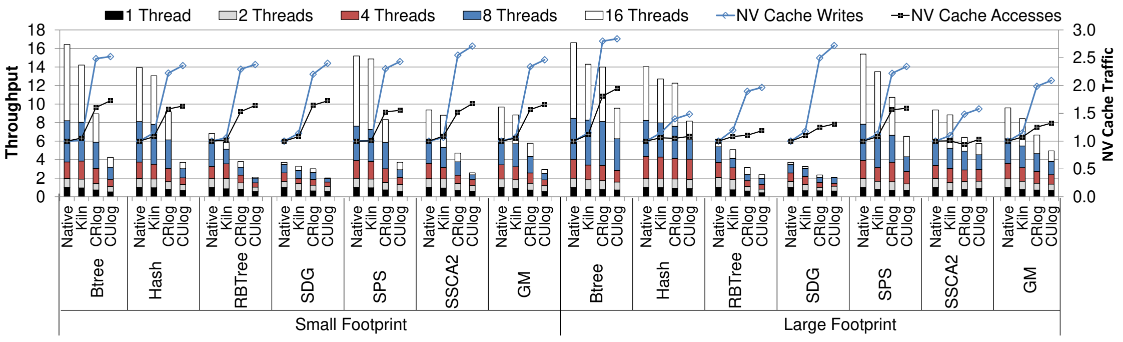


**Figure 5.10.** The throughput (bars) and NV cache traffic (broken lines) of Kiln. All the throughputs are normalized against the native system running 1 thread. For NV cache traffic, we only show the normalized results running 16 threads.

With two threads, the performance gap between CUlog and Native is less than 38% and 30% with small and large footprints, respectively. However, when the number of threads increases to 16, this performance gap also significantly increases up to 70% and 50% with small and large footprints, respectively. CRlog performs better than CUlog, however, the performance gap still increases from around 25% to 45% when the number of threads increases from two to 16. With a large number of threads running concurrently, the log size grows quickly and the NV cache will soon be filled by logs. Furthermore, the logs in the NV cache, which will not be reused anymore, can also lead to early evictions of reusable cache lines of the real data structures. Sophisticated replacement policies can be employed to prioritize the evictions of logs. However, this will be equivalent to bypassing the NV cache or flushing the logs all the way down to the NV memory.

### 5.4.3 Kiln Performance

The following experiments evaluate Kiln performance in terms of the throughput of insert/delete operations.

**Throughput and NV cache traffic:** For workloads of small and large footprints running 16 threads, Kiln achieves on average 91% and 88% of the throughput of the Native system (Figure 5.10). Therefore, the performance of Kiln is 1.6× and 3× of that of CRlog and CUlog with workloads of small footprints, and 1.2× and 1.5× of that of CRlog and CUlog with workloads of large footprints. Kiln performs worse for workloads of large memory footprints because the large number

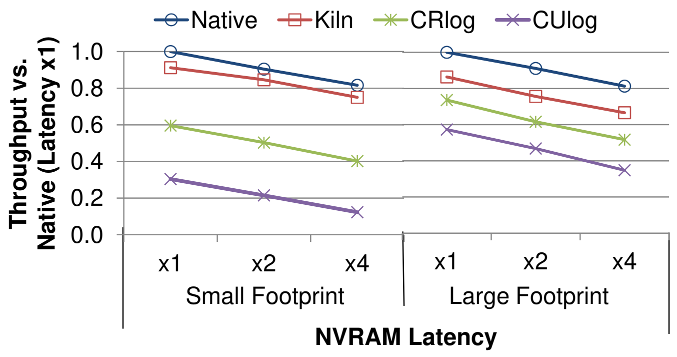


**Figure 5.11.** Throughput of insert/delete operations of 16-thread workloads with longer NVRAM latencies, normalized to the Native throughput with ×1 latency and 16 threads.

of pending cache lines (not allowed to be evicted to the NV memory) leads to early evictions of other reusable cache lines. Although CRlog allows the persistent data to be updated immediately after the logs reach the NV cache, it still does not perform as well as Kiln because CRlog needs to maintain the ordering of the log updates with `clfush` and `mfence`, which prevent the cache controllers from re-ordering the memory requests and block subsequent loads and stores. While log-based persistent memory designs double the write traffic to the NV cache, Kiln only generates 8% additional writes and 5% additional total accesses in NV cache traffic compared to the Native, due to clean-on-commit operations.

**Sensitivity to NVRAM latency:** The evaluations above are conducted with fixed NV cache and the NV memory latencies. We also evaluated the performance variation with longer NVRAM latencies. Figure 5.11 shows the results of normalized throughput with doubling and quadrupling the original NV cache and NV memory latencies (the NV memory clock rate is determined accordingly), averaged across the benchmarks running 16 threads. We observe that the benefit of Kiln remains at longer NVRAM latencies for workloads of both small and large footprints. Even with quadrupled NVRAM latency, Kiln still achieves 92% and 82% of Native throughput with workloads of small and large footprints.

**Frequency of NV cache overflow:** The frequency of NV cache overflow significantly affect system performance. Here we study the frequency of NV cache overflow by further increasing the memory footprints of the persistent memory transactions. We count the number of NV cache overflows during 100K persistent memory transactions inserting and deleting to a hash table. The keys are

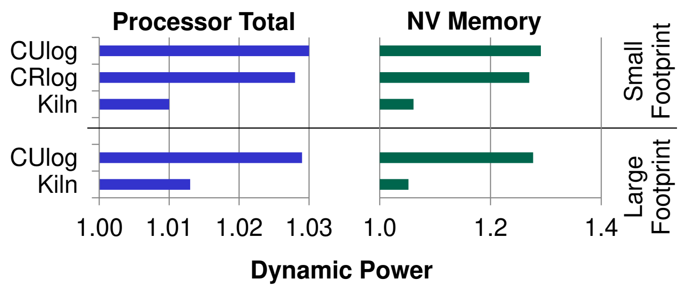


**Figure 5.12.** The average dynamic power consumption of processor (including the NV cache) and the NV memory, normalized to the Native (workloads running 16 threads).

four-byte integers. The value size ranges from 512KB to 64MB. Each persistent memory transaction inserts or deletes one entry of the hash table. When running a single thread, we do not observe any NV cache overflows even with the value size increased to 32MB. With multithreaded workloads, the frequency of NV cache overflow is lower than 0.1% (100 overflow events out of the total 100k transactions) when the total memory footprint of all the threads is smaller than 64MB. Unfortunately, the frequency reaches 100% when the total memory footprint of all the concurrent transactions is larger than the NV cache capacity. In such a case, Kiln falls back to hardware-controlled COW as described in Section 5.2.4, and the performance is similar to that of CRlog. We will leave the investigation of more efficient methods to resolve the overflow issue as future work.

### 5.4.4 Dynamic Power

Maintaining data persistence with Kiln incurs additional processor and memory dynamic power consumption due to the extra bookkeeping activities in cache controllers and the increased accesses to caches and the NV memory. We calculated the processor's dynamic power consumption by feeding the simulation statistics of processor and cache activities into McPAT [140]. We calculated the NV memory power consumption based on the number of memory accesses broken down into row buffer hits and misses, the memory energy configuration listed in Table 6.4, and the total execution time of each benchmark. As shown in Figure 5.12, Kiln provides up to a 23% dynamic power reduction for the NV memory compared to CXlog due to fewer memory accesses (Figure 5.10). Compared to the Native, Kiln results in dynamic power overheads of only 1.2% and 5% to the processor and the NV memory.

## 5.5 Summary

NVRAM technologies can provide promising solutions to persistent memory design. However, current NVRAM-based persistent memory designs are inefficient due to increased latency and bandwidth demands due to log-based or COW mechanisms. In this dissertation, we propose Kiln, a persistent memory design which employs a multiversioned memory hierarchy consisting of an NV cache and NV memory, enabling in-place updates to in-memory data structures, without the redundant writes required by logging or COW. Kiln provides persistence support with only a 9% performance overhead to the native system, hence up to 2× performance improvement to the log-based NVRAM persistent memory. In addition, Kiln provides a simple and intuitive software interface, as well as easy and fast recovery from failures. Our work rethinks the design of persistent memory in light of emerging NVRAM technologies, which is a critical step in reaping the full advantages of NVRAM technologies beyond simply replacing of DRAM in main memory.

Chapter 6

# FIRM: Fair and High-Performance Memory Scheduling for Persistent Memory Systems

Byte-addressable nonvolatile memory technologies promise a new type of "persistent applications" that access user-defined in-memory persistent data objects by loads and stores without paging from disks or flash. With such significant benefits, these applications also bring new challenges to the design of systems that run both persistent and non-persistent applications. One important issue is the competition for shared resources, e.g., shared caches, memory interface, and memory capacity. For example, a recent study [141] explored such resource competition at shared caches. In this chapter, we identify the resource competition at the memory interface raised by shared memory interface between these two types of applications and tackle this problem by redesign memory scheduling mechanisms implemented at the memory controllers.

Although our Kiln design does not require performing logging to ensure data persistence, logging has irreplaceable advantages in managing persistent memory. First, logs are portable; we can take the log from a failed system and directly install it in a completely different system. Second, we can maintain more than two versions in a log. Third, logging is currently the most efficient approach for large-granularity updates. Kiln also falls back to logging or COW with large-granularity updates which cause frequent NV cache overflows. Consequently, we expect Kiln and logging (or COW) mechanisms to coexist in persistent memory designs, with Kiln to be used to accommodate small-granularity persistent updates and logging (or COW) used to manage large-granularity persistent updates.

## 6.1 The Problem

NVRAMs can be used as persistent memory or a replacement of the DRAM technology in general-purpose systems [4, 142, 45, 35, 3, 143, 144, 145, 146, 147, 148, 149]. In these studies, applications that traditionally use DRAM to hold the working set can leverage NVRAM as an extra physical memory space, which can save the static and refresh energy consumed by a large size of DRAM. These prior work focuses on improving the performance or energy-efficiency of such a system, without differentiating between persistent and nonpersistent applications. Therefore, strikingly little attention has been paid to the study of cases when persistent and nonpersistent applications concurrently run in a computing system.

In such a system, these two types of applications can compete for various system resources, such as processing units, caches, memory interface, and memory capacity. In this work, we are interested in tackling the challenges incurred by sharing the memory interface. We find that such a sharing imposes significant resource contention at the memory interface due to the unique memory access behaviors of persistent applications.

Previous memory scheduling schemes, which are designed for nonpersistent applications, become inefficient under this new scenario. First, persistent-memory applications enforce in-order updates to persistent data structures in NVRAM. The beginning of a subsequent memory update depends on the completion of previous updates. As a result, writes are on the critical execution path of persistent-memory applications, a reverse case in most nonpersistent applications with reads on the critical execution path [150]. Conventional memory scheduling policies that prioritize reads over writes will unfairly slow down persistent applications. Second, persistent applications periodically generate a burst of writes to update the persistent data. These write requests may overflow the write queue and force the memory controller to drain it [150], aggressively servicing these write requests by stalling all memory reads. Without carefully balancing the memory bandwidth usage between reads and writes, such contention can happen periodically and eventually slow down nonpersistent applications. Third, persistent applications typically allocate a contiguous memory space (e.g., a log) to store persistent updates. Because the write queue size is quite limited, all the writes will go to the same bank during a write drain period. Without any bank-level parallelism, the memory bandwidth is underutilized. Liu et al. proposed a memory scheduling mechanism to exploit the bank-level parallelism present in the address stream of persistent applications [151]. However, their memory scheduler focused on scheduling write requests and failed to explore the contention between reads and writes in persistent applications.

Our goal is to design a memory scheduling scheme that achieves both fair memory accesses and high system throughput in a system concurrently running persistent and nonpersistent applications. We propose FIRM, a fair and high-performance memory scheduling mechanism that allows write requests from persistent applications to have the same priority as reads and balances the bandwidth usage between persistent and nonpersistent applications. Compared with the best case of traditional memory scheduling designs, FIRM can achieve system throughput improvement and fairness improvement by 8% and 29%, respectively.

