The Pennsylvania State University
The Graduate School
The Department of Computer Science and Engineering

ARCHITECTURE-LEVEL DESIGNS USING EMERGING NON-VOLATILE MEMORIES

A Dissertation in
Computer Science and Engineering
by
Jue Wang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2014

The dissertation of Jue Wang was reviewed and approved* by the following:

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

Mary Jane Irwin
Professor of Computer Science and Engineering
Robert E. Noll Professor
Evan Pugh Professor
Dissertation Co-Advisor, Co-Chair of Committee

Mahmut Taylan Kandemir
Professor of Computer Science and Engineering

Xiaolong Zhang
Associate Professor of Information Sciences and Technology

Raj Acharya
Professor of Computer Science and Engineering
Department Head

*Signatures are on file in the Graduate School.

Abstract

SRAM and DRAM have been used to build our memory systems for decades, but their scalability is facing more and more challenges in terms of leakage power and density. Meanwhile, new emerging non-volatile memory technologies (NVMs) are being explored, such as Phase-Change RAM (PCM or PCRAM), Spin-Torque Transfer RAM (STTRAM or MRAM), and Resistive RAM (ReRAM). They have common advantages of high density, low standby power and non-volatility. It could bring benefits by using NVMs to replace SRAM and DRAM in our memory systems.

However, NVM technologies still have some disadvantages. First, the NVM write operation is much more expensive in terms of longer latency and higher energy. It causes negative impacts on the system performance and energy efficiency. Second, NVMs usually have limited write endurance, which brings challenges on system reliability. Last but not least, the size of NVM sense amplifier is larger, and how to maintain the area utilization is an issue. All of these NVM characteristics are caused by their basic mechanisms, and they are very difficult to be improved by changing cell designs. Therefore, new architecture techniques are necessary for mitigating these issues and building efficient and reliable systems with NVMs.

In this dissertation, NVMs are evaluated as alternatives of traditional memory technologies for different memory levels. We explore NVMs as main memory systems, on-chip caches and GPGPU register files. We analyze their impact on system level and propose several techniques on architecture level to mitigate their disadvantages. We believe these techniques make NVMs more attractive in the future computer systems.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank my advisor, Professor Yuan Xie. He encouraged me to begin my Ph.D. journey, and gave me unconditional help throughout my Ph.D. study. He gave me suggestions about how to pick topics and how to do research step by step, and he encouraged me when I faced hard times. He is always supportive no matter what happens. I really appreciate all his help, which let me grow not only in research but also in personality.

I also would like to thank my committee members, Professor Mary Jane Irwin, Professor Mahmut Taylan Kandemir and Professor Xiaolong Zhang, for their great supports and invaluable feedback on my research work.

Besides, I want to thank my colleagues and friends from MDL group: Yang Ding, Xiaoxia Wu, Guangyu Sun, Yibo Chen, Jin Ouyang, Tao Zhang, Dimin Niu, Jishen Zhao, Jing Xie, Cong Xu, Matt Poremba, Jia Zhan, Ping Chi, Ivan Stalev, Yang Zheng, Shuangchen Li and Ziyang Qi, for their help on both my research and life. Especially, I want to thank my roommates, Qiaosha Zou and Hsiang-Yun Cheng, for so caring and friendly. The life in PSU is precious memory for me.

I also want to thank my co-authors and co-workers, Dr. Norm P. Jouppi from Hewlette-Packard Labs, Xiaochun Zhu and Seung Kang from Qualcomm. They gave me a lot of valuable advises on my work, and I appreciate the opportunities to work with these leading researchers.

While I was an intern at Qualcomm GPU group, I got a lot of help from my managers and collaborators, including but not limited to Golf Jiao, Raghu, Jianxun Liang, Long Chen, Sherry Shao, Guocai Zhu, Priyanka and Jian Liang. I have learned a lot from them and we worked together to deliver the next generation Qualcomm GPU. It was very exciting.

Finally, I would like to thank my family. I want to thank my parents for their support in every decision I've made. I also want to thank my husband, Xiangyu Dong, for his unconditional love. He has always encouraged and pushed me to do better. Last but not least, I want to thank my daughter, Sylvia, who is the sunshine of my life.

Chapter 1

Introduction

In this chapter, we first introduce the opportunities and the challenges of using emerging non-volatile memories as different memory subsystems, and then we give the organization of the remaining chapters.

1.1 Opportunities and Challenges of Emerging Non-volatile Memories

While Moore's law is still offering smaller and faster transistors, the improvement rate of microprocessor has exceeded the one of memory. The integration of multiple cores on a single die accentuates the already daunting *memory-wall* problem. The memory hierarchy design has to catch up and tremendously scale in performance, energy-efficiency, and storage capacity to sustain the processing demands of a wide variety of next-generation applications. Traditionally, SRAM and DRAM are the common technologies to build on-chip caches and main memory. However, they are facing constraints of cell area and leakage energy consumption with technology scaling.

Recently, some new emerging non-volatile memory (NVM) technologies, such as *Phase-Change RAM (PCM)*, *Spin-Torque Transfer RAM (STTRAM or MRAM)*, and *Resistive RAM (ReRAM)*, have been explored. Compared to SRAM and DRAM, NVMs have common advantages of high density, low standby power, better scalability, and non-volatility. Thus, it is attractive to replace SRAM/DRAM with

Table 1.1. Characteristics of different memory technologies.

	SRAM	DRAM	PCM	STTRAM	ReRAM
Cell Area (F^2)	50-200	6	4-10	6-20	4
Read Power	Low	Low	Low	Low	Low
Write Power	Low	Low	High	High	High
Other Power	Leakage Current	Refresh Power	None	None	None
Write Latency	Similar to Read	Similar to Read	Long	Long	Long
Write Endurance	Unlimited	Unlimited	10^8	10^{12}-10^{15}	10^{11}

NVMs given the potential energy and cost saving opportunities. Consistent with this expectation, such a technology shift is happening. For example, PCM-based storage [2, 3, 4] and main memory [5, 6, 7, 8, 9, 10, 11, 12, 13] have already been investigated, STT-RAM based on-chip caches [14, 15, 16, 17, 18] have also been heavily studied. On the industry side, HP and Hynix plan to use ReRAM products as flash replacement and later for DRAM/SRAM as well. In addition, Toshiba revealed their plan in using STT-RAM to replace a 512KB SRAM L2 cache for low power purpose [19]. Another example is that replacing DRAM with STT-RAM in data centers can reduce power by up to 75% [20].

However, adopting NVMs in practical products is not straightforward. As listed in Table 1.1, although NVMs usually have smaller cell area and zero standby leakage power compared to SRAM/DRAM, they have common disadvantages, which are limited write endurance and expensive write operations. These disadvantages of NVMs bring negative impacts on system performance, energy efficiency, and system reliability. Worse, the impact might be different when NVMs are adopted in different memory hierarchy levels. Therefore, we need to evaluate and address these issues carefully.

1.1.1 Exploring NVMs as Main Memories

Main memories consume a significant portion of power in computing devices. For example, the DRAM in a smartphone can consume up to 34.5% of the total power [21]. How to improve the memory subsystem power efficiency is a key question in designing future devices. To address this, DRAM power reduction techniques have been proposed [22, 23, 24, 25, 26]. However, DRAM is still a volatile memory technology and needs refresh. Previous work [27] shows that DRAM re-

fresh could contribute as much as 20% of total DRAM power, and this refresh overhead will become larger along the DRAM scaling roadmap [28, 29]. Thus, it is the time to explore alternative memory technologies.

NVMs, such as PCM and STT-RAM, are promising candidates for main memory systems. They have many attractive features including non-volatility, negligible standby leakage, fast read access, high cell density, and superior scalability [30]. However, enjoying these benefits is not free. First, NVMs require large write energy. For example, PCM consumes 2.2 times more energy than DRAM does [6]. Such high write energy offsets the energy saving from NVM's small leakage power. Second, NVM main memory usually has smaller page size. It lowers memory page hit ratio, and thus the performance of some workload might be degraded. Last but not least, NVM chip internal structure is not compatible with DRAM interfaces. This compatibility is not only highly required for a successful NVM early adoption, but also critical to enable a tiered memory system using refresh-free NVM and high-performance DRAM [31, 5, 32]. Everspin's effort [33] to implement a DDR3-compatible STT-RAM is an example to meet such request. Therefore, it is necessary to design an interface-compatible high performance and low power NVM main memory architecture.

1.1.2 Exploring NVMs as On-Chip Caches

Modern chip multiprocessors (CMP) designs tend to use more on-chip cache hierarchy levels and a larger capacity at each level. Today, SRAM and embedded DRAM (eDRAM) are the common technologies to build on-chip caches. However, neither SRAM nor eDRAM is scalable due to leakage power or cell density concern. NVMs provides benefits in terms of energy and cost savings (via cell size reduction), which makes on-chip NVM-based cache is an attractive option. For example, a 4Mb ReRAM macro [34] can achieve a cell size of $9.5\,F^2$ (15X denser than SRAM) and a random read/write latency of 7.2 ns (comparable to SRAM caches with same capacity). In addition, non-volatility can eliminate the standby leakage energy, which can be as high as 80% of the total energy consumption for an SRAM L2 cache [35].

However, it is still challenging to build NVM-based on-chip caches: caches han-

dle much more writes than storage and main memory do. The first issue is NVM only has a limited write endurance. For example, PCM can only sustain 10^8 writes before experiencing frequent stuck-at-1 or stuck-at-0 errors [8, 10, 9, 11, 12, 13]. The write endurance of ReRAM is much improved but is still limited at 10^{11} [36]. For STTRAM, a prediction of up to 10^{15} write cycles is often cited, but the best STTRAM endurance test result is less than 4×10^{12} cycles [37]. This problem is further amplified by the conventional cache management policies which result in significant non-uniformity in terms of writing to cache blocks, which would cause heavily-written NVM blocks to fail much earlier than most other blocks and reduce the product lifetime. Thus, a new cache management policy is needed for NVM caches. Second, due to process imperfection, a more severe problem of NVM cache is that there might be a portion of cells with worse quality and much shorter write endurance. This means the actual lifetime might be much shorter than the expected one even with perfect wear leveling since the worst quality might determine the entire cache lifetime. Thus, we need error-tolerant techniques for NVM caches. Third, another major disadvantage of NVM is the latency and energy overhead associated with its write operations. In particular, a long NVM write operation might obstruct other cache accesses [38, 17], especially when multiple processes are running in parallel. This could cause a severe performance degradation. Therefore, a mitigation technique to minimize the write overhead is required before any successful NVM cache design.

1.1.3 Exploring NVMs as GPGPU Register Files

GPGPUs usually have a large number of registers to hold states and contents of all the active threads. Compared to the RF in CPUs, the RF in GPGPUs is much larger (e.g., 2MB in total for the top-tier Fermi chip). Therefore, the area cost and the energy consumption of RFs should be carefully evaluated in GPGPU design.

Traditionally, SRAM is used to build the RFs in GPGPUs, but it brings two issues. First, SRAM-based RFs occupy a significant amount of chip area since each SRAM cell has six transistors. Second, SRAM is power-unfriendly because of its large leakage power. NVM is being studied as one of the potential alternatives for SRAM. However, despite the potential benefits of using NVM-based GPGPU

RFs, there are new challenges as well. NVM has similar read latency and energy consumption compared to SRAM, but it has much longer write latency and higher write energy. Thus, replacing SRAM RFs by NVM ones directly brings two problems: First, the GPGPU system performance is degraded significantly considering NVM write latency is much longer than SRAM; Second, the dynamic energy consumption of NVM RFs is much larger than SRAM ones because of its higher write energy, which offsets the benefit coming from the leakage energy saving. To solve these two issues, we need novel RF architecture designs for NVM-based GPGPU systems.

1.2 Organization

In this dissertation, we evaluate the feasibility of adopting NVMs on different memory levels and analyze the impact on system level. Furthermore, we propose several techniques to mitigate the disadvantages of NVMs and make them more attractive for practical products.

The following is the organization of the remaining chapters. Chapter 2 describes the backgrounds of different types of NVMs and the related work. Chapter 3 and Chapter 4 explore NVMs as the main memory. In Chapter 3, we study on multi-level cell PCM main memories, which increases memory density dramatically but has much higher write energy consumption than the single-level cell memory. An energy-efficient architecture is proposed to reduce the programming energy by manipulating the data stored in PCM. In Chapter 4, we study on STTRAM main memories. Several techniques are proposed to design an LPDDR3-compatible high performance and low power architecture. Next, we explore NVMs as on-chip caches. In Chapter 5, we address on the write count variation problem. Since NVMs usually have limited write endurance, the write traffic to each cache line needs to be balanced. A novel and low-overhead wear-levering technique is proposed for NVM caches. In Chapter 6, we study on the inherent variation of NVM cell's lifetime due to process variations and propose a hard-tolerant technique to correct hard errors and extend NVM cache lifetime. Another problem using NVMs caches is their more expensive write operations. We propose a technique to mitigate its impact on system performance in Chapter 7. Afterward, we evaluate

NVMs as GPGPU register files in Chapter 8, and then propose a write-aware NVM-based register file design, which contains two techniques to improve system performance and reduce the expensive write overhead. At last, Chapter 9 gives the conclusion.

Chapter 2

Background and Related Work

In this chapter, we describe the backgrounds of different types of NVMs and then introduce the related work.

2.1 Technology Background

Several different NVM technologies are being explored, such as Phase-Change RAM (PCM or PCRAM), Spin-Torque Transfer RAM (STTRAM or MRAM), and Resistive RAM (ReRAM). These techniques have common advantages of high density, low standby power and non-volatility. The backgrounds of these different NVM types are described in this section.

2.1.1 PCM Technology

PCM (phase change memory) is based on the phase-change behavior of chalcogenide alloys (GST). The data storage capability is achieved by the resistance differences between the amorphous (high-resistance, RESET state) and the crystalline (low-resistance, SET state) phase of GST. As shown in Fig. 2.1, every PCM cell contains one GST and one selector transistor. A small voltage is applied across the GST to read data stored in PCM cells. Stored data bits are sensed by measuring the resulting current since the SET status and the RESET status have a large difference in their equivalent resistances.

PCMs have two major failure modes: *stuck-RESET* and *stuck-SET* [39]. Stuck-

RESET is caused by void formation or delamination that catastrophically disconnects the electrical path between GST (i.e. the storage element of PCM) and access device. Instead, stuck-SET is caused by GST aging that makes GST more reluctant to create an amorphous phase after continuously experiencing write cycles, resulting in a degradation of the PCM RESET-to-SET resistance ratio. Both of these failure modes can be commonly observed. ITRS [40] projects that the average PCM write endurance is around 10^7-10^8.

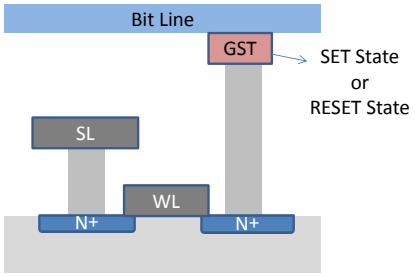

Figure 2.1. The conceptual view of a PCM cell.