## 6.2 Challenges of the Shared Memory Interface

This section investigates the challenges to the shared memory interface imposed by concurrently-running persistent and nonpersistent applications. To this end, we study the memory access behaviors of both applications with two microbenchmarks. From the results, we draw several key observations to discuss the reasons that previous memory scheduling methods fail to achieve fairness and efficiency for our scenario. Finally, we present a naive solution and elaborate its problems, motivating the more advanced mechanism in Section 6.4.

### 6.2.1 Persistent Applications

Most persistent applications are database and file system applications [69, 68, 152], which require critical data to survive sudden system failures, such as kernel crashes or power outage. Similar to traditional databases and file systems that maintain persistence in disks or flash, persistent applications perform logging [69, 70] or shadow updates [70] to make changes to persistent data, rather than directly overwriting the original data. In this manner, these applications can maintain multiple versions of data available for recovery once system failure happens. We refer to the memory stores to perform log and shadow updates as *persistent writes*. In a log-based persistent memory system, each log entry is a tuple consisting of the original data (or pointers referenced to the original data) and updated data values. Previous work implemented the log as a fixed size circular buffer [69]. Circular buffer implementations use contiguous memory space whenever the software initializer is able to do so. Otherwise, the initializer will allocate a linked list of fragmented chunks. In shadow update-based persistent memory, a shadow update copies the original data to a newly allocated memory space, modifies its values, and then redirects the pointer from the original data to the new data values. Each log or shadow update may generate arbitrary number of persistent writes depending on the granularity of a single data update. For example, in a key-value store, an update may be the addition of a new value at the granularity of several bytes, several kilobytes, or several megabytes.

To ensure the consistency of persistent data structure, persistent applications place rigorous ordering control over persistent writes with cache flush and memory fence instructions (e.g., `clflush` and `mfence`) or uncacheable writes to enforce that the order that log or shadow updates are written into NVRAM matches the order they are issued by CPU. Otherwise, the persistent data can lose consistency, for example, with a new data still remain in volatile caches (will be lost upon system crash) whereas a pointer referenced to it, being updated after the data in program, already exists in NVRAM. The cache flushes and uncacheable writes will periodically generate burst of writes at the memory interface. The memory fence will serialize memory accesses, making the subsequent memory accesses dependent on the completion of previous writes to NVRAM. Besides program phases with log and shadow updates, persistent applications also update the original data after a log or shadow update is completed (written into NVRAM). They may perform read-intensive activities such as searching a key in a key-value store, without modifying persistent data. During these program phases, persistent applications will appear to have similar memory access behavior as nonpersistent applications.

### 6.2.2 Memory Organization

Memory (either DRAM or NVRAM) is physically organized as two-dimensional arrays of bitcells. Each read or write access to memory requires an entire row in an array to be buffered in a row buffer (a set of sense amplifiers that act as latches of data values). The typical size of a row buffer can be up to 8K bytes [150]. Contiguous accesses to the same row will result in row-buffer hits in an “open” row, consuming much lower latency than accesses to a different row. Accessing a different row results in a row-buffer miss, which requires memory to close the open row by writing its data values back and activate the row to be accessed.

Memory arrays are further grouped into banks, ranks, and channels. The row buffer of each bank can be accessed independently, i.e., accesses to different rows of different banks can be performed in parallel. A memory channel consists of a number of banks, e.g., DDR3 memories typically adopt eight banks per channel. All the banks in a channel share a single memory bus, which only services single direction of memory accesses (reads or writes) at a time. Switching from one direction to the other incurs a bus turnaround latency, which is referred to as write-to-read delay (tWTR) of approximately 7.5ns and read-to-write delay(tRTW) of up to 15ns [153].

### 6.2.3 Conventional Memory Scheduling Mechanisms

Memory accesses issued from processors are buffered and scheduled by memory controllers. A memory controller employs a memory request queue, physically or logically separated as a read queue and a write queue, to store the memory requests waiting to be scheduled for service. A memory controller also utilizes a memory scheduler to decide which memory requests in the queue can be placed on the memory bus. A large body of previous work studied the decision making with various memory scheduling policies [154, 155, 156, 157, 158, 159, 160, 161, 162]. Most commodity systems employ first-ready first-come-first-serve (FR-FCFS) scheduling policy [154, 155], which prioritizes memory requests that lead to row-buffer hits. But it can unfairly deprioritize memory requests with low row-buffer hit rates, starve these requests and the corresponding threads for long time periods, and hurt overall system throughput [156, 157]. Several studies [158, 159, 160, 157, 161] endeavor to balance between system performance and fairness. Parallel-aware Batching Scheduling (PAR-BS) [161] improved both fairness and system throughput by batching requests based on their arrival times and prioritize the oldest batch over others. The proposed memory scheduling algorithm ranks batches from different applications to allow them to access different banks in parallel. ATLAS [158] improves system throughput by prioritizing applications that have received the least memory service. However, as demonstrated by Cluster Memory Scheduling (TCM) [159], doing so may unfairly deprioritize memory-intensive applications and hence slow down such applications. To address this issue, TCM [159] dynamically cluster applications into low and high memory-intensity clusters. It improves system throughput by prioritizing applications with low memory intensity; improves fairness by periodically shuffling priorities among applications with high memory intensity. Most previous memory scheduling schemes are designed for CPU applications. Staged Memory Scheduling (SMS) [162] addressed the scheduling challenges of CPU/GPU heterogeneous systems, where CPU applications may be unfairly slowed down by GPU applications that concurrently access multiple banks with high row-buffer hit rates. Unfortunately, none of these scheduling schemes are specifically designed for persistent memory systems. In the following, we show that running a combination of persistent and nonpersistent applications can pose various challenges to the design of memory scheduling schemes.

### 6.2.4 Memory Access Behaviors of Persistent and Nonpersistent Applications

To illustrate the different memory access behaviors of persistent and nonpersistent applications, we studied the memory accesses of three representative applications, *streaming*, *random*, and *btreelog*, in terms of four properties: memory intensity, write intensity, bank-level parallelism, and row-buffer locality. *Streaming* and *random* are both memory-intensive applications, performing consecutive and random accesses to a large array. *Btreelog* performs inserts and deletes of key-value pairs (25-byte keys and 2K-byte values) to a B+ tree data structure stored in main memory. We build this benchmark by implementing a redo logging (i.e., writing new data updates to the log instead of the original data addresses) interface on

| | Memory Access Behavior | | | |
|---|---|---|---|---|
| | MPKI | WR% | BLP | RBL |
| Streaming | High<br>100 | Low<br>47% | Low<br>0.05 | High<br>96% |
| Random | High<br>100 | Low<br>46% | High<br>6.3 | Low<br>0.4% |
| Btreelog | High<br>100 | High<br>77% | Low<br>0.05 | High<br>71% |
| Redo Logging | Very High<br>675 | High<br>92% | Low<br>0.01 | High<br>97% |

**Table 6.1.** Memory intensity, write intensity, bank-level parallelism, and row-buffer locality of different applications running individually. *Streaming* and *random* are two nonpersistent applications with streaming and random memory accesses. *Btreelog* performs inserts and deletes to a B+ tree-based key-value store with 25-byte keys and 2K-byte values and employs redo logging to ensure data persistence. In the last row of the table, we show the memory access behavior of *btreelog* when it is performing redo logging.

top of STX C++ B+ tree [137] to provide persistence support. Memory intensity is evaluated by the number of last-level cache misses per thousand instructions (MPKI) [159]. Write intensity is evaluated as the portion of write misses (WR%). Bank-level parallelism (BLP) is evaluated by the average number of banks to which there are outstanding memory requests the application has at least one outstanding requests [161] in our memory configuration. Row-buffer locality (RBL) is evaluated as the average hit rate of the row buffer across all banks [154]. The three applications were specifically constructed and intensity. We also construct *streaming* and *random* in the manner of dramatically different bank-level parallelism and row-buffer locality. Therefore, they are two extreme cases of memory-intensive applications with opposite memory access behaviors in bank-level parallelism and row-buffer locality.

Table 6.1 lists the memory access behaviors of these applications running separately. Compared with *streaming* and *random*, *btreelog* has much higher write intensity. This is because each insert or delete operation triggers a redo logging operation that appends a log entry containing the addresses and the data of the modified key-value pair, generating extra write traffic in addition to the original data updates. As shown in the last row of Table 6.1, the redo logging operation of *btreelog* results in significantly increased memory intensity. Writes make up almost all the memory traffic. *Btreelog*, especially when it is performing redo logging, has low bank-level parallelism and high row-buffer locality, making its write behavior similar to *streaming*'s reads but significantly different from *random*. *Btreelog* consists of key-value pairs at the granularity of more than 2K bytes and so do the log entries. As a result, each log update makes consecutive writes to the same bank. The first write request may be a row-buffer miss, but all the rest requests will hit in the row-buffer.

## 6.3 Key Observations

In the following, we analyze the above characterization results in detail, and summary our key observations obtained from these results.

**Persistent writes are also on the critical path:** Persistent applications enforce the ordering of persistent data updates by employing cache flushes and memory fence write after issuing each log or shadow update or making the persistent writes uncacheable. Consequently, during the period of updating a log entry or shadow data copy, any subsequent memory reads and writes need to stall until all the persistent writes are written into NVRAM. All the computation operations that depend on these memory accesses also need to wait. As a result of such ordering control, persistent writes are on the critical path of persistent applications execution. This is in contrast to the case with nonpersistent applications which have reads on the critical path [150]. Conventional memory scheduling policies that prioritize reads over writes can stall persistent applications while letting nonpersistent applications make forward progress, unfairly slowing down persistent applications.

**Bursts of persistent writes can overflow write queue:** A log or shadow update can generate a large number of write requests, for example, when performing inserts and deletes key-value pairs at the granularity of several kilobytes [69, 152]. Legacy database and file system program codes may update persistent data at an even larger granularity of megabytes or gigabytes. With a limited size, the write queue cannot accommodate such large number of write requests consecutively generated in a short period of time. Therefore, the memory scheduler may need to frequently drain the write queue, when persistent and nonpersistent applications are running together. As a result, nonpersistent applications can be significantly slowed down with stalled reads.

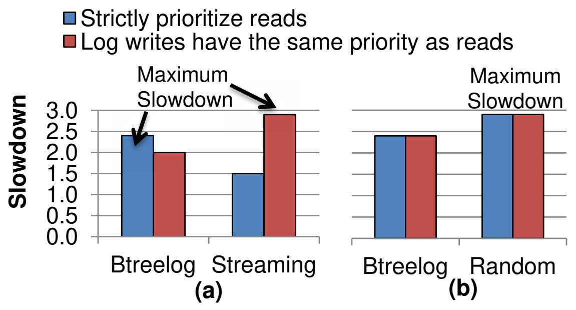


**Figure 6.1.** Effect of simply assigning persistent writes the same priority as reads on different workload combinations. (a) *WL1* consists of *Btreelog* and *streaming*. (b) *WL2* consists of *Btreelog* and *random*.

**Conventional memory scheduling schemes** [158, 159, 160, 157, 161, 163] make two assumptions to concurrently running applications. First, they assume that reads are on the critical path of application execution. This is sound when most nonpersistent applications abound with read-dependent arithmetic, logic, and control flow (branch) operations. Therefore, most previous memory scheduling schemes prioritize reads over writes. Second, they assume that applications are read intensive so they can delay the writes without frequently filling the write queue. Most previous memory scheduling schemes buffer writes to let reads aggressively utilize the memory bus. The memory scheduler may eventually need to drain the write queue, when the write queue is full (or filled to a predefined "high watermark") to process a batch of waiting writes to prevent stalling the entire processor pipeline. During write drain, the memory bus can only service writes, refuting any read requests. Therefore, frequent write drain can significantly slow down reads and harm the performance of read-intensive applications. Unfortunately, these assumptions do not hold when persistent applications are sharing the memory interface. Conventional memory scheduling schemes fail to preserve fair and high-throughput memory accesses when persistent and nonpersistent memory applications are running together.

### 6.3.1 A Naive Solution

A natural solution to resolve the issue is to assign persistent writes with the same priority as reads. However, in practice, we find that this method fails to achieve fairness of memory accesses on various workload combinations.

In Figure 6.1, we studied the fairness of two different memory scheduling schemes, TCM [159] and a modified TCM which assigns persistent writes the same priority as reads, by evaluating individual application's slowdown and the maximum slowdown between the two applications in different workload combinations. In a system with *streaming* and *btreelog* workloads (*WL1* in Figure 6.1(a)), *btreelog* has the maximum slow down with original TCM [159]. With modified TCM, the *btreelog* workload benefits from assigning persistent writes the same priority as reads, while *streaming* performs much worse. Because both *btreelog* and *streaming* have high row-buffer locality, a memory scheduler with modified TCM tends to evenly partition the memory bandwidth between persistent writes and *streaming*'s reads, e.g., by servicing a batch of persistent writes followed by a batch of *streaming*'s reads, and vice versa. As a result, the memory bus has to be explicitly switched between persistent writes and *streaming*'s reads. This bus turnaround delay is at the scale of 7.5 ns [153]. Frequent bus turnarounds add considerable delays to total memory access latency, which eventually slow down both reads and persistent writes. *Streaming* is more vulnerable to the frequent bus turnarounds because it keeps performing intensive reads all the time, and most of the reads are slowed down by such bus-turnaround overhead. The naive solution may improve *btreelog* performance by reducing the stall time of persistent writes, however, significantly slows down *streaming* and increases the maximum slowdown in the workload combination.

For *WL2* (Figure 6.1(b)) with applications of very different bank-level parallelism and row-buffer locality, *random* always has the maximum slow down. The memory accesses of *random*, with bank-level parallelism, can easily interference with persistent writes of *btreelog*. As demonstrated by TCM [159], the interference can reduce *random*'s bank-level parallelism and hurt its performance. Consequently, even though persistent writes have the same priority as reads, the modified TCM still prioritizes *random* over *btreelog*'s persistent writes. As a result, the naive solution does not affect the slowdown of the two applications at all, leading the two sets of bars in Figure 5.1(b) to be the same. Unfortunately, *btreelog*'s intensive persistent writes can force the memory scheduler to frequently drain the write queue. The effect is the same as forcing the memory scheduler to prioritize per-

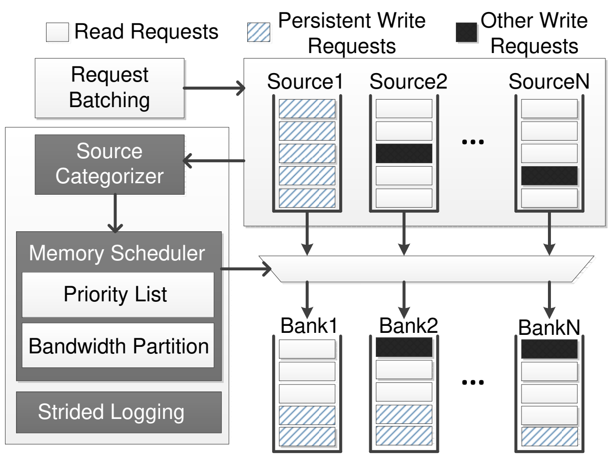


**Figure 6.2.** Overview of FIRM design. Note that we show a logical view of source request queues. The physical locations of memory requests are read and write queues in memory controllers.

sistent writes over *random*'s reads, eventually slowing down *random*. As a result, *random* will remain to be maximally slowed down with the naive solution. Commodity memory scheduling schemes, e.g., FR-FCFS, may slow down *random* even more by prioritizing *btreelog*'s persistent writes with high row-buffer locality.

In summary, the naive solution fails because conventional memory scheduling designs focus on resolving contention between read requests, i.e., single-direction bus transfers. New solutions need to be designed to manipulate the sharing of memory interface between equivalently prioritized, dual-direction data transfers.

## 6.4 Mechanism

Our goal is to design a memory scheduling mechanism that achieves fairness and high throughput in a system running both persistent and nonpersistent applications. Figure 6.2 depicts an overview of proposed FIRM design. Components of FIRM include source categorization, memory scheduling policy, and strided logging mechanism. We batch the memory requests of different applications based on their row-buffer locality, i.e., row-hit requests from the same application will be batched together [161]. The source categorizer monitors memory access behaviors of running applications and dynamically classify them into persistent and various nonpersistent sources. The memory scheduler decides which request batch can be serviced. Finally, we employ a strided logging mechanism to accelerate log updates by augmenting their bank-level parallelism. This section presents the basic idea of FIRM mechanism. The detailed implementation methods, including software interface and architecture extensions, will be described in Section 6.6.

### 6.4.1 Categorizing Sources of Memory Requests

FIRM dynamically classifies the sources of memory requests in four categories: *nonintensive*, *persistent*, *streaming*, and *random*, based on their memory intensity, write intensity, bank-level parallelism, and row-buffer locality. A **source** is referred to as a process or a thread in a particular time period, when it is generating memory requests in a specific behavior. For example, a persistent application is classified as a *persistent source* when it is performing persistent updates (logging or shadow updates). But it may also be a nonintensive, a streaming, or a random source in other time periods. A *nonintensive source* has low memory intensity. Other sources generate memory-intensive requests. *Streaming* and *random* sources are typically read intensive. A *streaming source* generates memory accesses with low bank-level parallelism and high row-buffer locality. A *random source*, on the contrary, generates memory accesses with high bank-level parallelism and low row-buffer locality.

FIRM adopts program hints (with the software interface described in Section 6.6) to decide whether an application is a persistent application. This prevents FIRM to classify a nonpersistent applications with a temporary write-intensive program phase as a persistent source. FIRM identifies that a persistent application becomes a persistent source, when it has high memory intensity and high write intensity. We find that these two characteristics are sufficient to distinguish persistent updates from other program phases in a persistent application. Algorithm 2 shows the pseudo-code for the source categorization algorithm used by FIRM. We perform the categorization at the beginning of each time interval of one million cycles, which, based on our experiments, well trades off between accuracy and performance overhead. We use TCM's thread clustering mechanism [159] to identify nonintensive sources. We use the parameters $thresh_{wr}$ and $thresh_{blp}$ as the thresholds to identify high write intensity and high bank-level parallelism. We experimented with various threshold values and find that FIRM can well categorize persistent and random sources using 80% and 4 as the thresholds, respectively.

**ALGORITHM 2:** Source categorization algorithm.

```
Initialization:
$PersistentApps \leftarrow TID$
Categorization: (beginning of time interval)
$PersistentSource \leftarrow \varnothing$; $StreamingSource \leftarrow \varnothing$;
$RandomSource \leftarrow \varnothing$; $NonintensiveSource \leftarrow \varnothing$;
$Uncategorized \leftarrow \{Thread_i : 1 \leq i \leq N_{threads}\}$
for $Thread_i \in Uncategorized$ do
    $NonintensiveSource = TCM(Thread_i)$
    if $Thread_i \in$ PersistentApps and $WR\%_i \geq Thresh_{wr}$ then
        $PersistentSource \leftarrow PersistentSource \cup Thread_i$
    else if $BLP_i \geq Thresh_{blp}$ then
        $RandomSource \leftarrow RandomSource \cup Thread_i$
    else
        $StreamingSource \leftarrow StreamingSource \cup Thread_i$
    end if
end for
```

## 6.5 Memory Scheduling

**Priorities:** Table 6.2 shows the priority of memory requests from different sources. We allocate the highest priority to memory accesses from nonintensive sources. Because they access memory infrequently and consume only a small fraction of the total memory bandwidth, servicing them provides greater potential for the corresponding applications to make forward progress. We allocate the same priority to persistent writes and streaming reads, because they are both on the critical execution path and have similar memory access behavior except for the access direction. We prioritize random reads over persistent writes and streaming reads, because they are vulnerable to the memory interference with persistent writes and streaming reads [159].

**Partitioning Memory Bandwidth Between Persistent Writes and Streaming Reads:** As discussed in Section 6.2, simply assigning persistent writes the same priority of streaming reads can slow down both sources, by frequently

| Priority | Memory Requests |
|---|---|
| 1 | Non-intensive accesses |
| 2 | Random reads |
| 3 | (a) Persistent writes and (b) Streaming reads |

**Table 6.2.** Priority strategy when at least one active persistent source is present in the system.

turning around the driving direction of the memory bus. We address this issue by partitioning the memory bandwidth between the two types of sources based on their bandwidth demands and the constraint of bus turnaround overhead demanded by users. FIRM partitions memory bandwidth between the two types of sources with three steps: calculating memory bandwidth demand, merging the memory request batches of each source to be larger batches, and scheduling the merged batches. We calculate a source's bandwidth demand ($D_i$) as the number of its queued up memory requests, i.e., $D_i \in \{D_1, D_2, ..., D_n\}$. These memory requests may be batched by a previous batching scheme, such as PAR-BS [161], to exploit their row-buffer locality and bank-level parallelism. The grouping step merges the batches of a source based on their bandwidth demand, and reduces the bus turnaround overhead to below the threshold defined by users. We calculate the size of a memory request group ($G_i$) with the following equation:

$$G_i = k \left\lfloor \frac{D_i}{min\{D_1, D_2, ..., D_n\}} \right\rfloor$$

where $k$ is used to tune the size of the minimum-sized group to reduce the frequency of read-write switching. It is calculated based on the memory parameters of write to read delay ($t_{WTR}$) and row-buffer hit latency ($t_{hit}$):

$$k = \left\lceil \frac{t_{WTR}}{t_{hit}\mu_{turnaround}} \right\rceil$$

where $\mu_{turnaround}$ is a user-defined parameter that represents the percentage of bus turnaround time out of the total service time of memory requests. Users can define $\mu_{turnaround}$ by configuring the system BIOS. By merging the batches, we can effectively reduce the bus turn-around overhead. For example, with the memory configuration described in Table 6.4, the overhead of bus turnaround is up to 21%

(7.5ns tWTR over 36ns row-buffer hit latency) when the memory bus services single read and write requests in an interleaving manner. By scheduling the requests with groups of minimum 32 requests (that hit in a 2K-byte row-buffer), we can reduce the overhead to 0.6%. Finally in the memory scheduling step, we schedule the groups of requests in a round-robin manner.

### 6.5.1 Strided Logging

Memory bus can only service one direction of data transfers at a time. When memory bus is servicing persistent writes, reads from other applications must stall. But log updates, with low bank-level parallelism and high row-buffer locality, tend to make consecutive accesses to a single bank in a given period of time. As described in Section 2, most persistent applications create logs by allocating one or chunks of contiguous memory space to store a circular buffer [69]. Commodity memory controllers employ randomized higher-order address bits as bank index to avoid bank conflicts of random and strided accesses (Figure 6.3(a)), however, may fail to map log updates to different banks. As shown in Figure 6.3(b), only lower-order address bits change among a sequence of persistent updates; the higher-order bits are fixed. As a result, a sequence of log updates are mapped to the same bank (Figure 6.3(b)), and thus are significantly slowed down by bank conflicts and refuse any read requests from other applications. The memory bandwidth is significantly under-utilized. It is impractical to exploit bank-level parallelism of log updates by buffering their write requests due to the large buffering capacity required. For example, when the write requests to a second bank are issued, we need to buffer at least 32K bytes (512 entries) of writes write requests. To fully utilize all the eight banks of a DDR3 channel, we need a write queue as large as 128K bytes.