2.1.2 STT-RAM Technology

STT-RAM (spin-transfer torque random-access memory) is the second generation of MRAM (Magnetic RAM). The basic storage element in STT-RAM is a magnetic tunnel junction (MTJ) as shown in Fig. 2.2. Each MTJ is composed of two ferromagnetic layers. One has a fixed magnetization direction; the other has a free one and can change its direction. The relative direction of these two layers is used to represent a digital "0" or "1". In STT-RAM technology, a switching current flowing from bitline to sourceline turns the cell into parallel state "0"; a switching current flowing from sourceline to bitline turns the cell into anti-parallel state "1" [38, 41].

MTJ can be unreliable for two reasons: time-dependent dielectric breakdown (TDDB) and resistance drift [42]. TDDB is an abrupt increase of junction current owing to a short forming through the tunneling barrier. Resistance drift is a gradual reduction of the junction resistance over time that can eventually lead to reduced read margin. Due to these two issues, the best STTRAM endurance test result so far is less than 4×10^{12} [37].

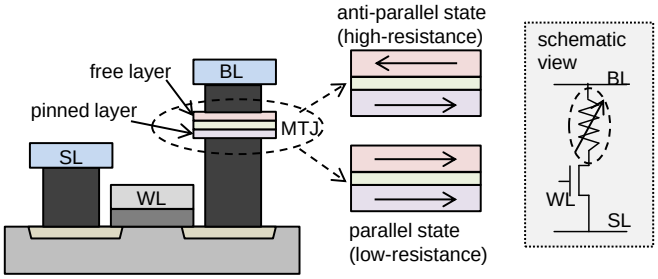

Figure 2.2. The conceptual view of an STT-RAM cell.

2.1.3 ReRAM Technology

Unlike PCM and STTRAM that are based on electrically induced resistive switching effects, we define ReRAM as the one that involves electro- and thermochemical effects in the resistance change of a metal/insulator/metal system. ReRAM offers the possibility of a very high density integration consuming even lower power. As shown in Fig. 2.3, a typical ReRAM cell is based on a solid electrolyte (e.g. oxides) sandwiched by one inert electrode (e.g. Pt) and one electrochemically active electrode (e.g. Cu or Ag). When there is a positive bias applied on the active electrode, metal ions (e.g Cu^+ or Ag^+) are formed. These ions then migrate through the solid electrolyte, and they are eventually discharged at the inert electrode, which leads to a growth of dendrite and forms a highly conductive filament in the ON state of the cell. When the applied voltage changes its polarity, an electrochemical dissolution of the filament takes place, and it switches the cell back to the OFF state[1].

ReRAM has different endurance failure types. One of the them is the anode oxidation induced interface reaction [43]. High temperature, large current/power process, and oxygen ions produced during forming/SET process cause the oxidation at the anode-electrode interface. Another endurance failure mechanism is extra vacancy attributed reset failure effect [43]. Electric field induced extra oxygen vacancy generation during switching may increase the filament size or make the filament rougher, accompanying with the reduced resistance in high-resistance states (R_{HRS}) and resistance in low-resistance states (R_{LRS}) as well as the increased

[1]Another type of ReRAM relies on anion migration (e.g. O^-) instead of ion migration, but the mechanism is the same.

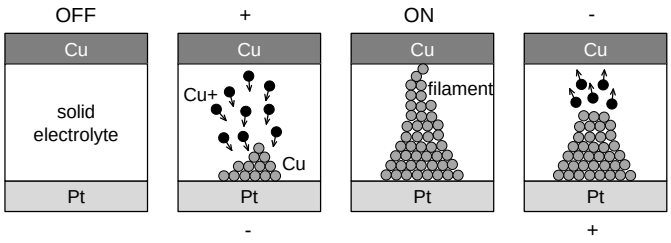

Figure 2.3. The conceptual view of an ReRAM cell and its filament formation and rupture.

reset voltage. Recent ReRAM prototypes demonstrate the best write endurance ranging from 10^{10} [44] to 10^{11} [36].

2.2 Related Work

NVMs can be adopted in different memory levels, such as main memory, on-chip caches, and even register-file. The related work on these topics are introduced in this section.

2.2.1 NVM Main Memory

Reduce write energy overhead of NVM main memory: Recent work has put the focuses on how to reduce the energy overheads of NVM write operations. Data comparison write [45, 7] was proposed to eliminate redundant bit-writes using a read-before-write operation, which can help identify such redundant bits and cancel those redundant write operations to save energy and reduce the impact on performance. Several data inverting schemes [46, 47] were proposed to further reduce the number of bit-writes. To take advantage of the asymmetric RESET and SET energy, Xu *et al.* [48] proposed selective-XOR operations to bias the data value distribution. However, most previous work focused on the SLC PCM without using the special feature of MLC PCM. *Mercury* [49], a fast and energy efficient MLC architecture, is designed to mitigate the MLC overhead by adaptively using reset-to-set or set-to-reset write schemes. Qureshi *et al.* [50] designed an adaptive PCM management infrastructure to dynamically partition MLC PCM into SLC when the memory capacity is over-provisioned. Similarly, *AdaMS* [51] was designed as an adaptive MLC and SLC partitioning architecture for PCM

file storages. Energy-aware data compression [52] was proposed to reduce the programming energy of MLC NAND flash, in which variable-length code reduces the total MLC programming energy, but such a sophisticated is implemented in the software layer.

Performance Optimizations for NVM Main Memory: NVMs include STT-RAM and PCM have been considered as scalable DRAM alternatives in some previous work. Some work has put the focuses on how to reduce the energy overheads and improve the performance of PCM, such as data comparison write [45, 7], data inverting schemes [46, 47], selective-XOR operations [48]. Lee *et al.* [6] proposed the technique to separate sense amplifiers and row buffers in PCM-based main memory, but unlike *EarlyPA*, their technique did not issue precharge commands in advance to hide the latency. Meza *et al.* [53] also noticed the NVM smaller page size problem. However, they treated it as an advantage in enabling energy-efficient fine-grained row activation for server applications where the row hit ratio is inevitably low. The other technique they proposed [54] issues the precharge as soon as sensing completes, and it sends the row address in PRE commands. This technique can be used only in close page policy. For open page policy, the write operation needs additional precharge. Emre *et al.* [55] evaluated STT-RAM as an main memory alternative and proposed a write scheme to bypass the row buffer write bypass, but they did not buffer multiple writes and did not use the separate write path to optimize the row buffer precharge operations for reads. Last but not least, Smullen *et al.* [16] and Sun *et al.* [18] traded off STT-RAM non-volatility for improved write speed and energy.

2.2.2 NVM On-chip Caches

Wear leveling techniques for NVMs: These techniques focused on evenly distributing unbalanced write frequencies to all memory lines. Zhou *et al.* [7] proposed a segment swapping policy for PCM main memory. Qureshi *et al.* proposed fine-grain wear leveling (FGWL) [5] and start-gap wear leveling [8] to shift cache lines within a page to achieve uniform wear out of all lines in the page. Seong *et al.* [9] addressed potential attacks by applying security refresh. However, this previous work was all focused on extending the lifetime of PCM-based main

memory. Other work on NVM caches [56] only extended wear-leveling techniques for main memory without considering the different operating mechanisms of main memory and caches. Some other work focus on the security threat of non-volatile main memories [57, 9, 8], and their key idea is to dynamically change the address mapping.

Error correction techniques for NVMs: The conventional Error Correction Code (ECC) is the most common technique in this category. Dynamically replicated memory [10] reuses memory pages containing hard faults by dynamically forming pairs of pages that act as a single one. Error Correction Pointer (ECP) [11] corrects failed bits in a memory line by recording the position and its correct value. Seong et al. [12] propose SAFER to efficiently partition a faulty line into groups and correct the error in the group. FREE-p [13] proposed an efficient means of implementing line sparing. These architectural techniques add different types of data redundancy to solve the access errors caused by limited write endurance.

Mitigate the performance penalty incurred by long write latencies: Xu *et al.* [14] proposed a dual write speed scheme to improve the average access time of STT-RAM cache. An early write termination scheme was proposed to reduce the unnecessary writes to STT-RAM cells [15]. Sun *et al.* [17] proposed a hybrid cache architecture with read-preemptive write buffers. Smullen *et al.* [16] and Sun *et al.* [18] traded off the STT-RAM non-volatility to improve the write speed and the write energy.

2.2.3 NVM GPGPU RFs

Some previous work focused on GPGPU RF architecture design using traditional SRAM. Gebhart, *et al.* [58] proposed a register file caching and a two-level warp scheduler to hide memory access latency. A compile-time managed multi-level register file hierarchy is also proposed [59]. A power efficient register file is evaluated by aggressively moving a register into drowsy state [60]. Some other work explored how to replace SRAM-based RFs with other memory technologies. Yu, *et al.* [61] proposed a SRAM-DRAM hybrid RFs in fine-grained multi-threading. Jing, *et al.* [62] proposed an eDRAM-based energy-efficient RF design. Goswami, *et al.* [63] proposed a resistive memory based RF with power-performance co-

optimization.

There are also some previous work focused on mitigating the performance penalty caused by long write latencies of NVMs. A dual write speed scheme [14] is proposed by Xu, *et al.* to improve the average access time. A hybrid cache architecture with read-preemptive write buffers [16] is proposed by Sun, *et al.*. Other work traded off the STTRAM non-volatility to improve the write speed [16, 18].

Chapter 3

NVM Main Memory: Expensive Write Operations

Phase-change memory (PCM) is a promising candidate for main memory systems. It has many attractive features including non-volatility, negligible standby leakage, fast read access, high cell density, and superior scalability [30]. PCM has made rapid progress in the recent years, and it is considered to have the read access latency that is comparable to DRAM and the non-volatility like NAND flash [6, 51]. Therefore, there has been extensive research of using PCM at the main memory [5, 6, 50].[1]

Most recently, the feasibility of multi-level cell (MLC) for PCM including programming into two and four bits per cell has been shown[1, 64, 65]. Although MLC increases the PCM bit density, the energy of programming MLC PCM is considerably larger than that of single-level cell (SLC) PCM [51, 49, 50]. The extra energy consumed by MLC PCM comes from the necessity of program-and-verify (P&V) scheme which causes multiple programming steps per MLC write operation for intermediate states, and this effect is similar to the well-known energy consumption difference in MLC and SLC NAND flash designs [66]. The general estimation that PCM consumes 2.2 times more energy than DRAM does [6]. Therefore, in this chapter, we design an energy-efficient architecture for the MLC PCM system

[1]This work is published as "Energy-Efficient Multi-Level Cell Phase-Change Memory System with Data Encoding" on ICCD2011, "Data Encoding for Multi-Level Cell Phase-Change Memory" on NVMW2012.

and reduce the programming energy by manipulating the data stored in MLC PCM [67, 68].

3.1 MLC PCM Background and Energy Model

PCM technology is based on the phase-change behavior of chalcogenide alloys (GST). The data storage capability is achieved by the resistance differences between the amorphous (high-resistance) and the crystalline (low-resistance) phase of GST. To SET the cell into its low-resistance state, an electrical pulse is applied to heat a significant portion of the cell above the crystallization temperature. This SET duration mainly depends on the crystallization speed of GST. Although SET pulses shorter than 10ns have been demonstrated [69], the typical value of the SET pulse duration is around 150ns [70]. On the other side, in the RESET operation, a larger electrical current is applied in order to melt the central portion of the cell. After this pulse is cut off abruptly, the molten material quenches into the amorphous phase. The RESET operation has shorter duration but tends to be current-hungry [70]. Generally, the energy consumptions of full RESET and SET operations are on the same order of magnitude.

Thanks to the large resistance contrast between the RESET and SET states (e.g. $10^2 - 10^3$), MLC PCM becomes feasible. However, the degree of success of such an MLC write depends on the resistance distributions over a large ensemble of PCM cells. Unlike SLC write, where the bit write quality can be ensured by over-SET or over-RESET, the intrinsic randomness associated with each write attempt and the inter-cell variability make it impractical to have a universal pulse shape for writing an intermediate state. In order to address this issue, resistance distribution tightening techniques have been developed based on the program-and-verify (P&V) technique [51]. P&V is a common programming technique for multi-bit writing and is widely used in MLC NAND flash products [66] and MLC PCM prototypes [64, 1]. In order to achieve non-overlapping resistance distributions of different bit levels, P&V needs to iteratively apply partial set pulses and then verify that a specified precision criterion is met, which leads to much longer write latency and hence the much larger programming energy. Similar to NAND flash, the MLC PCM programming energy can be more than 10 times of the SLC one.

Table 3.1. Write energy of programming every state [1]

State	Average current	Pulse duration	Average energy consumption
00	$300\mu A$	$40ns$	$36pJ$
01	$100\mu A$ - $548\mu A$	$220ns$ - $520ns$	$307pJ$
10	$576\mu A$ - $996\mu A$	$220ns$ - $520ns$	$547pJ$
11	$200\mu A$	$150ns$	$20pJ$

In this work, we evaluate 2-bit MLC PCM, of which four states can be stored within each cell. Bit patterns "00" and "11" are stored using complete RESET and complete SET state, respectively. Intermediate resistance states in the order of increasing resistance are used to represent combinations "10" and "01". Fig. 3.1 demonstrates the concept of the P&V programming technique for MLC PCM. Instead of applying a full SET-sweep pulse to securely set the cell to SET state or applying a powerful RESET pulse to set the cell to RESET state, intermediate MLC states have to be programmed in an iterative manner. In this work, we refer to an MLC PCM prototype design from Bedeschi *et al.* [1], and we use the SET-sweep pulse with $0.2mA$ peak current and $150ns$ duration and the RESET pulse with $0.3mA$ amplitude and $40ns$ duration. For the partial SET pulses, we assume that there are 32 possible P&V steps in total. To program intermediate states, the cell has to experience a full SET-sweep pulse and a full RESET pulse as the initialization sequence to improve the programming quality of following steps [1]. After that, a sequence of partial SET pulses with $15ns$ duration are applied until the intermediate resistance level is reached. For programming intermediate state "10", the the partial SET pulse amplitude starts from $0.1mA$ with $28\mu A$ stepping; for programming "01", the partial SET pulse amplitude starts from $0.576mA$ with $28\mu A$ stepping[2].

Based on our energy model assumption, the energy consumption values of programming every state are listed in Table 3.1. It should be noticed that process variation affects the MLC PCM behavior and the number of P&V rounds is different from cells to cells. Thus, the energy estimations in Table 3.1 are all average values.

[2]These parameters are scaled from an MLC PCM prototype with $90nm$ process node and bipolar-selected cells [1].

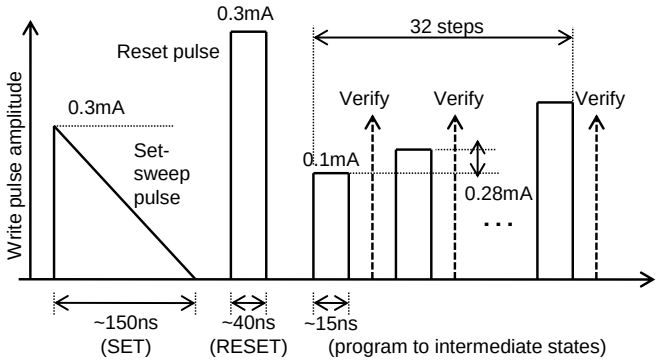

Figure 3.1. The pulse shapes of complete SET and RESET operations and the modeled program-and-verify scheme by using partial SET pulses.