To accelerate log updates by fully utilizing memory bandwidth, we propose a strided logging mechanism to improve their bank-level parallelism. Figure 6.3 illustrates the strided updates to a log in a circular buffer. The persistent application can still allocate a contiguous memory space for the circular buffer. But we modify the memory scheduling such that accesses to the circular buffer are in a strided manner and are mapped to different banks. Continuous log updates of less than a row-buffer size can still access a group of contiguous buffer entries to explore high

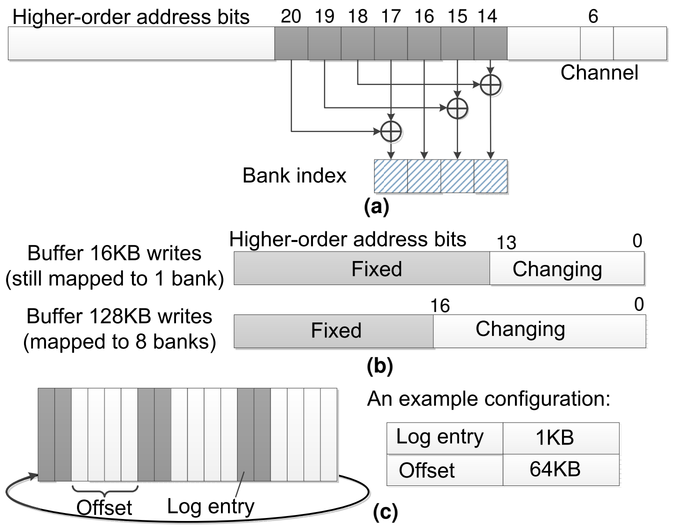

Higher-order address bits
20 19 18 17 16 15 14
6
Channel
Bank index
(a)
Buffer 16KB writes (still mapped to 1 bank)
Higher-order address bits
13
0
Fixed
Changing
Buffer 128KB writes (mapped to 8 banks)
16
0
Fixed
Changing
(b)
An example configuration:
Log entry
1KB
Offset
64KB
Offset
Log entry
(c)

**Figure 6.3.** Conventional address mapping scheme, address bits of persistent writes, and proposed bank shuffling.

row-buffer locality. But log updates beyond the size of the row-buffer will stride by an offset. The value of the offset is determined by the position of bank index bits used in the address mapping scheme. For example, with the address mapping scheme shown in Figure 6.3(a), the offset will be 128K bytes if we want to fully utilize the eight banks with log updates. Instead of modifying the memory controllers, our strided logging mechanism can be implemented in a user-mode library as well. We will describe detailed implementation in Section 6.6.

### 6.5.2 Summary of FIRM Mechanism

In summary, FIRM is a persistence-aware memory scheduling design that can well address the challenges posed by running persistent and nonpersistent applications concurrently. First of all, to accommodate the persistent writes being on the critical execution path, FIRM prioritizes them as if they were read requests. This in term prevents the persistent writes from filling up the write queue and reduces the chance of write drain. Furthermore, FIRM addresses the challenges incurred by the different directions of data transfers between reads and persistent writes – the overhead of bus turnaround time and the low bandwidth utilization when servicing persistent writes. FIRM reduces this overhead by merging those short memory request batches. It increases persistent writes' bandwidth utilization by performing strided logging.

## 6.6 Implementation

This section presents the implementation details. First, we provide a software interface and instruction set architecture (ISA) extensions for users to define a program as a persistent application. Second, we employ a set of hardware counters in memory controllers to monitor running workloads' memory access behavior. Third, we present our hardware and software modifications to implement the strided logging mechanism.

### 6.6.1 Software Interface and ISA Extension

FIRM adopts program hints to determine whether a persistent application is running in a computing system. We expose to programmers the following software interface as a declaration of performing persistent memory updates in the programs:

```
#pragma persistent_memory
```

Instead of defining the entire program as a persistent-memory application, programmers can also annotate particular threads in a program as persistent threads performing log or shadow updates. In this case, users can declare a `persistent` attribute when they create such a thread. In addition, we extend the ISA with a pair of new instructions, `PM_BEGIN` and `PM_END`, to indicate that a piece of code belongs to a persistent application or thread. The software interface can be translated to the ISA instruction with simple modifications to compilers. For example, the compiler will translate the `#pragma persistent_memory` by adding `PM_BEGIN` and `PM_END` at the beginning and the end of the application code.

### 6.6.2 Hardware Counters and Registers

**Persistence indicators:** Once a processor reads users' input of `PM_BEGIN`, it signals each memory controller by writing to a set of "persistence indicators" to

| Counter Name | Storage |
|---|---|
| Registers | $1 + log_2 N_{threads} = 5bits$ |
| Write counter | $N_{cores} \times 10 = 80bits$ |

**Table 6.3.** Storage required by hardware counters in each memory controller. The values are calculated based on the baseline configuration described in Section 6.7.

a global register to indicate the presence of a persistent application or thread. The persistence indicator contains two regions, a single-bit *persistence* and a $log_2 N_{threads}$-bit *thread identifier*. With a persistent-memory thread, the processor sets the persistence bit to “1” and writes the corresponding thread ID to the other region. With a persistent-memory application, the processor will set the thread identifier to “0”.

**Hardware counters:** Compared with previous work [161, 159], our design adds a set of 10-bit write counters in each memory controller, each correlated to a hardware thread of the processor. The memory controllers use these write counters to monitor the write intensity of each source. The write counter records the number of write misses generated by the last-level cache in each time interval. At the end of each time interval, a memory controller calculates the write intensity of each source using the corresponding write counter and MPKI counter (introduced by TCM [159]). We reset the counters at the beginning of each time interval, after the memory scheduler has made their scheduling decisions. To monitor memory intensity, BLP, and RBL of memory requests, we adopt a set of hardware counters similar to those used in TCM [159] (including a MPKI counter, load and BLP counters, and shadow row-buffer index and hit counters in each memory controller). Table 6.3 lists the storage required by these hardware counters in each memory controller, based on the configuration with an eight-core system described in Section 6.7.

**Supporting multiple memory controllers:** Because we maintain per-thread hardware counters in each memory controller, the memory controllers can independently make scheduling decisions about its local memory requests based on the hardware counter information it collects. Consequently, our design does not require a centralized arbiter to coordinate all the memory controllers.

### 6.6.3 Implementing Strided Logging

The strided logging mechanism can be implemented in memory controller hardware or by modifying the logging functions in a user-mode library (e.g., employed by Mnemosyne [69]).

To implement the mechanism in user-mode library, we implement the two counters in the circular buffer implementation, and modify the `log_append()` function to stride with the offset with each log append request defined by programmer. By supporting the strided log appends in memory controller, no software modification is needed. However, the memory controller needs to maintain the two counters. In addition, the memory controller needs to maintain a register to locate the starting and end addresses of the circular buffer.

To implement the strided logging in a memory controller, we employ a hardware counter and two registers. The two registers records the starting and the end addresses of a circular buffer based on the hint obtained from `log_create()` function, so that the memory scheduler can identify the boundary of the buffer. We call the hardware counter as *group index*. It records the number of appended log entries within a group. It is used to indicate when the group is fully occupied. In this case, memory controller will map the coming write requests to the next group in the circular buffer by striding with an offset. When a log append loops back to the original group, it will start to write in the neighboring group by further striding by the size of a group.

## 6.7 Experimental Setup

We evaluated FIRM design with a set of multithreaded and single-threaded workloads running on a multi-core system. This section describes our simulation framework, processor and memory configurations, and benchmarks.

### 6.7.1 Simulation Framework

Our experiments are conducted using McSimA+ [132], a Pin-based multi- and many-core cycle-accurate simulation infrastructure [133]. McSimA+ models out-of-order cores, caches, directories, on-chip networks, and memory channels. Ta-

**Table 6.4.** Parameters of the evaluated multi-core system.

| Processor/Fab. Proc. | Intel Core i7 / 22 nm |
|---|---|
| Cores | 2.5GHz, 2 threads per core |
| L1 Cache (Private) | 64KB, 4-way, 64B lines, 1.6ns latency |
| L2 Cache (Private) | 256KB, 8-way, 64B lines, 4.4ns latency |
| L3 Cache (Shared) | 2MB/core, 16-way, 64B lines, 10ns latency |
| Memory Controller | Two dual-channel memory controllers |
| | 128-/128-entry read/write buffer |
| DRAM DIMM | DDR3-1600, 512MB |
| NV Memory DIMM | STT-MRAM, 2GB, 8 banks, |
| | 2KB row buffer, 36ns row-buffer hit, |
| | 65/76ns read/write row-buffer conflict |

ble 6.4 lists the detailed parameters and architecture configurations of the processor and memory system to be evaluated with the proposed FIRM design. Each processor core is similar to one of the Intel Core i7 cores [134]. The processor incorporates SRAM-based volatile private and shared caches. The L3 cache is 16-way set-associative and multi-banked. The cores and L3 cache banks communicate with each other through a crossbar interconnect. A two-level hierarchical directory-based MESI protocol is employed to maintain cache coherence at the private caches and the L3 cache. The DRAM and the NVRAM are modeled as off-chip DIMMs compatible with DDR3[1]. They are mapped to a single physical address space. DRAM is used to store stacks and data transfer buffers, while the rest of data is mapped to the NVRAM address space. Memory requests to DRAM and the NVRAM are managed by two dual-channel memory controllers, respectively. The timing parameters of the NVRAM is calculated with NVSim [135], a performance, power, and area estimation tool for NVRAM. Our simulation framework models FIRM's memory controller design on source categorization, scheduling policy, and hardware-based strided logging.

### 6.7.2 Workloads

Table 6.5 lists the characterization results of our benchmarks running separately. We select four benchmarks, *mcf, lbm, leslie3d, povray*, from SPEC CPU 2006 benchmark suite [102] with different memory intensity, bank-level parallelism, and

[1]Everspin recently launched the DDR3 compatible STT-MRAM components [40], which transfers data at a speed comparable to current DDR3-1600.

**Table 6.5.** Benchmarks used in our experiments.

| **Benchmarks** | **MPKI** | **WR%** | **BLP** | **RBL** |
|---|---|---|---|---|
| Dbacl [164] | 28.2 | 38% | 4.21 | 12% |
| FFmpeg [165] | 10.5 | 33% | 1.43 | 30% |
| Masstree [166] (Logging) | 25.6 (528.8) | 27% (86%) | 1.52 (0.2) | 53% (90%) |
| mcf | 73.4 | 25.6% | 6.0 | 41.1% |
| lbm | 28.2 | 42.0% | 2.8 | 78.7% |
| leslie3d | 15.7 | 4.0% | 1.7 | 90.8% |
| povray | 0.1 | 6.0% | 1.2 | 77.6% |

**Table 6.6.** Workloads mixed with Masstree and various nonpersistent applications.

| | | | |
|---|---|---|---|
| **W1** | lbm, leslie3d, povray | **W2** | mcf (2), povray |
| **W3** | mcf, lbm, povray | **W4** | Dbacl (2), FFmpeg |

row-buffer locality. We also use the following three real-world applications in our workloads.

*Dbacl* is an offline text document (e.g. email) classifier based on Bayesian statistical principles. This tool has two modes, one to learn features from given sample text documents and store the results in a file, the other to classify documents with the given categories learned from the first mode. We use the CMU text learning group dataset [167] as, which contains emails from 20 newsgroups. We randomly choose 100 emails as the input for learning. The total size of the learning results is 6.5 MB; 321 KB per category. The application then classifies new emails from any of the 20 newsgroups by extracting features from new emails, loading training data, and comparing the input features with the training data set. *FFmpeg* [165] is a cross-platform solution to record, convert, and stream audio and video. We use it with x264 encoding library to convert a 60M-byte video file in MPEG-1 format to MPEG-4 format. *Masstree* [166] is a persistent in-memory key-value store. It adopts B+ tree as basic data structure and uses layers of B+ trees to form a tier. It uses write-ahead logging for persistence. In our experiments, we configure it with 25-byte keys and 1.8K-byte values (so each log update is approximate 2K bytes) and perform inserts and deletes to the key-value store.

To evaluate FIRM's effectiveness on resolving the contention on the shared memory interface, we map all the memory accesses of these applications to the NVRAM address space, except for program stacks and data transfer buffers. Our four workloads (Table 6.6) are combinations of Masstree [166] with various non-persistent applications. Workloads *W1*, *W2*, and *W3* incorporates applications with different fractions of random and streaming memory access behaviors. We use *W4* to model a possible application in real-world personal devices. In this workload, *Masstree* and *FFmpeg* are multithreaded applications, with two threads in our evaluation, while *dbacl* is single-threaded. Therefore, we run two copies of this application on different cores. We use the workload combinations in Table 6.6 to evaluate our design on a four-core system. We also evaluate 8-core and 16-core systems by combining multiple copies of each workload.

### 6.7.3 Metrics

We evaluate system throughput using weighted speedup (WS) [161]:

$$WeightedSpeedup = \sum_i \frac{Throughput_i^{shared}}{Throughput_i^{alone}}$$

where $throughput_i$ is calculated as instruction throughput, i.e., number of executed instructions per cycle (IPC) with *dbacl* and *FFmpeg*. It is calculated as operation throughput, i.e., number of completed inserts and deletes per cycle with *Masstree*. We evaluate fairness using maximum slowdown [159]:

$$MaximumSlowdown = \max_i \frac{Throughput_i^{alone}}{Throughput_i^{shared}}$$

## 6.8 Results

This section presents the evaluation results and analyze the reasons that lead to these results.

### 6.8.1 Performance and Fairness of the Naive Mechanism

We first study the performance of a naive memory scheduling scheme: assigning persistent writes the same priority as reads in conventional memory scheduling mechanisms. We evaluate this naive scheme applied to commodity FR-FCFS [154, 155] memory scheduling and a number of recently proposed designs [159, 162, 161,

**Table 6.7.** Weighted speedup and maximum slowdown of various workloads: a naive memory scheduling mechanism compared with conventional memory scheduling schemes.

| **Schemes** | **Normalized Weighted Speedup** | | | | **Normalized Maximum Slowdown** | | | |
|---|---|---|---|---|---|---|---|---|
| **Workloads** | W1 | W2 | W3 | W4 | W1 | W2 | W3 | W4 |
| **PFR-FCFS** | -8% | -37% | -19% | -10% | -25% | -72% | -52% | -38% |
| **PPAR-BS** | -5% | -21% | -9% | -7% | -12% | -53% | -38% | -20% |
| **PTCM** | -3% | 5% | 3% | 0% | -8% | -3% | -6% | -9% |
| **PSMS** | 2% | 5% | 4% | 3% | -6% | -1% | -6% | -7% |

158] that optimize fairness and throughput of nonpersistent applications. We add a prefix "P" (meaning persistence-aware) to represent the naive scheme combined with each conventional memory scheduling design. We study the weighted speedup and maximum slowdown of various workloads by employing the naive scheme and compare the results with each conventional memory scheduling design.

Table 6.7 shows the results evaluated on a four-core system. Because the naive scheme prioritizes persistent writes as if they were read requests, it improves the weighted speedup by up to 5% when incorporated with TCM and SMS mechanisms. Compared with conventional memory scheduling schemes, the naive scheme does not improve but degrades both fairness or system throughput with most workloads. In particular, *W2* incorporates two copies of memory-intensive *mcf*, which has high bank-level parallelism and low row-buffer locality. FR-FCFS prioritizes read requests that will hit in the row-buffer. With PFR-FCFS, persistent writes, i.e., *Masstree*'s log updates have the same priority as reads. Therefore, PFR-FCFS will prioritize *Masstree*'s log updates which have high row-buffer locality, and deprioritize *mcf*'s read requests. As a result, PFR-FCFS incurs a 72% fairness degradation in *W2*. In a workload mixed with applications performing consecutive memory accesses to a small number of banks (*W1*), PFR-FCFS tends to partition the memory bandwidth evenly among various applications. However, prioritizing persistent writes as if they are read requests can result in frequent switches of bus directions. As a result, both persistent and nonpersistent applications are slowed down. The original PAR-BS can improve the fairness among nonpersistent applications, by batching the memory requests of the same application and explore the bank-level parallelism between different nonpersistent applications. However, persistent writes and reads from nonpersistent applications

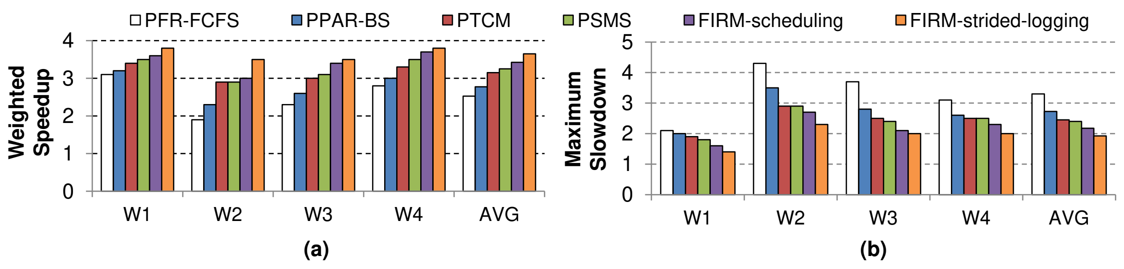


**Figure 6.4.** System throughput and fairness with various memory scheduling schemes. Among listed results, FIRM-scheduling employs the proposed source categorization and memory scheduling mechanisms but not strided logging. FIRM-strided-logging represents that all FIRM mechanisms, including hardware-based strided logging, are employed. (a) System throughput evaluated as weighted speedup. (b) System fairness evaluated as maximum slowdown.

cannot be serviced at the same time, and therefore PPAR-BS cannot explore the bank-level parallelism among them. Compared with the original PAR-BS mechanism, PPAR-BS can lead to significant fairness degradation(up to 53%). Both TCM and PTCM tend to prioritize the memory nonintensive application *povray* over memory-intensive applications in *W1*, *W2*, and *W3*. They also prioritize *mcf* with high bank-level parallelism over other memory-intensive applications in *W2* and *W3*. However, *Masstree* can become write intensive when it performs logging and fills the write buffer. Draining the write buffer can stall the read requests of *mcf*, and slow it down. PSMS can schedule a long sequence of persistent writes in small batches. However, servicing memory requests in small batches may result in frequent switches of bus direction, slowing down both persistent and nonpersistent applications. Overall, the naive memory scheduling scheme results in fairness degradation in all cases. It also degrades the weighted speedup compared with original FR-FCFS and PAR-BS mechanisms due to the significant unfairness it incurred.

### 6.8.2 FIRM Performance and Fairness

We evaluate the system throughput and fairness of our FIRM design by comparing it with the naive mechanisms that we have studied in Section 6.8.1. Figure 6.4 shows our evaluation results. To illustrate the performance and fairness benefits of different components of FIRM, we show individual results of FIRM-scheduling and FIRM-strided-logging. The bars of FIRM-scheduling illustrate the results when the proposed source categorization and memory scheduling policy are employed. In Figure 6.4, we set $\mu_{turnaround}$ to be 0.6%. As shown in the figure, FIRM-scheduling achieves better system throughput and fairness than the naive mechanisms. Compared with PSMS, the best of various naive mechanisms, it improves system throughput and fairness by 5% and 11%, respectively. The bars of FIRM-strided-logging illustrate the results of using hardware-based strided logging combined other two components of FIRM. We employ a 128-entry write buffer, which can store four log updates (8K bytes). Therefore, we configure the strided logging mechanism to map the four log updates to four different banks. Our results show that FIRM-strided-logging mechanism can further improve system throughput and fairness by 6% and 12%, respectively. Overall, compared with the best case of previous memory scheduling design (SMS), our design can improve system throughput and fairness by 8% and 29%, respectively.

We also evaluate the system throughput and fairness with software-based strided logging by modifying the logging functions. With it, we can eliminate the performance overhead of manipulating the counter, the *group index* described in Section 6.6. Accessing the counter only consumes a maximum of two cycles (to increment the counter), while a log update of 2K-byte can take up to 2K cycles (one row-buffer miss and 32 row-buffer hits) in our configuration (Table 6.4). Therefore, compared with hardware-based strided logging, software-based design reduces less than 1/1000 latency with each log update on average. As a result, we did not observe performance and fairness difference between using hardware- and software-based strided logging mechanisms.

### 6.8.3 Sensitivity to NVRAM Latency

The above evaluations employ STT-MRAM with the read and write latencies listed in Table 6.4. However, latency can vary among different NVRAM technologies [4]. For example, PCRAM can have a much longer write latency than read latency [4]. To evaluate FIRM's sensitivity to longer write latency, we study the throughput and fairness of our design with increased write latency. In Figure 6.5, we compare the naive and FIRM schemes' sensitivity to increased write latency, averaged

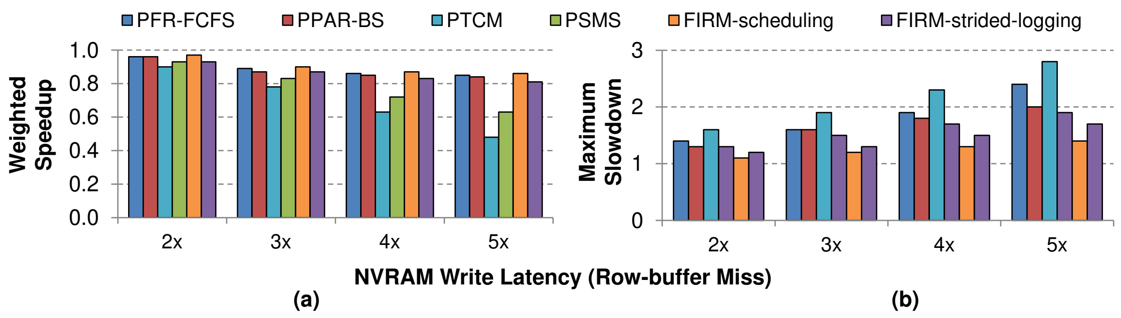


**Figure 6.5.** Results of average system throughput and fairness, when NVRAM write latency varies from 2× to 5× of the original write latency (Table 6.4). Weighted speedup (a) and maximum slowdown (b) are normalized to the case using the original write latency.

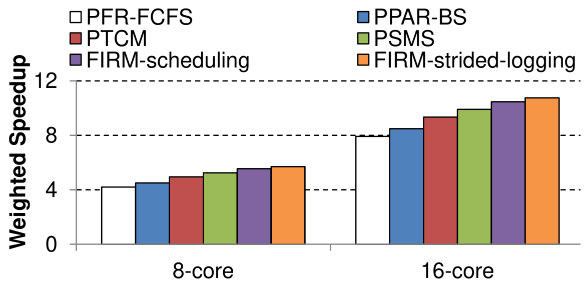


**Figure 6.6.** Average weighted speedup with various memory scheduling schemes on 8-core and 16-core systems.

across our evaluated workloads. As shown in the figure, our design retains high throughput and fair memory accesses with longer NVRAM write latency. FIRM-strided-logging increases the bank-level parallelism of persistent writes, and therefore also increases row-buffer misses. As a result, FIRM-strided-logging incurs 6% and 19% degradation on system throughput and fairness, respectively. Among various naive memory scheduling schemes, PTCM is sensitive to the increased write latency, because it tends to deprioritize persistent writes all the time so the write buffer can be easily filled by persistent write requests. Persistent writes that fill the write buffer will take longer time to complete during drain write periods with longer write latency.

### 6.8.4 Scalability with Cores

Figure 6.6 and Figure 6.7 illustrate the system throughput and fairness of our design with the same number of request queues but increased number of cores. We

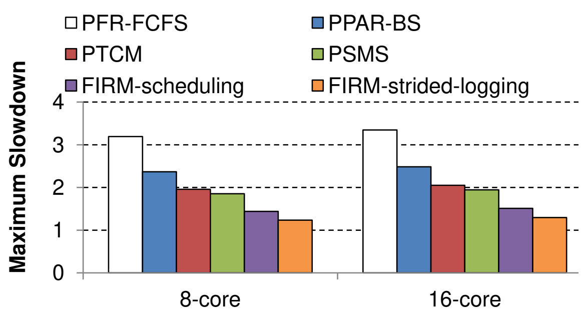


**Figure 6.7.** Average maximum slowdown with various memory scheduling schemes on a 8-core and 16-core systems.

combine two and four instances of each workload to saturate the system scaled up with 8 and 16 cores, respectively. The results shown in the figures are the average of all four workloads. Concurrently running multiple instances of workloads can cause more contention of memory bandwidth. Therefore, doubling the number of cores results in much less than twice of system throughput. As shown in Figure 6.6 and Figure 6.7, FIRM achieves the best system throughput and fairness among various memory scheduling mechanisms with both 8- and 16-core systems.

## 6.9 Summary

The emerging byte-addressable nonvolatile memory technologies open up opportunities for persistent applications that access user-defined in-memory persistent data objects by loads and stores without paging from disks or flash. On the other hand, both persistent and non-persistent applications can compete for shared resources, such as memory interface addressed in this dissertation. We proposed a persistence-aware memory scheduling scheme, FIRM, which achieves both fair memory accesses and high system throughput for the co-running applications. FIRM classifies persistent writes from other memory requests, coordinates their bandwidth usage with other concurrent memory requests, and increases overall system bandwidth utilization by augmenting bank-level parallelism of persistent writes. The experimental results show that FIRM can significantly improves both throughput and fairness compared with previous memory scheduling designs.