From Table 3.1, we can see that there are significant value-dependent programming energy variations in multi-level cell PCM. Programming RESET state "00" using full RESET pulse and SET state "11" using SET-sweep pulse need one order of magnitude less energy than programming other states. Thus, this phenomenon motivates us to propose a new MLC PCM architecture based on data encoding to increase the "11" and "00" states in the writing data and thereby reduce the programming energy.

3.2 Implementation

In this section, the MLC PCM data encoding architecture is described. At first, we propose a data encoding algorithm that maximizes the frequencies of the "11" (i.e. full SET) and the "00" (i.e. full RESET) stored in MLC PCM. Then, the advantage and overhead of this algorithm are analyzed. In the end, the circuit architecture of such energy-efficient MLC PCM data encoding is discussed.

3.2.1 Data Encoding Algorithm

The following terms are defined for the discussion of the data encoding algorithm:

- *Low Power States (LPS)*: the states which need less energy in programming, representing states "00" and "11" in 2-bit MLC PCM;

Algorithm 1 Data Encoding Algorithm

```
// N: memory line width
Write(A: address, D: data)
MT := Mapping Type (D); // get a mapping type
D_m := Map(D, MT); // map the data
for all i = 0, i ≤ N, i++ do
  write MLCs as D_m
end for
```

- *High Power States (HPS)*: the states which need more energy in programming, representing states "01" and "10" in 2-bit MLC PCM;
- *Original Data*: the data that are not encoded;
- *Encoded Data*: the data that are encoded according to the data encoding algorithm and are stored in PCM in an encoded form.

Basically, the key concept of the algorithm is to increase the total percentage of LPS in writing data by encoding the input data to ensure that the two most frequent states are mapped to "11" and "00". Therefore, the total writing energy can be reduced since programming "11" (i.e. full SET) and "00" (i.e. full RESET) states need less energy. Algorithm 1 captures the process of such data encoding.

In the encoding algorithm, the selection of *Mapping Type* is the critical part. For the special feature of MLC PCM programming, the mapping rules for this algorithm are designed as follows:

- *Mapping Rule 1*: Map the two most frequent states to "11" and "00".
- *Mapping Rule 2*: Maintain the original data in the encoding operation as much as possible.

Rule 1 ensures that the states "11" and "00" are the tow most frequent states in the encoded data. Rule 2 is devised to reserve the effectiveness of data comparison write (DCW), which we discuss later in Section 3.3.

According to the mapping rules, there are $C_4^2 = 6$ different ways of mapping the four states. A look-up table is used to implement the data encoding algorithm, as shown in Table 3.2.

Table 3.2. Encoding look-up table

Mapping of two most frequent states		Mapping of the other two states		Mapping type
00→00	11→11	01→01	10→10	0000
00→00	01→11	10→10	11→01	0001
00→00	10→11	01→01	11→10	0011
01→00	10→11	00→10	11→01	1100
01→00	11→11	00→01	10→10	1101
10→00	11→11	01→01	00→10	1111

For every memory line, the information of *Mapping Type* needs to be added to decode the encoded data. This mapping type needs 4 extra bits or 2 extra cells to present since there are 6 different mapping types and every memory cell has two bits. Because mapping types also need to be written to MLC PCM cells, we use as much as states of "00" and "11" to present the mapping type, as shown in Table 3.2.

The control and data flow of the encoder and decoder is shown as Fig. 3.2. For every memory line, the original data is stored in a data buffer. Then, the percentage of every state is counted by the encoder to choose the mapping type using the mapping lookup table. For example, if the two largest frequency states of one memory line data are "01" and "11", the encoder choose mapping type as "1101" according to the mapping lookup table. Then, the encoded data is written as the rules: "00", "01", "10", "11" are encoded to "01", "00", "10" and "11", respectively. Thereby, the states "11" and "00" are the two most frequent states in the encoded data.

3.2.2 Advantage and Overhead of Data Encoding Algorithm

For the original data, assume there are N 2-bit cells in a memory line and the probability of an MLC is LPS (i.e. "00" or "11") is 1/2. Therefore, the probability of having i LPS among the original data is $(1/2)^N C_N^i$ based on the binomial distribution. So the average percentage of LPS in N cells is:

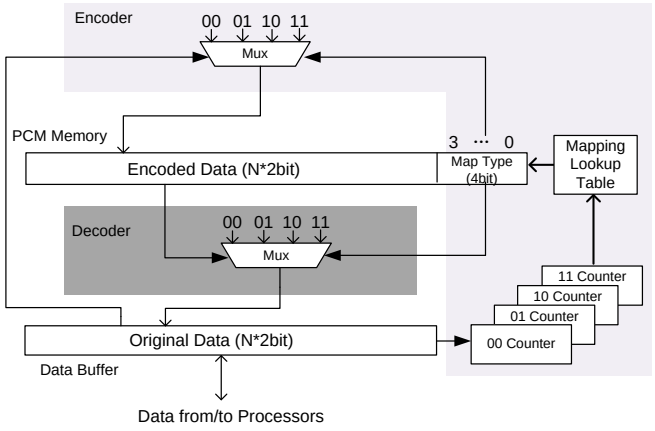

Figure 3.2. The control and data flow of encoder and decoder.

$$P_{LPS} = \sum_{i=0}^{N} \frac{i}{N} \frac{1}{2^N} C_N^i = \frac{1}{2} \tag{3.1}$$

Similarly, the average percentage of LPS in N cells after encoding is calculated as follows:

$$P'_{LPS} = \sum_{i=0}^{N/2} \frac{(N-i)}{N} \frac{1}{2^N} C_N^i + \sum_{i=N/2+1}^{N} \frac{i}{N} \frac{1}{2^N} C_N^i \tag{3.2}$$

In Equation 3.2, the result is split into two cases, of which the first part is mapping the HPS to LPS case since the LPS percentage is smaller ($i < N/2$) and the percentage of LPS is changed to $(N-i)/N$ after encoding. Therefore, we define the *data encoding algorithm's advantage ratio*, R, as:

$$R = \frac{P'_{LPS} - P_{LPS}}{P_{LPS}} = \sum_{i=0}^{N/2} \frac{(N-2i)}{N} \frac{1}{2^{N-1}} C_N^i \tag{3.3}$$

Compared with the original data, the percentage of LPS in the encoded data is increased by 37% (for a 4-bit memory line) or by 5% for a 64-byte memory line. The algorithm's advantage ratio is different depending on the width of memory lines as shown in Fig. 3.3. Moreover, this relative advantage is based on the assumption of random multi-bit value distribution. With realistic application, the benefits are different depending on different application workloads, which are shown in

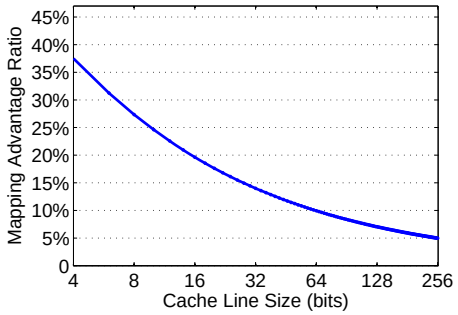

Figure 3.3. The data encoding algorithm's advantage ratio (which indicates how many extra LPS have been added) versus the width of memory lines.

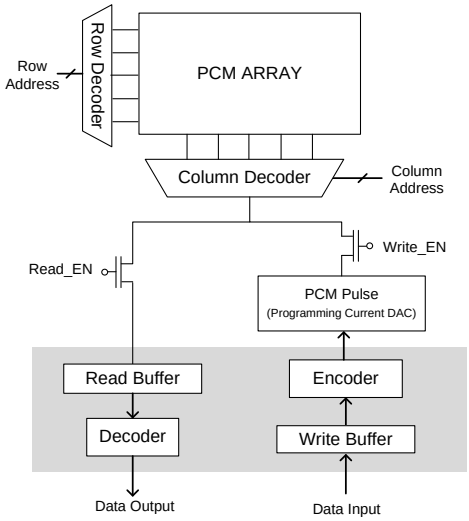

Figure 3.4. PCM memory architecture including the extra components for data encoding and decoding.

Section 3.4.

Moreover, this encoding algorithm is a fix-length encoding, which can ease the memory accessing management. As shown in Fig. 3.2, the encoded data is the mapped data in addition of the mapping type. Therefore, if the width of original data is $N \times 2$ bits, the width of encoded data is $N \times 2 + 4$ bits, which equals to the width of one memory line in our design. For a 64-byte-per-line MLC PCM, the overhead of the data size is about 1.5%.

3.2.3 Data Encoding Hardware

The traditional memory structure needs to be modified in order to apply the data management function efficiently. A new architecture of MLC PCM is designed which has the capability to encode the input data before writing, and decode output data after reading. Therefore, write buffer, read buffer, encoder component, and decoder component are all required in this system. As shown in Fig. 3.4, the components in gray are the extra hardware components that are required by the purposed MLC PCM data encoding architecture.

The data access pattern in such an architecture is described as follows:

- *Write Operation*: Store a memory line in the write buffer before writing to the PCM. When data in buffer is ready, the encoder transforms the input data to the encoded data which needs less writing energy. Then, the programming component writes the result to PCM. Usually, processors' speed is much faster than writing to PCM's MLC, which requires about 270ns in the worst case according to our estimations as listed in Table 3.1. Therefore, processor just need to store the memory line data to write buffer in this architecture and *Direct Memory Access* (DMA) makes the performance overhead negligible.

- *Read Operation*: When a memory line is read out, it is stored in the read buffer at first. Then decoder maps the encoded data to original data with the mapping type information, which is the least significant 4 bits of the memory line data.

3.3 Combine Data Encoding with DCW

Data comparison writes (DCW) is a common scheme for PCM system, which has been implemented in some PCM prototypes [71]. Therefore, in this section we study how to use the proposed encoding architecture in MLC PCM system with adopting DCW scheme.

3.3.1 Introduction of DCW

DCW technique [45] is designed to remove the redundant bit writes. For MLC-2, the statistical bit-write redundancy is 25% if writing every state is equally likely.

The basic concept of DCW is preceding a write with a read. Write the cell only if the data is changed after writing. It will reduce the unnecessary write operations, which improve the endurance of PCM, and reduce the writing energy consumption. In PCM operations, reads are much faster and consume much less energy than writes. It is this asymmetry can we benefit from to improve the PCM endurance and reduce the write energy.

3.3.2 Modified Data Encoding Algorithm for DCW

When DCW is adopted in this memory system, the question is how to combine the data encoding and DCW techniques efficiently. We propose two techniques to solve this problem: *Maximum Maintained Mapping* and *Energy Hamming Distance Comparison.*

- *Maximum Maintained Mapping:*

In the encoding algorithm, Rule 2 is to maintain the original data in the encoding operation as much as possible. It ensures that most of the data in the mapping will not be changed. From the encoding table, we can see that the Hamming distance between original data and encoded data is $N/2$ if writing every state is equally likely.

- *Energy Hamming Distance (EHD) Comparison:*

When encode one memory line, we can choose from two mapping types: the old mapping type which is read from the old memory data and the new mapping type which is calculate from the new data. The more similar the new data and old data are, the more efficient DCW is. In this case, the old mapping type should be chosen to save more energy using DCW. Otherwise we should choose the new mapping type to save energy through data encoding. To decide which case every data line belongs, we define Energy Hamming Distance (EHD) to quantify this problem. EHD is the Hamming Distance between two data with the weights which are set

Algorithm 2 Combining Data Encoding with DCW

```
// N: memory line width
Write(A: address, D: data)
D_old_m(old), MT_old := Read(A)
// Map new data as old mapping type
D_new_m(old) := Map(D, MT_old);
// Get the new mapping type
MT := Mapping Type (D);
// Map new data as new mapping type
D_new_m(new) := Map(D, MT);
if EHD(D_old_m(old), D_old_m(old)) ≤ EHD(D_new_m(new), D_old_m(old)) then
   MT := MT_old; // Keep the old mapping type
end if
for all i = 0, i ≤ N, i++ do
   if Map(D, MT) ≠ D_old_m(old) then
      update MLCs to Map(D, MT)
   end if
end for
```

as the states' writing energy. Before writing the data, calculate the EHD between the old data and the new data encoded as the old mapping type, as well as the old data and the new data encoded as the new mapping type. Then the one which has smaller EHD is chosen to write to memory through DCW scheme, so that the effectiveness of encoding and DCW both can be maximally reserved. Algorithm 2 captures the scheme combing data encoding and DCW.

3.3.3 Modified Architecture of MLC PCM for Combining Data Management and DCW

Some modifications are needed to implement on the PCM architecture to combine data encoding management and DCW. The modified PCM memory read/write data path architecture is shown in Fig. 3.5, which is based on Fig. 3.4. The gray part in Fig. 3.5 is added for DCW and the other part is the same as the architecture in Fig. 3.4. The *Diff* and *EHD Count* are used to calculate the Energy Hamming Distance of two input data. Depending on which distance is smaller, the mapping type with smaller EHD are selected to save more energy. It should be noticed that the only difference of these two choices is to use the new mapping type or the old

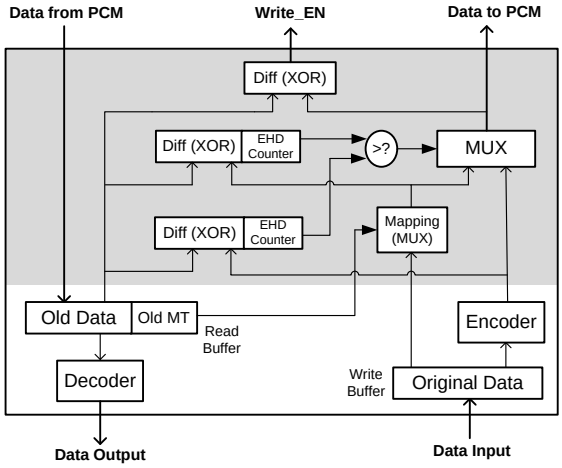

Figure 3.5. Modified PCM memory read/write data path architecture combining the data encoding components and DCW.

one which is stored in PCM. As a result, the decoder part does not need to change since it can decode all the data as encoded data using mapping type information.

The performance overhead of adding this part is small because the gates and reading operation is much smaller than the MLC PCM write latency. On the other hand, only several XOR gates, counters, and multiplexers are needed and only one block is added in PCM data path. Thus, the overhead of hardware is also negligible. We discuss more details about the hardware overhead in Section 3.4.

3.4 Experimental Results

In this section, we evaluate the energy improvement after applying the data encoding based MLC PCM architecture.

3.4.1 Experiment Methodology

Our experiment simulates a 3.2GHz chip-multiprocessor with four SPARCV9-like cores. Each core has their private 2-way-associative I-L1 and D-L1 caches with the identical capacity of 64KB. The 16-way-associative L2 cache is shared by all the cores and has a capacity of 4MB. The size of write buffer before PCM memory is set to 128 entries. The write buffer is large enough to mitigate the impact of long write latency of PCM memory. For all the benchmarks, the write buffer is

Table 3.3. Baseline configurations

Processor	4-core, 3.2GHz, SPARCV9-like cores
I-L1/D-L1 caches	private, 64KB/64KB, 2-way, 64-Byte cache line
L2 cache	shared, 4MB, 16-way, 64-Byte cache line
PCM module	4GB, 128-entry write buffer

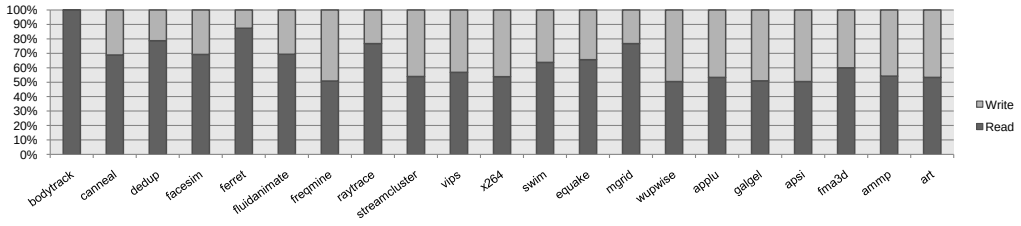

Figure 3.6. Ratio of read and write memory access times

never full so that the read operation will not be blocked for a long time if there is a burst of write operations. The parameters of baseline configuration are listed in Table 3.3.

For our 4-core system, our experimental evaluation makes use of 4-thread OpenMP version of workloads from PARSEC 2.1 [72] and SPEC OMP 3.2 [73] benchmark suites. The input size of the SPEC OMP benchmark is medium, and the native inputs are used for the PARSEC benchmark to generate realistic program behavior. We exclude 2 workloads from PARSEC and 1 workload from SPEC OMP[3], hence we evaluate 23 workloads in total. We collect the memory accesses to the PCM module by using the Simics full-system simulator [74]. Each Simics simulation run is fast forwarded to the pre-defined breakpoint at the code region of interest, warmed-up by 100 million instructions, and then simulated in the detailed timing mode for 1 billion cycles.

3.4.2 Hardware Overhead

We evaluate the hardware overhead of the proposed encoding with DCW by using Design Compiler to estimate the area and energy consumption with $45nm$ TSMC CMOS library. According to the area analysis, the proposed encoder and decoder architecture incurs extra area of $0.025mm^2$, which is negligible compared to the

[3] *blackscholes* and *swaptions* are excluded because these two workloads generate too few memory access traffic; *gafort* causes segmentation fault when executed in the parallel.

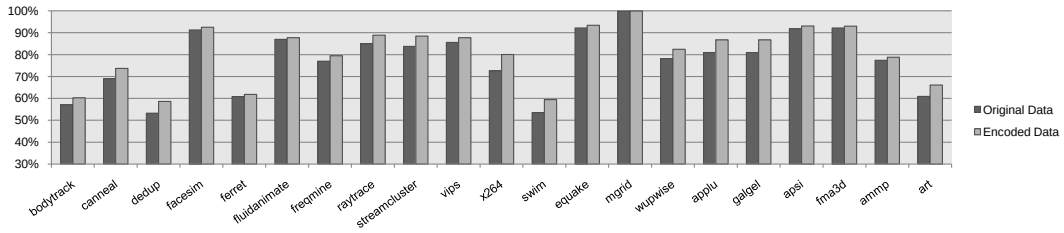

Figure 3.7. Percentage of LPS in original data and encoded data.

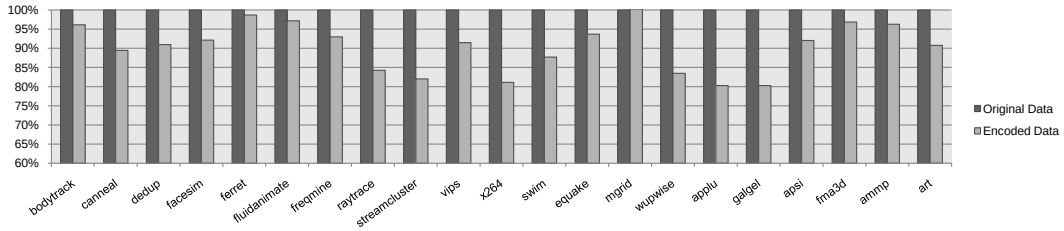

Figure 3.8. Reduction of energy consumption after encoding.

area of PCM system. According to the energy analysis, the encoder and decoder incur extra energy consumption of $0.971pJ$ and $0.449pJ$ per memory line access, respectively. For different workloads, the read and write access numbers are different, which are show in Fig. 3.6. These results are used in our simulation to calculate the encoder and decoder's energy overhead.

3.4.3 Evaluation of Data Encoding Algorithm

Fig. 3.7 shows the result of the LPS's percentage in the data. It shows that the LPS's percentage is increased by 4.6% on average (up to 11.1%). It can be seen that if the percentage of LPS in the workloads' original data is smaller, the advantage of the encoding technology is larger. It is because the improvement of LPS's percentage is limited when it is already large enough in the original data. Therefore, for *mgrid* from Fig. 3.7, the percentage of LPS is 99.9% in original data and is increased only 0.1% after encoding; for *swim*, the percentage of LPS is 53.5% in original data and is increased as much as 11.1% after encoding.

Fig. 3.8 compares the results of energy consumption for different applications with original data and encoded data, which include the energy of writing and reading, respectively. The energy consumption of writing every different state is

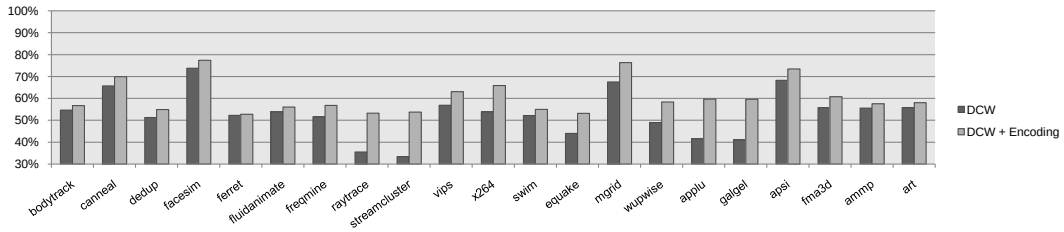

Figure 3.9. Percentage of LPS in original data and encoded data after the DCW technique is adopted.

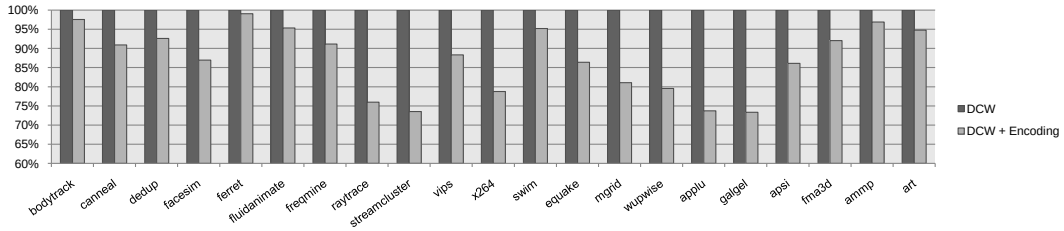

Figure 3.10. Reduction of energy consumption after the combination of data encoding and DCW.

listed in Table 3.1. Besides, we also consider the energy of encoder and decoder for every write and read access. Fig. 3.8 shows that the total energy consumption is reduced by 9.6% on average of different workloads (up to 19.8%). Basically, if the increasing of LPS's percentage is larger, more energy is saved in the workload. However, if the number of memory read accesses is too large relative to that of write accesses, the overhead of decoder's energy will influence the effectiveness of the encoding technique since the decoder consumes energy for every reading access. From Fig. 3.7 and Fig. 3.8, we can see that for workload *swim* and *galgel*, the improvement of LPS's percentage is 11.1% and 7.1%. But, *galgel* saves 19.8% energy, which is more than *swim* of 12.3%, because decoder energy overhead of *galgel* is smaller since its ratio of write to read is larger than *swim*, which can be seen from Fig. 3.6.

3.4.4 Evaluation of Combining Data Encoding and DCW

The benefits of combining data coding and DCW together are also evaluated. Fig. 3.9 shows the result of percentage of LPS in the writing data. In the first

case, only the DCW technology is used. In the second case, encoding is used in together with DCW. The result shows that the percentage of LPS is increased by 16.8% on average (up to 60.9%). It can be noticed that generally the improvement of LPS's percentage is larger than the memory system without DCW. The reason is DCW technique eliminate the unnecessary writes, and most of them is writing state "00" repeatedly. So the percentage of LPS in the workloads' original data is smaller for the memory system with adopting DCW. Therefore, the effectiveness of data encoding becomes larger.

Fig. 3.10 shows that the total writing and reading energy of two different configurations: original data using DCW, encoded data using DCW. After adding the DCW scheme, every write access includes one read access. The energy consumption of these reading operations are also calculated in the simulation. Moreover, the energy overhead of the encoder and decoder components are also considered. The result shows that the total energy of writing and reading using the proposed architecture is reduced by 12.9% on average (up to 26.7%) compared to the memory system which only uses DCW.

It should be noticed that adopting data encoding saves much more energy in the memory system with DCW than the one without DCW for some workloads, such as *streamcluster*. There are two reasons: the first one is that LPS's percentage of original writing data is decreased from 83.8% to 33.4% after DCW scheme, so that data encoding algorithm can map much more writing data from HPS to LPS; the second reason is the ratio of read to write in *streamcluster* is small which is 1.17, thus the reading energy overhead is small. On the other hand, for some other workloads, such as *swim*, adopting data encoding saves less energy in the system with DCW than the one without DCW. Because of the character of the writing data in *swim*, DCW cannot decrease the LPS's percentage in original data too much, just from 53.5% to 52.2%. Moreover, the energy overhead is large which comes from two aspects: the decoder's energy for every reading access and the reading energy overhead in DCW since every write access includes one read access.

These simulation results shows that the proposed encoding architecture is effective to the MLC PCM system without or with adopting the DCW technique.

3.5 Summary

PCM is considered as one of the most promising technologies among emerging non-volatile memories. Besides the common single-level cell (SLC) technology, recent PCM prototypic chips demonstrate that multi-level cell (MLC) is practical. Compared to SLC, MLC can store more than 1 bits on every PCM cell, thus it is one of the attractive properties of PCM that helps to achieve higher storage density. However, programming MLC PCM involves the program-and-verify scheme and incurs much more energy consumption. In this work, we propose a new MLC PCM architecture using data encoding before writing to implement an energy-efficient MLC PCM system based on the observation that there are significant value-dependent energy variations in programming MLC PCM. In addition, we adopt data comparison write (DCW) to enhance the effectiveness of the proposed data encoding architecture. The experimental results show that the total energy consumption of writing and reading using the proposed architecture is reduced by 9.64% on average (up to 19.8%) on plain MLC PCM systems, and by 12.9% on average (up to 26.7%) on DCW-adopted MLC PCM systems.

Chapter 4

NVM Main Memory: Small Page Size

Besides the issue of high write energy we discussed in Chapter 3, how to improve the performance and how to design compatible interfaces are still challenges for building NVM main memory. In this chapter, we use STT-RAM as an example to show our efforts on this topic.[1]

STT-RAM is another attractive NVM technology to implement main memory systems, and its power saving opportunities have been heavily exploited [53, 55, 54]. However, enjoying the STT-RAM power-saving benefit is not free. It is a consensus that STT-RAM cannot compete with DRAM in performance, and more importantly, STT-RAM chip internal structure is not compatible with DRAM interfaces. This compatibility is not only highly required for a successful STT-RAM early adoption, but also critical to enable a tiered memory system using refresh-free STT-RAM and high-performance DRAM [31, 5, 32]. Everspin's effort [33] to implement a DDR3-compatible STT-RAM is an example to meet such request.

The industry has realized the internal structure difference between NVM and DRAM, and the LPDDR2 standard once includes an NVM-based solution called LPDDR2-NVM [75], in which the activation command is divided to Pre-Active and Activate to transfer the long row address. However, it needs a dedicated controller for 3-phase access and a new software interface with much higher design

[1]This work is published as "Enabling High-Performance LPDDRx-Compatible MRAM" on ISLPED2014.

complexity. Due to the design complexity, this LPDDR2-NVM spec was removed from LPDDR3 standard. How to design LPDDR3-compatible NVM architecture remains to be a challenge.

In this chapter, we study STT-RAM's unique features that are different from DRAM. In particularly, three unique STT-RAM properties, including *small page size*, *non-destructive read*, and *independent write path*, are investigated, and we propose four techniques to enable the design of LPDDR3-compatible high performance and low power STT-RAM architecture [76]:

- We keep the LPDDRx interface and propose *Combinational Row/Column Address Strobe* (**ComboAS**) to handle the unbalanced row and column address bits caused by the STT-RAM's *small page size*;
- We add *Dynamic Latency* (**DynLat**) to alleviate the performance degradation introduced by ComboAS;
- We devise *Early Precharge/Activation* (**EarlyPA**) to improve the STT-RAM performance by utilizing the STT-RAM *non-destructive read* property;
- We improve the write performance by using *Buffered Write* (**BufW**) technique, which buffers multiple writes and leverages the STT-RAM's unique property of *separate write path*.

Combined these all together, we come up with a DRAM-swappable STT-RAM solution with a significant performance improvement. In another word, our techniques enable an LPDDRx-compatible STT-RAM with DRAM-competitive performance and freely exploit the STT-RAM low power features.

4.1 Background

In this section, we describe the architecture of traditional LPDDRx devices and the architecture of STT-RAM LPDDRx devices.

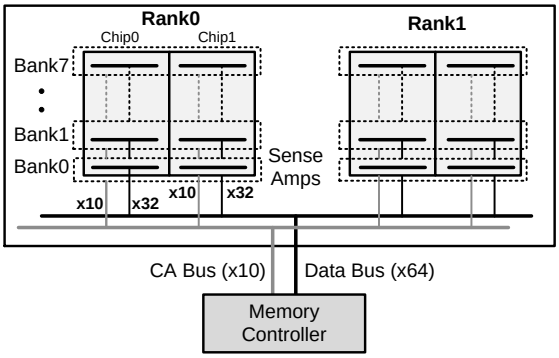

Figure 4.1. The memory organization for LPDDRx chips.

4.1.1 Architecture of Traditional LPDDRx Devices

LPDDRx is the dominant memory interface for modern mobile devices. JEDEC released the first LPDDR specification in 2009. Today, almost all the mobile DRAM chips use LPDDR2 or LPDDR3 interface [77].

Figure 4.1 is an exemplary LPDDRx memory subsystem. LPDDRx devices have wider I/O (e.g. x16 or x32) than DDRx ones (e.g. x4 or x8). In our example, four LPDDRx devices form a 2-rank memory subsystem with a 64-bit data bus. LPDDRx uses a multiplexed command/address (CA) bus to reduce the pin count. The 10-bit CA bus contains command, address, and bank information.