# Chapter 7

# Conclusion

The memory hierarchy is becoming a fundamental performance and energy bottleneck in computer systems, due to the performance and energy challenges posed by the requirements of modern applications and the performance and energy limitations in traditional memory technologies. Emerging NVRAM technologies yield abundant opportunities and challenges in memory hierarchy designs as demonstrated in this dissertation. NVRAMs are prime candidates for high-performance, energy-efficient memory, and novel hierarchy design. By leveraging NVRAM's performance and energy efficiency benefits, we can *replace* the traditional memory technologies employed by current memory hierarchy designs in CPU and GPU systems. Furthermore, we can *re-architect* the memory/storage stack by leveraging their nontraditional feature – incorporated with both memory and storage properties.

## 7.1 Summary of Contributions

This dissertation has discussed how to re-architect the memory hierarchy to achieve high-performance, energy-efficient data storage and movement, by leveraging NVRAMs to replace the traditional memory technologies and redesign the memory/storage stack. In particular, this dissertation presented three contributions.

The first contribution is a bandwidth-aware reconfigurable cache hierarchy (BARCH) design method applied to CPU systems. It consists of three components: the hybrid cache hierarchy, the reconfiguration method, and the prediction engine. The hybrid cache hierarchy leverages different memory technologies to provide an optimized bandwidth-capacity curve to the on-chip memory system. Based on such hybrid cache hierarchy, we dynamically reconfigure the cache space at each level adaptive to the demands of different applications. We also present an accurate statistical prediction engine to facilitate such reconfiguration. We evaluate the proposed design method with both multithreaded and multiprogrammed workloads. Experimental results show that reconfigurable hybrid cache leads to 58% and 14% performance improvements to multithreaded and multiprogrammed applications, respectively.

The second contribution is an energy-efficient graphics memory designs for GPU systems. We have developed a hybrid graphics memory that improves both memory bandwidth and system energy efficiency. The key insight in our work is that hybrid graphics memory design is especially suitable for GPU applications. The memory access patterns of these applications are naturally used to hide the latency issue of NVMs. Our initial results are very promising for future GPU systems, improving 33% in system energy efficiency. Our migration mechanism limits the frequency of write operations, and therefore we do not expect significantly degradation of the lifetime.

Finally, this dissertation presented our studies on re-architecting the memory and storage stack to design a persistent memory, by leveraging NVRAM's unique feature of incorporating memory and storage properties in a single device. I have presented a persistent memory design, Kiln, which employs a multiversioned memory hierarchy consisting of an NV cache and NV memory, enabling in-place updates to in-memory data structures, without the redundant writes required by logging or COW. Kiln provides persistence support with up to 2$\times$ performance improvement to the log-based NVRAM persistent memory. This dissertation also presented a persistence-aware memory scheduling scheme, FIRM, which achieves both fair memory accesses and high system throughput for the co-running applications. FIRM classifies persistent writes from other memory requests, coordinates their bandwidth usage with other concurrent memory requests, and increases overall system bandwidth utilization by augmenting bank-level parallelism of persistent writes. The experimental results show that FIRM can significantly improves both throughput and fairness compared with previous memory scheduling designs.

## 7.2 Future Research Directions

Future research in re-architecting the memory hierarchy with NVRAMs can proceed in several different directions.

### 7.2.1 Re-architecting the Memory/Storage Stack

NVRAMs incorporate both memory and storage properties, promising to disrupt current two-level memory/storage stack with the capability of accommodating fast accesses to permanent data storage in a unified nonvolatile memory. This feature brings new opportunities to address the massive online data storage and processing requirements of big data applications, allowing them to directly access permanent data storage in memory without the performance and energy overheads of transferring data from/to storage. Unfortunately, current hardware and hardware/software interface are optimized for the two-level memory/storage stack with vastly discrepant speed (fast memory and slow storage), interfaces (memory buses and storage I/Os), and functions (hardware-accelerated memory access and software managed permanent data storage). Consequently, architects need to redesign the memory/storage stack to unify the two functions through a memory interface with optimized system energy efficiency and reliability. I am actively working in this area. In particular, this dissertation is the first to demonstrate substantial performance improvement on a stand-alone computer with hardware-based management of permanent data storage in memory, eliminating the performance and energy overheads of storage systems (e.g., file systems and databases) software mechanisms. Yet this is only an initial step. I wish to conduct substantial follow-up research to address various design challenges.

- **Memory hierarchy design:** Memory hierarchy (processor caches, memory controllers, memory buses, and their management mechanisms) has been well-studied for fast access but not for efficient, reliable accesses to permanent data. For instance, this dissertation shows a large system performance degradation with current memory controller designs when applications need to update permanent data stored in memory 6. We addressed this issue by designing sophisticated memory scheduling policies that balance the service of reads and writes. Memory controller is just one of the various components in the memory hierarchy. It will be interesting to investigate the optimal design of various components and mechanisms, e.g., data mapping in shared caches and memory address space, cache coherence, and data prefetching.

- **Reliability:** Current storage system software supports strong reliability by replicating data in disks (e.g., with RAID techniques). With permanent data stored in memory, copying data will stress memory bus bandwidth and capacity with more than doubled redundant reads and writes. To address this issue, It will be interesting to investigate hardware acceleration for reliability management. I would like to start from exploring initial approaches based on novel circuit and architecture designs: mitigating the memory bus stress by modifying memory circuit designs to enable data copies with internal buses; reducing redundant memory copies by continuously monitoring memory errors and replicating data only needed.

- **Security/privacy:** Storing permanent data in memory also motivates the development of new security/ privacy features. Existing architectural support for security, such as paging and segmentation, can only protect data from being corrupted, but not permanently retain them. Software-based encryption used in storage systems are complex and time consuming. Consequently, it is time to rethink architectural support for security/privacy in light of critical data being permanently stored in memory.

- **From single node to distributed systems (e.g., data centers):** I also wish to investigate the scalability of the solutions to aforementioned challenges in distributed systems. Furthermore, distributed systems may raise additional challenges to memory system designs, e.g., data portability.

### 7.2.2 Hybrid Memory Architectures

It is also interesting to extend our exploration to the overall memory hierarchy design, including on-chip caches, shared memories, and off-chip memories. Furthermore, we want to explore energy optimization techniques to enhance the energy efficiency of the system. For example, instead of considering the application's bandwidth demand as the only metric for reconfiguration, we can reconfigure the cache hierarchy based on a cost function of both bandwidth demand and power consumption predictions. Another interesting research direction is to improve the memory lifetime. For example, in the hybrid graphics memory, the STT-MRAM can be employed as the replacement of portions of ReRAM with error bits. How to balance between performance and power of such design needs to be studied.

# Bibliography

[1] Wu, X., J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie (2009) "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the International Symposium on Computer Architecture*, pp. 34–45.

[2] Sun, G., X. Dong, Y. Xie, J. Li, and Y. Chen (2009) "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 239–249.

[3] Qureshi, M. K., J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali (2009) "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14–23.

[4] Lee, B. C., E. Ipek, O. Mutlu, and D. Burger (2009) "Architecting Phase Change Memory As a Scalable Dram Alternative," in *International Symposium on Computer Architecture*, pp. 2–13.

[5] McKee, S. A. (2004) "Reflections on the memory wall," in *Proceedings of the Conference on Computing Frontiers*, p. 162.

[6] Burger, D., J. R. Goodman, and A. Kägi (1996) "Memory bandwidth limitations of future microprocessors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 78–89.

[7] et al, B. M. R. (2009) "Scaling the bandwidth wall: challenges in and avenues for CMP scaling," in *Proceedings of the International Symposium on Computer Architecture*, pp. 371–382.

[8] Huh, J., D. Burger, and S. W. Keckler (2001) "Exploring the design space of future CMPs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 199–210.

[9] Lindholm, E., J. Nickolls, S. Oberman, and J. Montrym (2008) "NVIDIA Tesla: a unified graphics and computing architecture," *IEEE Micro*, **28**, pp. 39–55.

[10] Yu, C. and P. Petrov (2010) "Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms," in *Proceedings of the Design Automation Conference*, pp. 132–137.

[11] AMD (2012), "AMD Radeon$^{\mathrm{TM}}$HD 7970 Graphics," Http://www.amd.com/us/products/desktop/graphics/7000/7970/Pages/radeon-7970.aspx.

[12] NVIDIA (2010), "Quadro 6000 - Workstation Graphics Card for 3D Design, Styling, Visualization, CAD, and More," Http://www.nvidia.com/object/product-quadro-6000-us.html.

[13] Elpida (2010) "Introduction to GDDR5 SGRAM," .

[14] Zhao, J., C. Xu, and Y. Xie (2011) "Bandwidth-aware Reconfigurable Cache Design with Hybrid Memory Technologies," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 48–55.

[15] Janzen, J., "The Micron system-power calculator," Http://www.micron.com/products/dram/syscalc.html.

[16] Zhao, J. and Y. Xie (2012) "Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 81–87.

[17] Zhao, J., S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi (2013) "Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support," in *Proceedings of the 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, Washington, DC, USA.

[18] Waser, R. (2009) "Resistive Non-volatile Memory Devices," *Microelectron. Eng.*, **86**(7-9), pp. 1925–1928.

[19] Hosomi, M., H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano (2005) "A novel nonvolatile memory with spin

torque transfer magnetization switching: spin-RAM," in *Electron Devices Meeting, IEDM Technical Digest*, pp. 459–462.

[20] Zhao, W., E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle (2006) "Macro-model of Spin-Transfer Torque based Magnetic Tunnel Junction device for hybrid Magnetic-CMOS design," in *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, pp. 40–43.

[21] Raoux, S., G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam (2008) "Phase-change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development*, **52**(4), pp. 465–479.

[22] Viking Technology (2012), "Understanding non-volatile memory technology whitepaper," Http://www.vikingtechnology.com/uploads/nv_whitepaper.pdf.

[23] Degraeve, R., A. Fantini, S. Clima, B. Govoreanu, L. Goux, Y. Y. Chen, D. Wouters, P. Roussel, G. Kar, G. Pourtois, S. Cosemans, J. Kittl, G. Groeseneken, M. Jurczak, and L. Altimime (2012) "Dynamic hourglass model for SET and RESET in HfO2 RRAM," in *Proceedings of the Symposium on VLSI Technology*, pp. 75–76.

[24] Goux, L., A. Fantini, G. Kar, Y. Chen, N. Jossart, R. Degraeve, S. Clima, B. Govoreanu, G. Lorenzo, G. Pourtois, D. Wouters, J. Kittl, L. Altimime, and M. Jurczak (2012) "Ultralow sub-500nA operating current high-performance TiN\Al2O3\HfO2\Hf\TiN bipolar RRAM achieved through understanding-based stack-engineering," in *Proceedings of the Symposium on VLSI Technology*, pp. 159–160.

[25] Cagli, C. (2012) "Characterization and Modelling of Electrode Impact in HfO2-based RRAM," in *Proceedings of the Memory Workshop*.

[26] Sousa, V. (2012) "Phase change materials engineering for RESET current reduction," in *Proceedings of the Memory Workshop*.

[27] Kim, K.-H., S. Hyun Jo, S. Gaba, and W. Lu (2010) "Nanoscale resistive memory with intrinsic diode characteristics and long endurance," *Applied Physics Letters*, **96**(5), pp. 053 106.1–053 106.3.

[28] Lin, W. S., F. T. Chen, C. H. L. Chen, and M.-J. Tsai (2010) "Evidence and solution of over-RESET problem for HfOx based resistive memory with sub-ns switching speed and high endurance," in *Proceedings of the International Electron Devices Meeting*, pp. 19.7.1–19.7.4.

[29] Kim, Y.-B., S. Lee, D. Lee, C. Lee, M. Chang, J. H. Hur, M.-J. Lee, G.-S. Park, C. J. Kim, U. Chung, I.-K. Yoo, and K. Kim (2011) "Bi-layered RRAM with unlimited endurance and extremely uniform switching," in *Proceedings of the Symposium on VLSI Technology*, pp. 52–53.