Each LPDDRx device internally has 8 banks, and each bank can independent process a different memory request. The internal LPDDR3 datapath uses an 8n prefetch architecture (LPDDR2-S2 is 2n prefetch, and LPDDR2-S4 is 4n prefetch). The LPDDRx interface transfers 2 data bits per DQ pin during every clock period.

Same as DDRx, LPDDRx accesses begin with an activation command (ACT), which includes an RAS (row access strobe) signal, a bank address, and a row address. Memory controllers send ACT commands to memory devices, and memory devices activate the corresponding bank and the row. The data from the activated row is latched in the sense amplifier (S/A) after a tRCD delay (row address to column address delay). Then, memory controllers can continue to issue column read or write commands with a CAS (column access strobe) signal and the starting column address for the burst access.

The S/A acts as a temporary data storage and drives the amplified data until the array is pre-charged again. Therefore, the S/A is essentially a row buffer that

caches the entire row data (which can be 1-4 KB in modern DRAMs, and it is called "a page"). Each memory bank has its own S/A.

4.1.2 Architecture of STT-RAM LPDDRx Devices

We can use the same memory organization in Figure 4.1 for STT-RAM devices. We simulate two 4Gb LPDDR3 modules with DRAM and STT-RAM, respectively, on a 28nm DRAM process node. Table 4.1 lists the parameters. We use our modified-version of CACTI [78] and NVSim [79] to generate this estimation, and we can verify that the estimated numbers match to the actual LPDDR3 DRAM and STT-RAM prototypes. The major parameter differences between DRAM and STT-RAM include:

- *Small page size*: STT-RAM uses current sensing, which is generally more complex than DRAM voltage sensing and significantly bigger. To get the similar area, STT-RAM reduces number of S/A and thus has a smaller page size. Our circuit simulation shows the page size of a 4Gb LPDDR3 STT-RAM device is only 256B, 16X smaller than its DRAM counterpart. STT-RAM industry companies also have consensus on this. For example, a 2012 EverSpin patent [80] discloses that their STT-RAM page is only 512-bit large, 32X smaller than a DRAM page.

- *Non-volatility*: STT-RAM is non-volatile and needs no refresh. Hence, both the tREF and tRFC of STT-RAM are zeros. STT-RAM auto-refresh current (IDD5[2]) and self-refresh current (IDD6) are zeros as well.

- *Non-destructive read*: STT-RAM has smaller tRTP and can issue precharge command sooner because STT-RAM reads are non-destructive and do not need write-back.

- *Fast page close*: STT-RAM has faster precharge speed and smaller tRP because DRAM precharge needs to balance the bitlines (BL and $\overline{\text{BL}}$) to VDD/2, but STT-RAM precharge can skip this step.

[2]In auto-refresh mode, STT-RAM peripheral circuitry still consumes power so that IDD5 is essentially IDD2P.

Table 4.1. Comparison of DRAM and STT-RAM LPDDR3 devices.

Parameters	LPDDR3 DRAM		LPDDR3 STT-RAM	
Clock	533 MHz		533 MHz	
Page size	4 KByte		256 Byte	
Bank bit	BA2-BA0		BA2-BA0	
Row bit	R13-R0		R17-R0	
Column bit	C9-C0		C5-C0	
tREF	3900 ns		N/A	
tRCD	10 cycle		13 cycle	
tRL	8 cycle		6 cycle	
tWL	4 cycle		4 cycle	
tRP	10 cycle		7 cycle	
tRC	32 cycle		18 cycle	
tRTP	4 cycle		2 cycle	
tRRD	6 cycle		6 cycle	
tCCD	4 cycle		4 cycle	
tWTR	4 cycle		4 cycle	
tWR	8 cycle		14 cycle	
tFAW	27 cycle		27 cycle	
tRFC	70 cycle		N/A	
Voltage	VDD1 (1.8 V)	VDD2 (1.2 V)	VDD1 (1.8 V)	VDD2 (1.2 V)
IDD0	7.8 mA	28.3 mA	4.8 mA	41.2 mA
IDD2N	1.8 mA	3.7 mA	1.8 mA	3.7 mA
IDD2P	1.8 mA	1.7 mA	1.8 mA	1.7 mA
IDD3N	3.5 mA	6.9 mA	3.5 mA	6.9 mA
IDD3P	3.5 mA	6.9 mA	3.5 mA	6.9 mA
IDD4R	3.5 mA	140.4 mA	0 mA	170.9 mA
IDD4W	3.5 mA	145.4 mA	0 mA	267.2 mA
IDD5	23.8 mA	78.2 mA	1.8 mA	1.7 mA
IDD6	1.0 mA	3.8 mA	0.0 mA	0.0 mA

- *Slow page open*: STT-RAM MTJ has smaller on/off resistance ratio (e.g. 200%), and it is hard to sense the data. Therefore, the STT-RAM row activation speed is slower, and tRCD of STT-RAM is larger.

- *Slow write*: STT-RAM has longer write latency and higher write energy. Thus, STT-RAM has larger tWR and IDD4W.

4.2 Technique 1: Combinational Row/Column Address Strobe (ComboAS)

In this section, we propose the first technique, Combinational Row/Column Address Strobe, and we introduce its motivation, operation and implementation.

4.2.1 Motivation: Balance Row/Column Address Transfers

LPDDR2 and LPDDR3 uses multiplexed command and address bus (CA bus). Each command occupies the CA bus for one cycle and is clocked at both positive and negative edges. CA bus is 10-bit width and hence each command contains 20 bits in total. The activation command (ACT) uses 2 bits for command decoding, 3 bits of the bank address, and the remaining 15 bits are for the row address. Therefore, this scheme can address up to 32K row. It works well for existing DRAM devices but not for our targeted STT-RAM devices.

The major problem is caused by the STT-RAM small page size. As explained in Section 4.1.2, as STT-RAM S/A is much more complex and larger, the number of S/A per bank is correspondingly reduced. Compared to DRAM devices usually equipping 1KB-4KB pages, STT-RAM page size is much smaller. For example, EverSpin's STT-RAM page size is 512 bits [81]. This is also a common drawback for other non-volatile memories that use current sensing (e.g. Micron's 1Gb PCM only has page size of 512 bits [82]). In this work, our simulated 4Gb STT-RAM device page size is 256B, 16 times smaller than its DRAM counterpart (see Table 4.1). Since the total memory capacity and the bank count are the same, STT-RAM needs 4 additional row address bits than DRAM does. While it is not a problem for low-density STT-RAM devices (e.g. EverSpin 64Mb STT-RAM [33] with 64B page can still be DDR3 interface-compatible), the existing LPDDRx interface is not compatible with gigascale STT-RAM devices.

A naive solution to this problem is to add two more CA bus pins, but we do not consider it as an option. First, the row and column address bits are highly unbalanced, and the extra two pins are only useful in ACT commands. It is a contradiction to the basic multiplexing concept behind the CA bus design. Second,

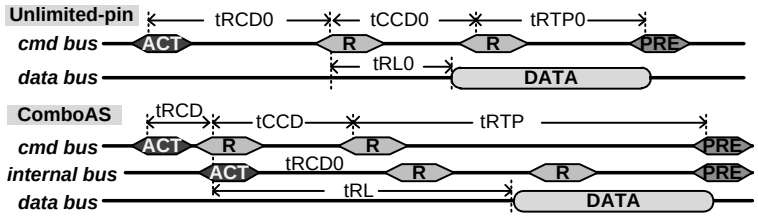

Figure 4.2. The timing diagrams of Unlimited-pin and ComboAS.

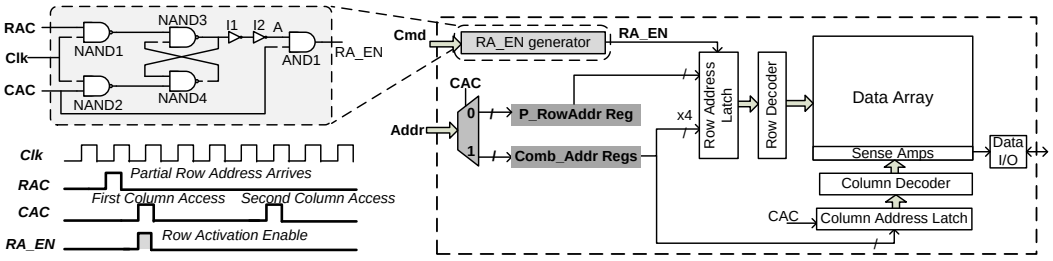

Figure 4.3. The architecture of ComboAS. The zoom-in figure shows the detailed circuit design of the RA_EN generator.

adding two pins implies the STT-RAM LPDDRx bus is not compatible to the existing DRAM bus. The bad consequences include but not limited to: (1) an industry-wise pin ball redesigns and PHY interface redesigns; (2) for the host that tends to mix DRAM and STT-RAM, two different memory interfaces are required thereby increasing both area and cost. Neither of them is good for the STT-RAM early adoption. Instead, our goal is to make a DRAM-swappable STT-RAM solution with a fully LPDDRx-compatible interface.

4.2.2 ComboAS Operation

We propose *Combinational Row/Column Address Strobe* (ComboAS) to balance the STT-RAM long row address and short column address caused by its smaller page size. The basic concept of *ComboAS* is to offload the overflowed row address from RAS commands (i.e. ACT) to CAS commands (i.e. READ or WRITE). Hence, RAS commands only carry parts of the row address, and we transfer the remaining row address together with the column address in CAS commands.

Considering now we split the row address into RAS and CAS commands, we need both of them before a new row activation. Consequently, instead of waiting for tRCD, we should issue a CAS command immediately after every RAS command.

Table 4.2. Timing parameters for unlimited-pin and ComboAS.

	Unlimited-pin system	ComboAS
tRCD	$tRCD_0$	1 tCK
tCCD	$tCCD_0$	$tCCD_0$
tRL	tRL_0	$tRCD_0 + tRL_0$
tWL	tWL_0	$tRCD_0 + tWL_0$
tRTP	$tRTP_0$	$tRCD_0 + tRTP_0$

Figure 4.2 compares the timing diagram of ComboAS against the incompatible solution of adding 2 more CA pins.

The ComboAS timing is similar to posted-CAS [83] and posted-RAS [24] as they all issue RAS and CAS commands back-to-back, but they are essentially different. The actual row activation in posted-CAS [83] starts before CAS is received, and its CAS command does not carry any row information. The posted-RAS scheme proposed by Udipi *et al.* [24] is the most similar work to our ComboAS technique. However, posted-RAS only works for close-page policy where there is only one CAS command after opening one row, and it does not optimize the timing parameters to mitigate the performance overhead.

To make ComboAS work for both close- and open-page policies, we adjust memory timing parameters as follows:

- **Minimize tRCD:** Because we now wait for CAS to start a row activation, and there is no circuit dependency between RAS and CAS, we can reduce tRCD to 1 clock cycle.

- **Adjust tRL, tWL, and tRTP:** In ComboAS, the arrival of CAS only means the start of a row activation. It delays the actual column access (read or write) by the physical row activation time. Therefore, we need to increment both the column read and write latencies (tRL and tWL) by a row activation delay. Similarly, it is necessary to adjust tRTP (read to precharge delay) in the same way.

Figure 4.2 compares the ComboAS external and internal command buses. The actual row activation in the memory is delayed by 1 cycle to wait for the remaining row address bits carried by CAS, and the read/write accesses are delayed by $tRCD_0$ (the original row activation delay). Table 4.2 lists the detailed adjustments.

4.2.3 ComboAS Implementation

To implement ComboAS, the key issue is to let the memory chip wait till the first CAS command arrives before activating the accessing row. Our modifications include: a register to store the partial row address in RAS; a set of registers to hold the column addresses in the early-arrived CAS commands; a signal generator to latch the remaining row address from the first-arrived CAS command.

Address Registers: Normally, we only require one address register because the row address and column address are completely separate. However, as ComboAS divides row addresses into both RAS and CAS, we need two registers as shown in Figure 4.3: *P_RowAddrReg* stores the partial row address in RAS commands; *Comb_AddrRegs* hold the combination of the remaining row address and the column address in CAS commands.

In Figure 4.3, RAC and CAC are the signals to represent the arrivals of RAS and CAS, respectively. They can be decoded from the LPDDRx command truth table. Since RAS and CAS share the same CA bus, we add a multiplexer to choose from these two registers with CAC as the select signal.

Note that we design *Comb_AddrRegs* to be capable of holding more than one entries since multiple CAS commands can arrive to the memory chip before the requested row is opened as the latency to open a row ($tRCD_0$) is larger than the minimum delay between two column commands ($tCCD_0$). We decide the number of column address registers by $\lceil tRCD_0/tCCD_0 \rceil$ (i.e. 3 in this work). Similar to the traditional posted-CAS DRAM, ComboAS uses countdown circuits to delay the external CAS command by $tRCD_0$.

RA_EN Generator: Normal memory devices use the RAC signal to trigger row activations. However, in ComboAS, only the first-arrived CAC signal shall trigger this step; all the latter CAC signals should be filtered from the row activation control. We design a *RA_EN generator* for this purpose as shown in Figure 4.3. In *RA_EN generator*, the signal A is 0 when RAC and CAC are both 0. When RAS arrives and RAC becomes 1, A changes to 1; when the first CAS command comes and CAC becomes to 1, A toggles to 0 and remains unchanged when latter CAS commands arrive. Thus, we ensure only the first CAS triggers the row activation.

Memory Controller: The modification to the memory controller is negligible in ComboAS. Only a small latch (4-bit in this work) after the PHY interface is

needed to temporarily hold the extra row address bits from RAS command and later deposit them into the next CAS command. Second or latter CAS commands do not need to carry any row address bits. In addition, the memory timing parameters are adjusted according to Table 4.2.

4.3 Technique 2: Dynamic Latency (DynLat)

In this section, we introduce the motivation, operation and implementation of Dynamic Latency.

4.3.1 Motivation: Remove Unnecessary Latencies

Figure 4.2 shows that the ideal ComboAS scheme should only delay the memory access by one cycle. Unfortunately, we cannot guarantee that ComboAS works ideally all the time: the one-cycle penalty of ComboAS only occurs when CAS commands are back-to-back. In another word, it requires the interval of CAS commands is the minimum delay (tCCD[3]) as demonstrated in Figure 4.2. However, not all the column accesses are back-to-back. For example, when there is a data dependency, the address of the 2nd read depends on the data returned from the 1st read, and thus the interval between these two CAS commands are longer than tCCD.

Note that ComboAS unconditionally adds $tRCD_0$ on top of every tRL, tWL, and tRTP to avoid memory internal hardware conflicts. However, such additional latency is unnecessary when column accesses are not back-to-back. For those non-ideal cases, we define a metric called ***bubble*** to indicate the difference between the actual interval and the minimum interval. When the bubble is big, ComboAS can cause severe performance loss due to the aforementioned reason.

4.3.2 DynLat Operation

Taking a deep look into this issue, while the conventional tRL_0, tWL_0, and $tRTP_0$ are all static values and determined by the memory hardware limitation, the new

[3]Strictly speaking, the minimum interval between two column accesses is max(tCCD, BL/2) where BL is the burst length. Normally, tCCD=4 and BL=8 for LPDDRx usecase.