[30] Ahn, S., Y. Song, C. Jeong, J. Shin, Y. Fai, Y. Hwang, S. Lee, K. Ryoo, S. Lee, J.-H. Park, H. Horii, Y. Ha, J. Yi, B. Kuh, G. Koh, G. Jeong, H. Jeong, K. Kim, and B.-I. Ryu (2004) "Highly manufacturable high density phase change memory of 64Mb and beyond," in *Proceedings of the International Electron Devices Meeting*, pp. 907–910.

[31] Kitagawa, E., S. Fujita, K. Nomura, H. Noguchi, K. Abe, K. Ikegami, T. Daibou, Y. Kato, C. Kamata, S. Kashiwada, N. Shimomura, J. Ito, and H. Yoda (2012) "Impact of ultra low power and fast write operation of advanced perpendicular MTJ on power reduction for high-performance mobile CPU," in *Proceedings of the International Electron Devices Meeting*, pp. 29.4.1–29.4.4.

[32] Yoda, H., S. Fujita, N. Shimomura, E. Kitagawa, K. Abe, K. Nomura, H. Noguchi, and J. Ito (2012) "Progress of STT-MRAM technology and the effect on normally-off computing systems," in *Proceedings of the International Electron Devices Meeting*, pp. 11.3.1–11.3.4.

[33] Abe, K., H. Noguchi, E. Kitagawa, N. Shimomura, J. Ito, and S. Fujita (2012) "Novel hybrid DRAM/MRAM design for reducing power of high performance mobile CPU," in *Proceedings of the International Electron Devices Meeting*, pp. 10.5.1–10.5.4.

[34] Schechter, S., G. H. Loh, K. Straus, and D. Burger (2010) "Use ECP, Not ECC, for Hard Failures in Resistive Memories," in *Proceedings of the International Symposium on Computer Architecture*, pp. 141–152.

[35] Ipek, E., J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda (2010) "Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, ACM, New York, NY, USA, pp. 3–14.

[36] Seong, N. H., D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee (2010) "SAFER: Stuck-At-Fault Error Recovery for Memories," in *Proceedings of the International Symposium on Microarchitecture*, pp. 115–124.

[37] Seong, N. H., D. H. Woo, and H.-H. S. Lee (2010) "Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-change Memory with Dynamically Randomized Address Mapping," in *Proceedings of the International Symposium on Computer Architecture*, pp. 383–394.

[38] Yoon, D. H., N. Muralimanohar, J. Chang, P. Ranganathan, N. Jouppi, and M. Erez (2011) "FREE-p: Protecting non-volatile memory against both hard and soft errors," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 466–477.

[39] (http://www.itrs.net), "International Technology Roadmap for Semiconductors, Process Integration, Devices, and Structures 2010 Update," .

[40] Janesky, J. (2013) "Device performance in a fully functional 800MHz DDR3 Spin Torque Magnetic Random Access Memory," in *IMW*.

[41] Dorsey, P. (2010) "Xilinx stacked silicon interconnect technology delivers breakthrough FPGA capacity, bandwidth, and power efficiency," in *Xilinx White Papers*.

[42] Zhao, J., X. Dong, and Y. Xie (2010) "Cost-aware Three-dimensional (3D) Many-core Multiprocessor Design," in *Proceedings of the 47th Design Automation Conference*, pp. 126–131.

[43] Xie, Y., G. H. Loh, B. Black, and K. Bernstein (2006) "Design Space Exploration for 3D Architectures," *J. Emerg. Technol. Comput. Syst.*, **2**(2), pp. 65–103.

[44] Loh, G. H. (2008) "3D-stacked memory architectures for multi-core processors," in *Proc. of the International Symposium on Computer Architecture*, pp. 453–464.

[45] Dong, X., Y. Xie, N. Muralimanohar, and N. P. Jouppi (2010) "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, IEEE Computer Society, Washington, DC, USA, pp. 1–11.

[46] Kgil, T., S. D'Souza, and A. Saidi et al. (2006) "PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 117–128.

[47] Liu, C. C., I. Ganusov, M. Burtscher, and S. Tiwari (2005) "Bridging the processor-memory performance gap with 3D IC technology," *IEEE Design Test*, **22**, pp. 556–564.

[48] Loi, G. L., B. Agrawal, N. Srivastava, S.-C. Lin, T. Sherwood, and K. Banerjee (2006) "A thermally-aware performance analysis of vertically integrated (3-D) processor-memory hierarchy," in *Proc. of the Design Automation Conference*, pp. 991–996.

[49] Gu, S., P. Marchal, M. Facchini, F. Wang, M. Suh, D. Lisk, and M. Nowak (2008) "Stackable memory of 3D chip integration for mobile applications," in *Proc. of Intl. Electron Devices Meeting*, pp. 1–4.

[50] Woo, D. H., N. H. Seong, L. D.L., and H.-H. Lee (2010) "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *Proc. of the International Conference for High Performance Computing*, pp. 1–12.

[51] Kim, J.-S., C. S. Oh, and H. Lee et al. (2011) "A 1.2V 12.8GB/s 2Gb mobile wide-I/O DRAM with 4 × 128 I/Os using TSV-based stacking," in *Proc. of Intl. Solid-State Circuits Conf.*, pp. 496–498.

[52] Micron (2013), "Hybrid Memory Cube Specification 1.0," .

[53] Tezzaron Semiconductors (2010), "FaStack 3D stackable DRAM," Http://www.tezzaron.com/memory/.

[54] Loi, I. and L. Benini (2010) "An Efficient Distributed Memory Interface for Many-core Platform with 3D Stacked DRAM," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 99–104.

[55] Jevdjic, D., S. Volos, and B. Falsafi (2013) "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *Proceedings of the International Symposium on Computer Architecture*, pp. 404–415.

[56] (2014), "AMD and Hynix announce joint development of HBM memory stacks," Http://electroiq.com/blog/2013/12/amd-and-hynix-announce-joint-development-of-hbm-memory-stacks/.

[57] JEDEC (2013), "High bandwidth memory (HBM) DRAM," Http://www.jedec.org/standards-documents/docs/jesd235.

[58] Lin, C. J., S. H. Kang, Y. J. Wang, K. Lee, X. Zhu, W. C. Chen, X. Li, W. N. Hsu, Y. C. Kao, M. T. Liu, W. C. Chen, Y. C. Lin, M. Nowak, N. Yu, and L. Tran (2009) "45nm low power CMOS logic compatible embedded STT MRAM utilizaing a reverse-connection 1T/1MTJ cell," in *Proceedings of the International Electron Devices Meeting*, pp. 11.6.1–11.6.4.

[59] Ramakrishnan, R. and J. Gehrke (2007) "Database Management Systems, Third Edition," .

[60] Herlihy, M. and J. E. B. Moss (1993) "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the International Symposium on Computer Architecture*.

[61] Park, S., T. Kelly, and K. Shen (2013) "Failure-atomic msync(): a simple and efficient mechanism for preserving the integrity of durable data," in *Proceedings of the European Conference on Computer Systems*.

[62] Lee, H. G. and N. Chang (2003) "Energy-aware Memory Allocation in Heterogeneous Non-volatile Memory Systems," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 420–423.

[63] Huang, Y., T. Liu, and C. J. Xue (2011) "Register Allocation for Write Activity Minimization on Non-volatile Main Memory," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 129–134.

[64] Bathen, L. A. and N. Dutt (2012) "HaVOC: A Hybrid Memory-aware Virtualization Layer for On-chip Distributed ScratchPad and Non-volatile Memories," in *Proceedings of the Design Automation Conference*, pp. 447–452.

[65] Li, Y., Y. Chen, and A. K. Jones (2012) "A Software Approach for Combating Asymmetries of Non-volatile Memories," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 191–196.

[66] Narayanan, V., V. Saripalli, K. Swaminathan, R. Mukundrajan, G. Sun, Y. Xie, and S. Datta (2011) "Enabling Architectural Innovations Using Non-volatile Memory," in *Proceedings of the Great Lakes Symposium on Great Lakes Symposium on VLSI*, pp. 439–444.

[67] Roberts, D., T. Kgil, and T. Mudge (2009) "Using Non-volatile Memory to Save Energy in Servers," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 743–748.

[68] Condit, J., E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee (2009) "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, ACM, New York, NY, USA, pp. 133–146.

[69] Volos, H., A. J. Tack, and M. M. Swift (2011) "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, ACM, New York, NY, USA, pp. 91–104.

[70] Coburn, J., A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson (2011) "NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 105–118.

[71] Venkataraman, S., N. Tolia, P. Ranganathan, and R. H. Campbell (2011) "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, pp. 1–15.

[72] Lee, E., H. Bahn, and S. H. Noh (2013) "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, pp. 73–80.

[73] Wu, M. and W. Zwaenepoel (1994) "eNVy: A Non-volatile, Main Memory Storage System," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 86–97.

[74] Butterworth, P., A. Otis, and J. Stein (1991) "The GemStone Object Database Management System," *Commun. ACM*, **34**(10), pp. 64–77.

[75] Singhal, V., V. Kakkad, and P. R. Wilson (1992) "Texas: an efficient, portable persistent store," in *Proceedings of the International Workshop on Persistent Object Systems*, pp. 11–33.

[76] White, S. J. and D. J. DeWitt (1995) "QuickStore: A High Performance Mapped Object Store," *The VLDB Journal*, **4**(4), pp. 629–673.

[77] Lamb, C., G. Landis, J. Orenstein, and D. Weinreb (1991) "The ObjectStore Database System," *Commun. ACM*, **34**(10), pp. 50–63.

[78] Andrews, T. and C. Harris (1987) "Combining Language and Database Advances in an Object-oriented Development Environment," in *Proceedings of Object-oriented Programming Systems, Languages and Applications*, pp. 430–440.

[79] Cattell, R. G. (1994) *Object Data Management: Object-Oriented and Extended*, 1st ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[80] (http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html), "Java persistence API," .

[81] Marquez, A., J. N. Zigman, and S. M. Blackburn (2000) "Fast Portable Orthogonally Persistent Java," *Softw. Pract. Exper.*, **30**(4), pp. 449–479.

[82] Satyanarayanan, M., H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler (1993) "Lightweight Recoverable Virtual Memory," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 146–160.

[83] Chen, P. M., W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell (1996) "The Rio File Cache: Surviving Operating System Crashes," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 74–83.

[84] Sears, R. and E. Brewer (2006) "Stasis: Flexible Transactional Storage," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pp. 29–44.

[85] Hagmann, R. (1987) "Reimplementing the Cedar File System Using Logging and Group Commit," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 155–162.

[86] Tweedie, S. C. (1987) "Journaling the Linux ext2fs sile system," in *Linux Expo*.

[87] Lowell, D. E. and P. M. Chen (1997) "Free Transactions with Rio Vista," in *Proceedings of the ACM Symposium on Operating Systems Principles*, pp. 92–101.

[88] Giles, E., K. Doshi, and P. Varman (2013) "Bridging the Programming Gap Between Persistent and Volatile Memory Using WrAP," in *Proceedings of the International Conference on Computing Frontiers*, pp. 30:1–30:10.

[89] Hitz, D., J. Lau, and M. Malcolm (1994) "File System Design for an NFS File Server Appliance," in *Proceedings of the USENIX Winter 1994 Technical Conference*, pp. 19–19.

[90] Bressoud, T. C., T. Clark, and T. Kan (2001) "The Design and Use of Persistent Memory on the DNCP Hardware Fault-Tolerant Platform," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 1–12.

[91] Copeland, G., T. Keller, R. Krishnamurthy, and M. Smith (1989) "The Case for Safe RAM," in *Proceedings of the International Conference on Very Large Data Bases*, pp. 327–335.

[92] Eskesen, F., M. Hack, A. Iyengar, R. King, and N. Halim (1998) "Software Exploitation of a Fault-Tolerant Computer with a Large Memory," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, pp. 336–345.

[93] Caulfield, A. M., T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson (2012) "Providing Safe, User Space Access to Fast, Solid State Disks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 387–400.

[94] Narayanan, D. and O. Hodson (2012) "Whole-system Persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, ACM, New York, NY, USA, pp. 401–410.

[95] Xu, C., X. Dong, N. P. Jouppi, and Y. Xie (2011) "Design implications of memristor-based RRAM cross-point structures," in *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 1–6.

[96] Sun, G., C. Hughes, C. Kim, J. Zhao, C. Xu, Y. Xie, and Y.-K. Chen (2011) "Moguls: a model to explore memory hierarchy for throughput computing," *to appear in Proceedings of the International Symposium on Computer Architecture.*

[97] Flautner, K., N. S. Kim, S. Martin, D. Blaauw, and T. Mudge (2002) "Drowsy caches: simple techniques for reducing leakage power," in *Proceedings of the International Symposium on Computer Architecture*, pp. 148–157.

[98] Chen, S. F. and J. Goodman (1996) "An empirical study of smoothing techniques for language modeling," in *Proceedings of the Annual Meeting on Association for Computational Linguistics*, pp. 310–318.

[99] Magnusson, P. S., M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner (2002) "Simics: a full system simulation platform," *IEEE Transactions on Computer*, **35**(2), pp. 50–58.

[100] Bienia, C., S. Kumar, J. P. Singh, and K. Li (2008) "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 239–249.

[101] SPEC OMP, “SPEC OMP2001,” Http://www.spec.org/omp/.

[102] SPEC CPU, “SPEC CPU2006,” Http://www.spec.org/cpu2006/.

[103] Gebhart, M., D. R. Johnson, and D. e. a. Tarjan (2011) “Energy-efficient mechanisms for managing thread context in throughput processors,” in *Proceeding of the International symposium on Computer architecture*, pp. 235–246.

[104] Yu, W.-K. S., R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh (2011) “SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading,” in *Proc. of the International symposium on Computer architecture*, pp. 247–258.

[105] Ren, D. Q. and R. Suda (2010) “Modeling and optimizing the power performance of large matrices multiplication on multi-core and GPU platform with CUDA,” in *Proc. of the International Conf. on Parallel Processing and Applied Mathematics*, pp. 421–428.

[106] Al Maashri, A., G. Sun, X. Dong, V. Narayanan, and Y. Xie (2009) “3D GPU architecture using cache stacking: performance, cost, power and thermal analysis,” in *Proc. of the International Conferenece on Computer Design*, pp. 254–259.

[107] Galal, S. and M. Horowitz (2011) “Energy-efficient floating-point unit design,” *IEEE Trans. on Computers*, **60**(7), pp. 913 –922.

[108] Wang, P.-H., Y.-M. Cheng, C.-L. Yang, and Y.-J. Cheng (2009) “A predictive shutdown technique for GPU shader processors,” *Computer Architecture Letters*, **8**(1), pp. 9–12.

[109] NVIDIA (2008), “PowerMizer 8.0 Intelligent Power Management Technology,” Http://www.nvidia.com/object/feature_powermizer.html.

[110] Jiao, Y., H. Lin, P. Balaji, and W. Feng (2010) “Power and Performance Characterization of Computational Kernels on the GPU,” in *Proceedings of the International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing*, pp. 221–228.

[111] Samsung, “DDR3 and GDDR5,” Http://www.samsung.com/global/ business/semiconductor/products/Products.html.

[112] Herbert, S. and D. Marculescu (2007) “Analysis of dynamic voltage/frequency scaling in chip-multiprocessors,” in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 38–43.

[113] Kaxiras, S. and M. Martonosi (2008) *Computer Architecture Techniques for Power-Efficiency*, 1st ed., Morgan and Claypool Publishers.

[114] Isci, C., G. Contreras, and M. Martonosi (2006) "Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management," in *Proceedings of the International Symposium on Microarchitecture*, pp. 359–370.

[115] Wu, Q., M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks (2005) "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," in *Proceedings of the International Symposium on Microarchitecture*, MICRO 38, pp. 271–282.

[116] David, H., C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu (2011) "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *Proceedings of the International Conference on Autonomic Computing*, pp. 31–40.

[117] Deng, Q., D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini (2011) "MemScale: Active Low-power Modes for Main Memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 225–238.

[118] Deng, Q., D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini (2012) "CoScale: Coordinating CPU and Memory System DVFS in Server Systems," in *Proceedings of the International Symposium on Microarchitecture*, pp. 143–154.

[119] Bakhoda, A., G. Yuan, W. Fung, H. Wong, and T. Aamodt (2009) "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. of Intl. Symp. on Performance Analysis of Systems and Software*, pp. 163–174.

[120] Hynix, "GDDR5 SGRAM datasheet," Http://www.hynix.com/products/graphics/.

[121] NVIDIA, "CUDA SDK," Http://www.nvidia.com/object/cudasdks.html.

[122] Che, S., M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron (2009) "Rodinia: a benchmark suite for heterogeneous computing," in *Proc. of Intl. Symp. on Workload Characterization*, pp. 44–54.

[123] Li, S., J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi (2009) "McPAT: an integrated power, area, and timing

modeling framework for multicore and manycore architectures," in *Proc. of the International Symposium on Microarchitecture*.

[124] Vick, E., S. Goodwin, G. Cunnigham, and D. S. Temple (2012) "Vias-last Process Technology for Thick 2.5D Si Interposers," in *3D Systems Integration Conference*, pp. 1–4.

[125] Intel Corporation (2012) "Intel architecture instruction set extensions programming reference, 319433-012 edition," .

[126] Jacobi et al., C. (2012) "Transactional memory architecture and implementation for IBM System Z," in *MICRO*.

[127] Wang, A., M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael (2012) "Evaluation of Blue GeneQ Hardware Support for Transactional Memories," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pp. 127–136.

[128] Moore, K. E., J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood (2006) "LogTM: log-based transactional memory," in *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, pp. 1–12.

[129] Atikoglu, B., Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny (2012) "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pp. 53–64.

[130] Kim, J.-S., C. S. Oh, H. Lee, D. Lee, H. R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S. K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, J. S. Choi, and Y.-H. Jun (2012) "A 1.2 V 12.8 GB/s 2 Gb mobile wide-I/O DRAM with 4x 128 I/Os using TSV based stacking," *IEEE Journal of Solid-State Circuits*, **47**, pp. 107–115.

[131] Pawlowski, J. (2011) "Hybrid memory cube," in *Proceedings of the Hot Chips*.

[132] Ahn, J. H., S. Li, O. Seongil, and N. Jouppi (2013) "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 74–85.

[133] Luk, C.-K., R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood (2005) "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, ACM, New York, NY, USA, pp. 190–200.

[134] (http://www.intel.com/content/www/us/en/processors/ core/core-i7-processor.html), "Intel Core i7," .

[135] Dong, X., C. Xu, Y. Xie, and N. Jouppi (2012) "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **31**(7), pp. 994–1007.

[136] Hammond, L., V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun (2004) "Transactional Memory Coherence and Consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 102–113.

[137] Bingmann, T. (http://panthema.net/2007/stx-btree), "STX B+ Tree, Sept. 2008," .

[138] Siek et al., J. (http://www.boost.org/doc/libs/), "Boost: adjacency list, ver. 1.52.0," .

[139] Bader, D. A. and K. Madduri (2005) "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *Proceedings of the 12th International Conference on High Performance Computing*, pp. 465–476.

[140] Li, S., J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi (2009) "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480.

[141] Kannan, S., A. Gavrilovska, and K. Schwan (2014) "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 1–12.

[142] Zhou, P., B. Zhao, J. Yang, and Y. Zhang (2009) "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology,"

in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 14–23.

[143] Qureshi, M. K., V. Srinivasan, and J. A. Rivers (2009) "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, ACM, New York, NY, USA, pp. 24–33.

[144] Meza, J., J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan (2012) "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE Comput. Archit. Lett.*, **11**(2), pp. 61–64.

[145] Yoon, H., J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu (2012) "Row Buffer Locality Aware Caching Policies for Hybrid Memories," in *International Conference on Computer Design*, pp. 1–8.

[146] Dhiman, G., R. Ayoub, and T. Rosing (2009) "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, ACM, New York, NY, USA, pp. 664–469.

[147] Lee, B. C., E. Ipek, O. Mutlu, and D. Burger (2010) "Phase Change Memory Architecture and the Quest for Scalability," *Commun. ACM*, **53**(7), pp. 99–106.

[148] Lee, B. C., P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger (2010) "Phase-Change Technology and the Future of Main Memory," *IEEE Micro*, **30**(1), pp. 143–143.

[149] Li, J. (2012) "A Case for Small Row Buffers in Non-volatile Main Memories," in *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD 2012)*, ICCD '12, IEEE Computer Society, Washington, DC, USA, pp. 484–485.

[150] Chatterjee, N., N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi (2012) "Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads," in *International Symposium on High-Performance Computer Architecture*, pp. 1–12.

[151] Liu, R.-S., D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang (2014) "NVM Duet: Unified Working Memory and Persistent Store Architecture," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 455–470.

[152] Venkataraman, S., N. Tolia, P. Ranganathan, and R. H. Campbell (2011) "Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, USENIX Association, Berkeley, CA, USA, pp. 5–5.

[153] Kim, Y., V. Seshadri, D. Lee, J. Liu, and O. Mutlu (2012) "A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 368–379.

[154] Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens (2000) "Memory Access Scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, ACM, New York, NY, USA, pp. 128–138.

[155] Rixner, S. (2004) "Memory Controller Optimizations for Web Servers," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, IEEE Computer Society, Washington, DC, USA, pp. 355–366.

[156] Moscibroda, T. and O. Mutlu (2007) "Memory Performance Attacks: Denial of Memory Service in Multi-core Systems," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, USENIX Association, Berkeley, CA, USA, pp. 18:1–18:18.

[157] Mutlu, O. and T. Moscibroda (2007) "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," in *Proceedings of the International Symposium on Microarchitecture*, pp. 146–160.

[158] Kim, Y., D. Han, O. Mutlu, and M. Harchol-Balter (2010) "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12.

[159] Kim, Y., M. Papamichael, O. Mutlu, and M. Harchol-Balter (2010) "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," in *International Symposium on Microarchitecture*, pp. 65–76.

[160] Muralidhara, S. P., L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda (2011) "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, ACM, New York, NY, USA, pp. 374–385.

[161] Mutlu, O. and T. Moscibroda (2008) "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," in *International Symposium on Computer Architecture*, pp. 63–74.

[162] Ausavarungnirun, R., K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu (2012) "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, IEEE Computer Society, Washington, DC, USA, pp. 416–427.

[163] Ebrahimi, E., R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt (2011) "Parallel Application Memory Scheduling," in *Proceedings of the International Symposium on Microarchitecture*, pp. 362–373.

[164] "Digramic bayesian classifier," Http://dbacl.sourceforge.net.

[165] "FFmpeg," Http://www.ffmpeg.org/.

[166] Mao, Y., E. Kohler, and R. T. Morris (2012) "Cache craftiness for fast multicore key-value storage," in *European Conference on Computer Systems*.

[167] Mladenic, D. (2001) "Uisng text learning to help Web browsing," in *SIGCHI*.

# Vita

## Jishen Zhao

Jishen Zhao received B.E. and M.E. degrees from Zhejiang University in China. She is currently a Ph.D. Candidate in the Computer Science and Engineering Department at the Pennsylvania State University. She works in the Microsystems Design Laboratory (MDL) under the supervision of Professor Yuan Xie. Her research is concerned with a broad range of computer architecture and electronic design automation, with a particular emphasis on memory systems, high-performance computing, and energy efficiency. Her past research results in 13 papers in top venues of computer architecture and electronic design automation, as well as 4 US patents. She is on the TPC of Workshop on Architectures and Systems for Big Data (ASBD) 2014. She has also served as a TPC for Great Lakes Symposium on VLSI (GLSVLSI) 2014 and peer reviewer for several conferences in the field of computer architecture, electronic design automation, and VLSI design, including Design Automation Conference (DAC) 2012, and Design Automation and Test in Europe Conference (DATE) 2012, International Symposium on High-performance Computer Architecture (HPCA) 2013, and International Conference on Parallel Architectures and Compilation Techniques (PACT) 2013.