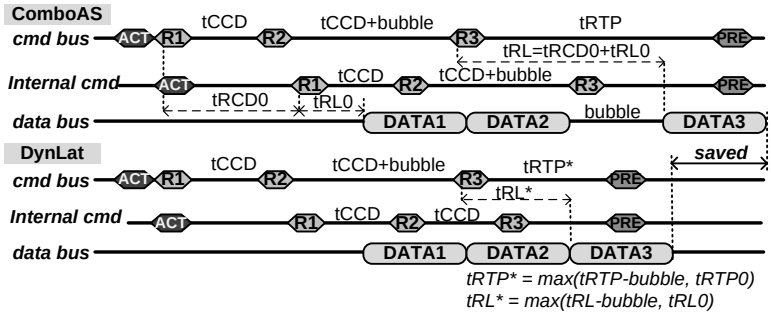

Figure 4.4. The timing diagrams of ComboAS and DynLat.

tRL, tWL, and tRTP parameters in ComboAS become variable as they include the row activation latency ($tRCD_0$) which we only need to pay once for one opened row. We can deduct this $tRCD_0$ overhead from tRL/tWL/tRTP if we find bubbles on the command bus. This observation leads us to a dynamic timing parameter settings, in which such parameters as tRL, tWL, and tRTP are adjustable on-the-fly. We call this technique *Dynamic Latency (DynLat)*.

To demonstrate the idea and the benefit of DynLat, we use Figure 4.4 as an example. The differences between the timing diagrams without and with DynLat are:

- There is no bubble between the first read command **R1** and the second one **R2** (i.e. back-to-back column accesses). To avoid the memory chip internal hardware conflict, the accesses **R2** in ComboAS and DynLat both have the original tRL setting (tRL_0+$tRCD_0$ as listed in Table 4.2).

- Accesses **R2** and **R3** are not back-to-back. In ComboAS, the tRL of **R3** remains tRL_0+$tRCD_0$, which causes bubbles on both the internal command bus and the data bus (the bubble between DATA2 and DATA3). In DynLat, the bubble on the data bus is eliminated by setting the tRL for **R3** to be $max(\text{tRL} - \text{bubbleLength}, \text{tRL}_0)$. By forcing tRL larger than tRL_0, we ensure the command meets the memory chip internal hardware constraint; by subtracting bubbleLength, we guarantee the bubble is removed.

To implement DynLat, we track ***bubble*** and update tRL, tWL, and tRTP after each access:

Accumulated Bubble Length (ABL): We use ABL to store the total bubble

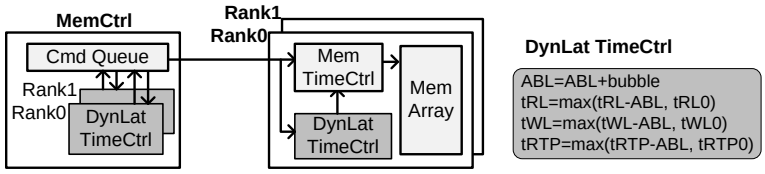

Figure 4.5. The DynLat implementation.

length during transferring access commands for each memory rank[4]. We reset ABL to 0 upon every ACT command and accumulate the bubble length upon every READ or WRITE command according to Equation 4.1.

$$\text{ABL}' = \text{ABL} + (\text{curCycle} - \text{lastCmdCycle}) - \text{minReqDelay} \tag{4.1}$$

In general, it is the summation of the old ABL value and the newly detected bubble length, and it keeps increasing during a page open cycle. In practice, we can set an upper limit to the ABL value to reduce the hardware counter overhead.

Updating parameters: Based on ABL, we then calculate the new tRL, tWL, and tRTP[5] according to Equation 4.2,

$$\begin{aligned} \text{tRL} &= max(\text{tRL}_0 + \text{tRCD}_0 - \text{ABL},\ \text{tRL}_0) \\ \text{tWL} &= max(\text{tRL}_0 + \text{tRCD}_0 - \text{ABL},\ \text{tWL}_0) \\ \text{tRTP} &= max(\text{tRTP}_0 + \text{tRCD}_0 - \text{ABL},\ \text{tRTP}_0) \end{aligned} \tag{4.2}$$

where tRL_0, tWL_0, tRTP_0, and tRCD_0 are the original timing parameters defined by the memory device.

4.3.3 DynLat Implementation

Figure 4.5 shows a memory architecture with DynLat scheme adopted. A DynLat control logic is added to both memory device and memory controller:

Memory Device: Since DynLat introduces variable read and write latencies, the memory device shall track the latest tRL and tWL, so that it can return the

[4]DynLat is designed to track the rank-level timings since all banks share the same interface and tCCD constrains the access interval in one rank.

[5]Write-to-precharge latency equals to tWL+BL/2+tWR, and it is adjusted together with tWL.

data for read or latch the data for write at the correct cycle. For this purpose, we add a new component called *TimeCtrl* to each memory device as shown in Figure 4.5. TimeCtrl tracks the ABL value and updates the timing parameters to the device internal signal delaying circuitry according to Equation 4.2. If a memory rank contains multiple memory devices, their TimeCtrl logics behave in a lockstep mode.

Memory Controller: The same TimeCtrl logic is duplicated in the memory controller so that the optimized command intervals can be correctly generated from the controller. The number of duplications is the same as the memory subsystem rank count. For example, in Figure 4.5, we duplicate two TimeCtrl in the memory controller for a 2-rank configuration.

4.4 Technique 3: Early Precharge/Activation (EarlyPA)

To improve the system performance further, in this section we propose the third technique, Early Precharge/Activation.

4.4.1 Motivation: Leveraging Non-destructive Read

DynLat can remove the unnecessary latency and alleviate the performance drop brought by ComboAS, but it cannot mitigate the performance drop caused by reduced page hit ratio, which is another side effect of the STT-RAM small page size. Although a smaller page size is preferred to avoid over-activation and reduce the energy waste [24], it is only meaningful to close-page memory systems where page locality is not utilized. While today's servers and data centers mostly use close-page policy due to their low data locality, mobile devices still commonly use open-page policy, and their performance is highly sensitive to memory page size.

Figure 4.6 compares the performance of a DRAM system with 4KB pages size and an STT-RAM LPDDR3 system with 256B pages (see Section 4.6 for the detailed simulation methodology). The performance difference greatly depends on the page hit ratio change. The biggest performance loss occurs when page hit ratio

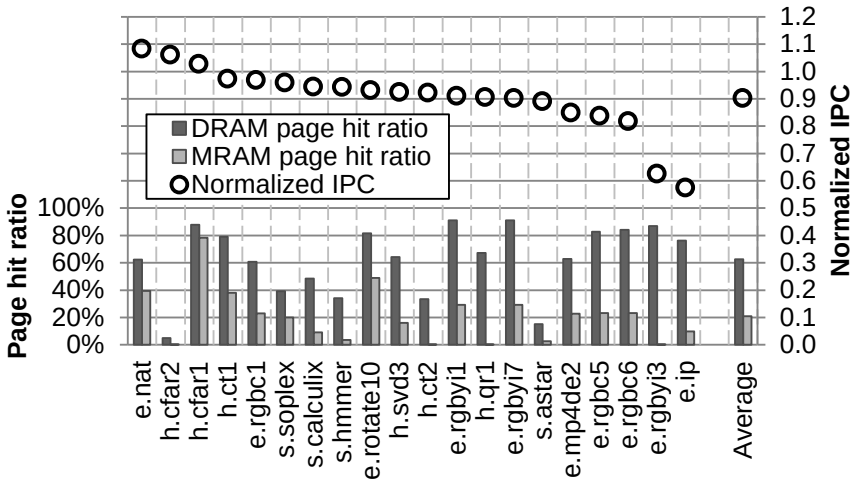

Figure 4.6. The IPC and page hit ratio comparison between a DRAM system and an STT-RAM system.

is significantly reduced[6]. On average, the STT-RAM page hit ratio is decreased by 66% due to its 16X smaller page size. As a result, replacing LPDDR3 DRAM with STT-RAM degrades the performance by 10% on average and up to 24%.

Although the fundamental STT-RAM sensing scheme results in such a disadvantage, we shall notice that the same sensing scheme gives STT-RAM a unique advantage: unlike DRAM reads that destroy the data stored in the DRAM cell, STT-RAM reads are non-destructive. Based on this unique property, we devise our third optimization technique: *Early Precharge/Activation (EarlyPA)*.

4.4.2 EarlyPA Operation

Because DRAM reads are destructive and need data restoration, the DRAM S/A always connects to the bitline during page open, and it also serves as a row buffer. On the contrary, STT-RAM reads are non-destructive. The "row buffer" part of the S/A is not necessary to connect to the input bitline after the data is correctly sensed out. The only reason to reconnect row buffers and bitlines is to write a new data. Previous work also considered decoupling S/As and row buffers [6], but they did not utilize this characteristic to optimize the operation timing. Our EarlyPA

[6]Note that for the workloads whose page hit ratio is less sensitive to the page size, STT-RAM outperforms DRAM (e.g. *e.nat*) because STT-RAM has faster precharge speed and needs no refresh.

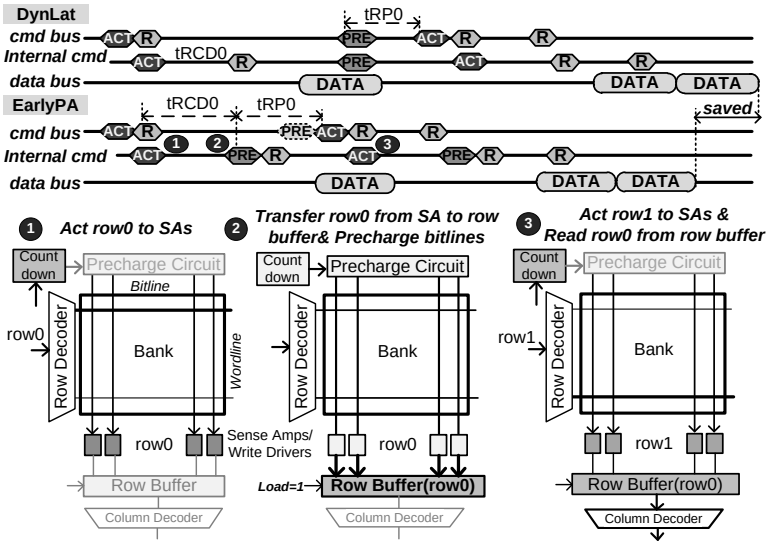

Figure 4.7. The timing diagrams of DynLat and EarlyPA.

technique is to precharge bitlines right after data sensing so that the next ACT command can be issued earlier.

To decouple the "data latching" function out of a normal S/A, we first extract the last stage amplifier (usually a pair of cross-coupled inverters) from the S/A, and evolve it into a full SRAM cell. After this change, we still call the remaining part of the "data sensing" circuit as the S/A, and the SRAM cells then become the row buffer.

The decoupling enables the EarlyPA operations. A read-only example is illustrated in Figure 4.7:

- **Time slot 1**: Upon the first ACT arrival, the S/A starts data sensing, and a self-precharge counter starts counting down from $tRCD_0$.

- **Time slot 2**: The counter triggers a bitline self-precharge (an internal PRE command[7]) after $tRCD_0$. The S/A finishes data sensing, and the row buffer holds a copy of the data.

- **Time slot 3**: When the second ACT arrives, bitlines and S/As are ready for another row activation (row1). At the same time, all the column read accesses to row0 keep proceeding from the row buffer to I/Os.

[7]This self-precharge is different from the auto-precharge operation used in close-page policy. In EarlyPA the self-precharge only precharges the bitlines but the row buffer data remain intact.

The decoupled row buffer allows bitlines to be early-precharged during the buffer column accesses, and we can improve the read performance by issuing PRE and ACT commands for the next row in advance. However, if there is a write access, we need another PRE after the dirty data write-back. Therefore, when memory write occurs, the minimum required delay to issue the next PRE operation (*write-to-precharge delay*) is the same to the conventional scheme (i.e. tWL+BL/2+tWR). Our proposed EarlyPA technique handles write accesses as follows:

- If a write comes before the self-precharge is internally issued, we postpone the self-precharge so that we can leverage the unfinished row activation cycle. To do that, we update the self-precharge counter and reset it to the write-to-precharge delay (i.e. tWL+BL/2+tWR).

- If a write comes after the self-precharge is internally issued, it means that we already disconnect the corresponding row in the memory array. In this case, we need to turn on the corresponding wordline again for writing the data, which brings some latency overhead (i.e. 3 cycles in this work). In addition, we have to reset the self-precharge counter to the *wordline-turn-on delay* plus the *write-to-precharge delay* (e.g. 3+tWL+BL/2+tWR), and the previous self-precharge operation is wasted.

Although frequent write accesses still undermine the EarlyPA performance, it does not cause any timing violation. That is because the memory access order remains unchanged, and the next ACT command is never issued until all the WRITE commands to that row are drained.

4.4.3 EarlyPA Implementation

Similar to the previously proposed ComboAS scheme, the EarlyPA implementation can be transparent to the memory controller and only requires some timing parameter manipulations. The controlling policy for column access commands (READ/WRITE) remains the same. As shown in Figure 4.7, we modify two precharge-related timing parameters:

- tRAS (activation-to-precharge delay) of EarlyPA is set as $tRCD_0$+tRP_0.

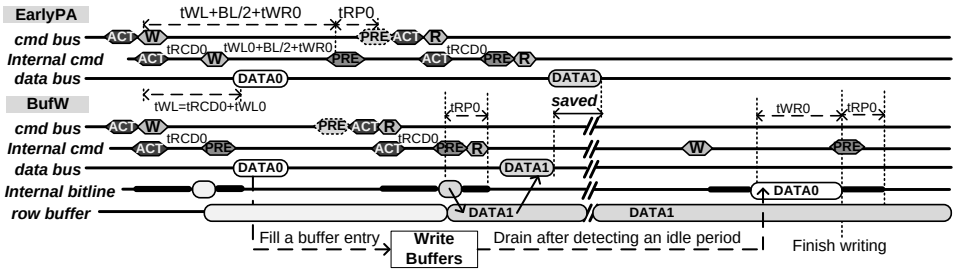

Figure 4.8. The timing diagrams comparison between EarlyPA and BufW.

- tRP (precharing time) value is set as 1 so that the next ACT command can be issued immediately when the self-precharge is finished.

Memory Device: Devices ignore all the PRE commands from the memory controller as EarlyPA automatically precharges the bitlines in advance. Instead, a self-precharge counter is added to each memory device control logic. The counter is set to $tRCD_0$ after every ACT command and reset to tWL+BL/2+tWR or 3+tWL+BL/2+tWR after every WRITE command depending on whether the counter reaches zero or not at the WRITE command arrival. Furthermore, the memory device skips precharge-related timing rule (e.g. tRTP checking) except the tRAS checking as S/As and row buffers are decoupled in the EarlyPA mode.

Memory Controller: Symmetrically, the memory controller manipulates the timing parameters in the same way as memory devices do. An additional modification to the memory controller change the write-to-precharge latency control: after issuing a WRITE command, the minimum required delay for the next PRE command is tWL+BL/2+tWR+tRP_0 instead of tWL+BL/2+tWR.

4.5 Technique 4: Buffered Writes (BufW)

Last but not least, we propose Buffered Writes in this section to optimize the STT-RAM writes.

4.5.1 Motivation: Independent Write Path

EarlyPA utilizes the STT-RAM non-destructive read to issue PRE command in advance, but we also discuss that a WRITE command might undermine the Ear-

lyPA scheme in terms of performance. Fortunately, we can leverage our decoupled row buffers and bitlines to make another optimization.

Traditional DRAM access protocols handle writes during the row activation cycle. It is a good choice for DRAMs because DRAM reads are destructive and need data restoration. Therefore, it is beneficial to leverage the row buffer for write operations so that we can overlap the write latency with the data restoration process. However, STT-RAM reads are non-destructive, and we no longer need to buffer the write data in the row buffer. Instead, since we now have row buffers disconnected from bitlines, we can set up an read-independent data write path and buffer the write data in a separate place. This observation leads us to a *buffered write scheme (BufW).*

4.5.2 BufW Operation

The basic concept of BufW is to store the incoming WRITE commands in a small buffer placed in the memory bank, use a dedicated write path, and only try to issue the internal write operations when we detect a bus idle period or the buffer becomes full. The detailed BufW description is:

- If a write comes and the write buffer is not full, we allocate a new write buffer entry and store the data together with its address into this entry. Because the buffer is essentially a small SRAM array, we assume the buffer allocation can be finished within one memory clock cycle. Also, this write does not affect the EarlyPA operation.

- If a write comes and the write buffer is full, we fall back to basic EarlyPA and complete the write through normal write data path (using row buffer as the data latch).

- When the number of idle cycles for a memory bank exceeds a threshold[8], we switch the memory rank into a write buffer draining phase, in which an FSM moves the data from the write buffer to the memory array. During each drain, we use the address information in write buffer to turn on the

[8]This functionality and the counter are similar to the circuit that controls the DRAM power-down mode.

corresponding wordline but without data sensing. Only the specified column is written while the other columns are masked. The procedure is repeated until the write buffer becomes empty. After that, the bitlines are precharged. Or, if a new command arrives from the memory controller during a write, the write is cancelled, and the coming command is served at first. In another word, the write buffer is read-preemptive [17].

- For each read access, a write buffer lookup is needed since the write buffer might hold the latest copy of the data. Since the write buffer is usually small (e.g. 10-entry), this lookup process can be operated in parallel with the normal column access to the row buffer, hiding the lookup latency. In case of write buffer hit, we add extra read latency to pretend that the data is returned by the memory array.

Figure 4.8 shows the difference between EarlyPA and BufW. When WRITE arrives, In BufW, DATA0 is held in the write buffer until an idle period is detected on the memory bus. BufW outperforms EarlyPA because it can issue the next PRE and ACT earlier when a write occurs. Note that the memory device enters the write buffer draining phase automatically (e.g. an idle period is detected) and exits it implicitly (e.g. the write buffer becomes empty or a bus activity is detected). Therefore, unlike entering/exiting DRAM self-refresh mode, we do not need explicit LPDDRx commands for the entry and exit of the write buffer draining phase.

4.5.3 BufW Implementation

Memory Device: Figure 4.9 shows write buffer implementation. First, a SRAM-based FIFO-organized buffer with n entries is added. Each entry includes row address, column address, the data field (32 bytes in this work), and a bit indicating if it is occupied. We evaluate the hardware overhead of these additional components using NVSim [79] under a 32 nm technology node. The result shows that the area overhead is about 0.004 mm^2 for each memory bank (smaller than 0.1% compared to a 4 Gb DRAM LPDDR3 chip fabricated using 32nm technology with die area of 82 mm^2 [84]). The energy overhead of one access is about 3.6 pJ and the latency is about 0.5 ns.

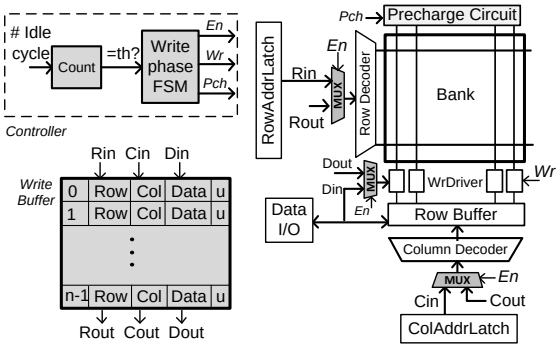

Figure 4.9. BufW Implementation.

Table 4.3. Simulation settings.

Core	1-8 cores, 2GHz, out-of-order ARM cores
SRAM I-L1/D-L1 caches	private, 32KB/32KB, 8-way, LRU, 64B cache line, write-back, write allocate
SRAM L2 cache	shared, 1MB, 16-way, LRU, 64B cache line, write-back, write allocate
Memory controller	open page policy, 32-entry read/write queues per controller, FR-FCFS scheduler
Main Memory	**1**/2/4 channel, 1/**2**/4 ranks-per-channel, 4/**8** banks-per-rank. Timing is configured as Table 4.1

Second, 3 multiplexers are added on the memory array interface to choose different sources for row address, column address, and input data.

Third, a small controller is added to monitor bank states and drain the write buffer when a long idle state is detected.

As a summary, the hardware overhead is negligible.

Memory Controller: The write buffer draining process is transparent to the memory controller until the buffer is full and a new write command comes. In this case, we need to switch the memory controller back to the basic EarlyPA mode. To synchronize the write buffer overflow event between the memory controller and the memory device, we add a *virtual write buffer* to the memory controller for each memory rank. This buffer has the same number of entries as the ones on the memory device side, but each entry only has one bit to indicate if the corresponding entry in the memory device is occupied or not. We update this virtual write buffer using the same algorithm as the one in the memory device. Therefore, the memory controller is able to know the status of the actual write buffer on the memory devices and switch to the basic EarlyPA mode when necessary.

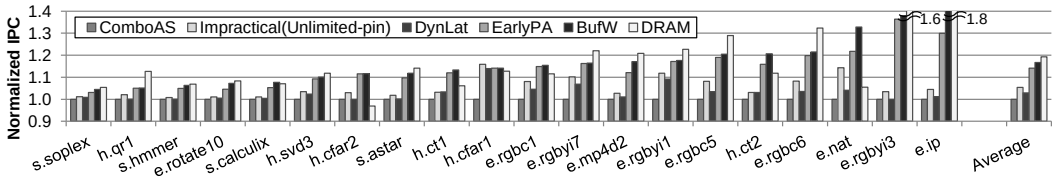

Figure 4.10. Normalized IPC of main memory system with each technique: ComboAS as the baseline, Unlimited-pin, DynLat, EarlyPA, BufW, and DRAM systems.

4.5.4 BufW Discussion

First, as later shown in Section 4.6, the BufW technique is most beneficial to the workloads with heavy write traffic. Therefore, unlike the previous three techniques (i.e. ComboAS, DynLat, and EarlyPA) that we consider are the essential techniques for the commodity STT-RAM success, the BufW technique might serve as an optional technique that is only added for a system that is known to handle heavy memory write traffic.

Second, the BufW technique is different from the previous "cached DRAM" effort [85, 86]. Cached DRAM adds a large amount of SRAMs on a DRAM chip to buffer multiple DRAM rows. The hardware overhead on the DRAM device side is tremendous because their buffer entry is in the unit of page size. For example, an 8KB SRAM cache is added to a 4MB DRAM [85], and it can cause at least 5% die size increase (assuming a 24:1 SRAM/DRAM cell area ratio). On the contrary, our BufW buffers data in the unit of a memory burst length, and it only causes less than 0.1% die size overhead. In addition, fabricating SRAM on a DRAM process degrades the performance. Our BufW technique is immune to this degradation because the buffer is filled on a non-critical path. However, cached DRAM suffers from this problem as cache is latency-sensitive.

4.6 Experiments

4.6.1 Simulation Methodology

We model a 2GHz out-of-order ARMv7 microprocessor using our modified version of gem5 [87]. DRAMSim2 [88] is integrated and modified to model the main memory system. Open-page policy with FR-FCFS [89] scheduling is accurately

modeled.

We user the timing and power parameters in in Table 4.1 to simulate our DRAM and STT-RAM devices. Both of DRAM and STT-RAM are projected to work on a 533MHz LPDDR3 bus. Unless specified, our default system configuration comprises a single-core processor with a main memory system with 1 channel, 2 ranks, and 8 banks. We also give detailed sensitivity studies to vary the number of cores, channels, ranks, and banks in Section 4.6.4. We use open-page policy with row-interleaving which is widely used to maximize the memory-level parallelism. More details for the simulation setting are provided in Table 4.3.

We select 20 memory-intensive benchmarks from SPEC 2006 [90], EEMBC 2.0 [91], and HPEC [92]. We form the multi-core workloads by randomly choosing from all the workloads. We fast-forward each simulation to the pre-defined break-point at the code region of interest, warm-up 10 million instructions, and simulate for at least 1 billion instructions. To measure the system performance, we use instruction per cycle (IPC) as the metric.

4.6.2 Performance Speedup of Individual Benchmarks

Figure 4.10 shows the performance speedup of the STT-RAM system with each proposed technique. The DRAM system performance is also provided for comparison. We use ComboAS as the baseline which has the worst performance. The second bar is the performance of an impractical implementation where 2 more pins are added (referred to as Unlimited-pin in the chart). Compared to Unlimited-pin, Figure 4.10 shows that the performance of ComboAS is degraded by 5% on average. However, after adopting DynLat to reduce the unneccesarry latency overhead caused by ComboAS, the sytem performance is bounsed back by 3% on average (up to 14%). Thus, the performance of ComboAS system with DynLat is comparable to the Unlimited-pin in most cases.

In addition, by leveraging the STT-RAM non-destructive read with EarlyPA, we improve the performance by 14% (up to 36%). Furthermore, by adding BufW scheme, we provide another performance boost, and the overall performance improvement reaches 17% on average (up to 42%). After adopting all the proposed techniques, the overall performance of the projected STT-RAM system is compet-

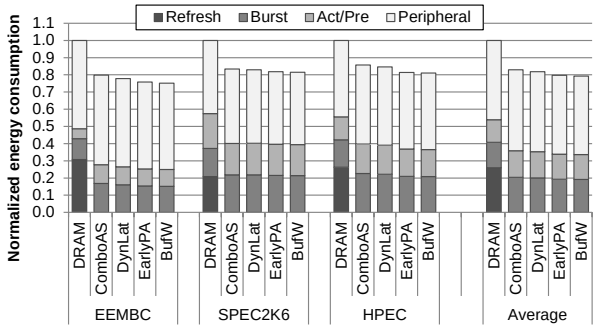

Figure 4.11. Normalized energy consumption of DRAM system and STT-RAM system adopted different techniques.

itive to the DRAM counterpart.

The performance improvement of each workload is different because of two reasons. First, memory-intensive workloads benefit more from our proposed techniques because more efficient memory accesses provide larger system performance improvement. Second, write-intensive workloads benefit more from BufW, but the benefit is reduced if the write ratio is too high and the write buffer is always full. We also give the sensitivity study on the number of write buffer entries in Section 4.6.4.

4.6.3 Energy Consumption Analysis

Sleeping Mode. The battery life is critical to every mobile device. To reduce standby power, modern devices (e.g. smartphones) turn off as many components (e.g. CPU, GPU, GPS, etc.) as possible during the sleeping mode. However, DRAM cannot be turned off because it is volatile. Commonly, DRAM is switched from auto-refresh mode to self-refresh mode before the memory controller becomes off-line. Although the self-refresh mode can generally reduce the DRAM refresh power by 50%-80% (depending on the ambient temperature), it is still a dominant power contributor to the standby power. For instance, a smartphone usually comsumes 25mW-30mW during standby (e.g. iPhone 4S), but its 512MB DRAM still consumes 6mW even using self-refresh. Replacing DRAM with STT-RAM can eliminate the memory standby power (STT-RAM IDD6 is 0), and easily improve the mobile device battery life.

Operating Mode. While the performance of our optimized STT-RAM system

is similar to the conventional DRAM system, the real deal breaker is the energy consumption saving. Figure 4.11 shows the comparison of energy consumption between the DRAM and the STT-RAM systems, in which each value is divided to refresh energy, burst energy, activation/precharge energy and background peripheral circuit energy[9]. The energy overhead of each proposed technique is also included.

Compared to DRAM, STT-RAM-based system does not consume any refresh energy because of its non-volatility, and it is the major source of the STT-RAM energy saving. We need to mention that, as shown in Table 4.1, the read/write energy of STT-RAM is larger than DRAM because STT-RAM has smaller sense margin and the memory cell is difficult to write. Thus, the STT-RAM burst energy is usually larger than the DRAM one. The energy consumed by peripheral circuits is similar between DRAM and STT-RAM because we do not apply any circuit optimization to it in this work. But we should note that the peripheral energy of STT-RAM can be further reduced if STT-RAM is allowed to go into power-collapse mode frequently during the idle state.

Figure 4.11 shows the ComboAS STT-RAM system reduces the total energy consumption by more than 17% compared to DRAM system. After adopting proposed DynLat, EarlyPA and BufW techniques, the energy consumption of STT-RAM system can be further reduced by 4.5% on average since the performance is increased and the total execution time is reduced. Considering the comparable performance and smaller energy consumption, STT-RAM is an attactive candidate to build the main memory system.

4.6.4 Sensitivity Study

The number of Channels, Ranks, and Banks. To evaluate our techniques under different memory configurations, we change the number of memory channels, ranks, and banks but keep the total memory capacity the same. Figure 4.12 is the normalized IPC under each configuration showing that the proposed techniques improve the performance by 16%-20% under different configurations. When the

[9]In this work, we do not model the self-refresh mode for DRAM systems and the power-collapse mode for STT-RAM systems. The realistic background energy can be smaller than our simulated numbers.

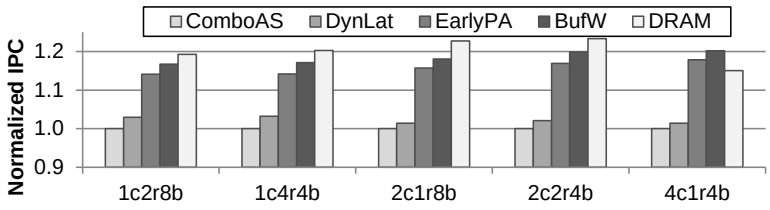

Figure 4.12. Normalized IPC of each technique when the number of channels, ranks and banks are changed.

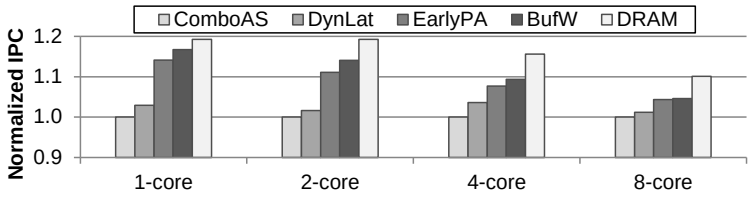

Figure 4.13. Normalized IPC of each technique when the number of cores are changed.

number of higher-level parallelism (channels) is increased, the performance difference between naive STT-RAM and DRAM system is decreased. In this case, the performance of STT-RAM system (e.g. 4c1r4b) after adopting all proposed techniques may be better than the DRAM system. But adding higher-level parallelism is expensive, especially for mobile systems. Therefore, using our proposed techniques is a more effective method to boost the performance.

The Number of Cores. Figure 4.13 shows the performance improvement of our proposed techniques under multi-core system configuration. After adopting all the proposed techniques, the performance of 2core/4core/8core system is improved by 14%/10%/5% on average. When the number of cores is increased, the page hit ratio of main memory system is decreased as the number of processes simultaneously accessing memory goes up, which is also proved in some previous work [24, 53]. The good news is that the performance hit caused by STT-RAM smaller page size is also less severe in these cases, and our techniques bring extra performance gain.

The Number of Write Buffer Entries. For BufW, the number of write buffer entries is a predetermined parameter and should be studied. The possibility of write buffer overflow decreases as the number of entries increases, and the system performance is improved. In case of write buffer overflow, our solution is to utilize the conventional write path which is through the row buffer. We need another PRE

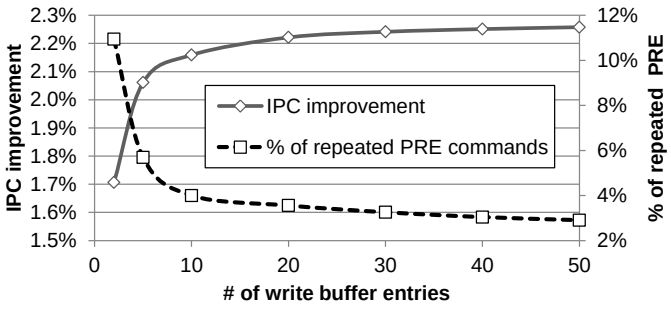

Figure 4.14. The IPC improvement of BufW over EarlyPA and the percentage of repeated PRE commands when the write buffer size increases.

command after that to close the row, and the previous PRE command issued by EarlyPA in advance becomes useless. A larger write buffer can reduce the chances of repeated PRE commands. However, adding more write buffer entries has area overhead. Therefore, we need a trade-off between performance and cost.

Figure 4.14 shows the IPC improvement and the percentage of repeated PRE commands when the number of write entries is increased from 2 to 50. The result shows that IPC is increased rapidly when the number of write buffer entries increases from 2 to 10, and the trend becomes flatten after 10. Therefore, we select 10 as our write buffer size.

4.7 Summary

The shift from PCs to mobile devices is requesting low-power memory solutions, and non-volatile memory technologies such as STT-RAM are promising candidates. Compared to DRAM, STT-RAM has many unique features such as *small page size*, *non-destructive read*, and *independent write path*. The smaller page size brings challenges in designing commodity STT-RAM that can be deployed on the same LPDDRx interface as DRAM, and can cause performance degradation to mobile systems where the page hit ratio is important. In this work, we propose four techniques: *ComboAS* and *DynLat* to solve the DRAM-compatibility issue; *EarlyPA* and *BufW* to further improve the performance by exploiting the STT-RAM unique features, and they mitigate the performance loss caused by lower page hit ratio. Combined together, our solution enables a commodity STT-RAM on LPDDR3 interface with a much optimized performance (17% on average and

up to 42%). It makes LPDDR3 STT-RAM have competitive performance but save 21% energy compared to LPDDR3 DRAM does. The proposed architecture is a step forward to the future energy-efficient memory design.

Chapter 5

NVM Caches: Wear Leveling

Compare to NVM main memories, building NVM-based on-chip caches is more challenging. It is because caches handle much more writes than storage and main memory do, but NVM only has a limited write endurance. Worse, this limited write endurance issue is further amplified by the conventional cache management policies: these polices were originally designed for SRAM caches and result in significant non-uniformity in terms of writing to cache blocks, which would cause heavily-written NVM blocks to fail much earlier than most other blocks. Therefore, in this chapter, we study on how to design an effective wear-leveling technique for NVM caches.[1]

There are already many wear-leveling techniques to extend the lifetime of NVM main memories [7, 8, 9], but the difference between cache and main memory operational mechanisms makes the existing wear-leveling techniques for NVM main memories inadequate for NVM caches. This is because writes to caches have *intra-set variations* in addition to *inter-set variations* while writes to main memories only have *inter-set variations*. According to our analysis, intra-set variations can be comparable to inter-set variations for some workloads. This presents a new challenge in designing wear-leveling techniques for NVM caches.

To minimize both inter- and intra-set write variations, we introduce i^{2}WAP (inter/intra-set Write variation-Aware cache Policy), a simple but effective wear-leveling scheme for NVM caches [93, 94]. i^{2}WAP features two schemes: 1) *Swap-*

[1]This work is published as "i2WAP: Improving Non-Volatile Cache Lifetime by Reducing Inter- and Intra-Set Write Variations" on HPCA2013.

Shift is enhanced from the existing main memory wear-leveling techniques and aims to reduce the cache inter-set write variation; 2) *Probabilistic Set Line Flush* is designed to alleviate the cache intra-set write variation, which is a severe problem for NVM caches and has not been addressed before.

5.1 Inter-Set and Intra-Set Write Variations

Write variation is a significant concern in designing any cache/memory subsystems with a limited write endurance. Large write variation can greatly degrade the product lifetime because only a small subset of memory cells that experience the worst-case write traffic can result in an entire dead cache/memory subsystem even when the majority of cells are far from wear-out.

While the write variation in NVM main memories has been widely studied [7, 6, 5, 8, 9], to our knowledge, the write variation in NVM caches has not. Wear-leveling in caches brings extra challenges since there are write count variations inside every cache set (i.e. *intra-set variations*) as well as across different cache sets (i.e. *inter-set variations*). In order to demonstrate how severe the problem is for NVM caches, we first do a quick experiment.

5.1.1 Definition

The objective of wear-leveling is to reduce write variations and make write traffic uniform. To quantify the cache write variation, we define the *coefficient of inter-set variations* (InterV) and the *coefficient of intra-set variations* (IntraV) as follows,

$$InterV = \frac{1}{W_{aver}} \sqrt{\frac{\sum_{i=1}^{N} \left(\sum_{j=1}^{M} w_{i,j}/M - W_{aver} \right)^2}{N-1}} \tag{5.1}$$

$$IntraV = \frac{1}{W_{aver} \cdot N} \sum_{i=1}^{N} \sqrt{\frac{\sum_{j=1}^{M} \left(w_{i,j} - \sum_{j=1}^{M} w_{i,j}/M \right)^2}{M-1}} \tag{5.2}$$

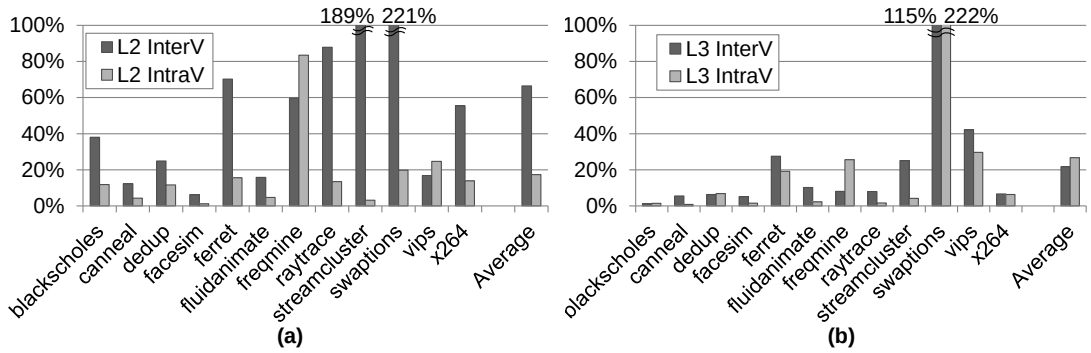

Figure 5.1. The coefficient of variation for inter-set and intra-set write count of L2 and L3 caches in a simulated 4-core system with 32KB I-L1, 32KB D-L1, 1MB L2, and 8MB L3 caches.

where we use Bessel's correction to calculate the standard deviation. $w_{i,j}$ is the write count of the cache line located at set i and way j, and W_{aver} is the average write count:

$$W_{aver} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{M} w_{i,j}}{NM} \tag{5.3}$$

where N is the number of cache sets, and M is the cache associativity. In short, InterV is the CoV (coefficient of variation) of the average write count within cache sets; IntraV is the average of the CoV of the write counts cross a cache set[2]. If $w_{i,j}$ are all the same, InterV and IntraV are both zero.

Figure 5.1 shows the experimental results of InterV and IntraV in our simulated 4-core system with 32KB 8-way I-L1, 32KB 8-way D-L1, 1MB 8-way L2, and 8MB 8-way L3 caches. The detailed simulation methodology and the setting are described in Section 5.6. We compare InterV and IntraV in L2 and L3 caches as we anticipate that NVM will be first used in low-level caches. We observe from Figure 5.1 that:

1. Large InterV: Cache lines in different sets might have totally different write frequencies because applications usually have biased memory residency. For instance, *streamcluster* has 189% InterV in L2, and *swaptions* has 115% InterV in L3. On average, InterV is 66% in L2 and 22% in L3.

[2]We use average CoV instead of maximum CoV to keep the definitions of intra-set (the CoV of the averages) and inter-set variations (the average of the CoVs) symmetric.

2. Large IntraV: A cache line holding hot data might absorb most of the cache write operations, and thus the remaining M-1 lines in the set (for an M-way associative cache) are written less frequently. This causes IntraV. For example, *freqmine* has 84% L2 IntraV, and *swaptions* has 222% L3 IntraV. On average, IntraV is 17% in L2 and 27% in L3.

3. L2 has larger InterV than L3: L2 caches are private for each processor but all processors share one L3 cache, thus L3 has smaller InterV since it mixes different requests. Considering the average write count to L2 is also higher, the larger L2 InterV makes the limited write endurance problem even worse.

4. IntraV is comparable to InterV: Our results show that IntraV is roughly the same or even larger compared to InterV for some workloads. Combining these two types of write variations together significantly shortens the NVM cache lifetime.

5.2 Cache Lifetime Metrics

Cache lifetime can be defined in two ways: *raw lifetime* and *error-tolerant lifetime*. We define the raw lifetime by the first failure of a cache line without considering any error recovery effort. On the other hand, we can extend the raw lifetime by using error correction techniques and paying overhead in either memory performance or memory capacity [10, 11, 12, 13], and we call it the error-tolerant lifetime. In this work, we focus on how to improve the raw lifetime at first as it is the base of the error-tolerant lifetime. Later, we discuss the error-tolerant lifetime in Section 5.7.2.

The target of maximizing the cache raw lifetime is equivalent to minimizing the worst-case write count to a cache line. However, it is impractical to obtain the worst-case write count throughout the whole product lifetime which might span several years. Instead, in this work, we model the raw lifetime by using three parameters: *average write count*, *inter-set write count variation*, and *intra-set write count variation*. The detailed methodology can be described as follows,

1. The cache behavior is simulated during a short period of time t_{sim} (e.g. 10 billion instructions on a 3GHz CPU).

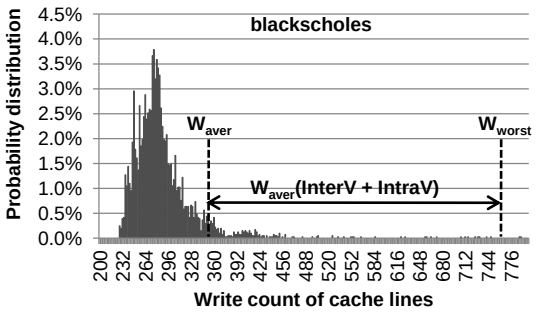

Figure 5.2. The L2 cache write count probability distribution function (PDF) of blackscholes.

2. Each cache line write count is collected to get a average write count W_{aver}. Also, we calculate InterV and IntraV according to Equation 5.1 and Equation 5.2.

3. Assuming the total write variation of a cache line is the summation of its inter- and intra-set variations[3], we then have $W_{var} = W_{aver} \cdot (InterV + IntraV)$.

4. The worst-case write count is predicted as $W_{aver} + W_{var}$ to cover the vast majority of cases. While it is approximate, Figure 5.2 validates the feasibility of this approach.

5. Assuming the general characteristics of cache write operations for one application do not change with time[4], the lifetime of the system can be defined as:

$$t_{total} = \frac{W_{max} \cdot t_{sim}}{W_{aver} + W_{var}} = \frac{W_{max} \cdot t_{sim}}{W_{aver}(1 + InterV + IntraV)} \tag{5.4}$$

where W_{max} represents the write endurance.

In addition, the lifetime improvement (LI) of a cache wear-leveling technique can be expressed by Equation 5.5, where W_{aver_base} and W_{aver_opt} are the average

[3]InterV and IntraV are not independent, but the worst-case variation can be modeled as the sum of them.

[4]If system runs different applications over time, the cache write variance can be reduced. However, in this work, we only consider the worst case. It occurs in some practical cases, such as embedded applications in which the data layout could largely remain the same.

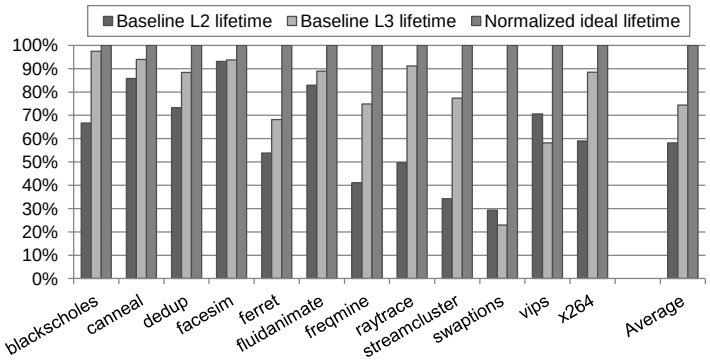

Figure 5.3. The baseline lifetime of L2 and L3 caches normalized to the ideal lifetime (no write variations in the ideal case).

write count before and after wear-leveling, respectively:

$$LI = \frac{W_{aver\_base}(1 + InterV_{base} + IntraV_{base})}{W_{aver\_opt}(1 + InterV_{opt} + IntraV_{opt})} - 1 \tag{5.5}$$

In order to increase LI, we need to reduce InterV and IntraV while not significantly increasing W_{aver}.

We apply the state-of-the-art LRU (least recently used) cache management policy as the baseline, and Figure 5.3 shows how large the write variation it can cause. Compared to the ideal case where cache writes are evenly distributed, LRU can shorten the cache raw lifetime by 20%-30% under some workloads (e.g. *swaptions*). If this is the case of future NVM caches, our system will quickly fail even when most of the NVM cells are still healthy. Therefore, it is critical to design a write variation-aware cache management for NVM caches, and we need a cache management policy that can reduce both InterV and IntraV.

5.3 Starting from Inter-Set Write Variations

5.3.1 Challenges in Cache Inter-Set Wear-Leveling

The existing wear-leveling techniques [7, 5, 8, 9] focus on increasing the lifetime of NVM main memory. The principle behind these techniques is to introduce an address re-mapping layer. This principle remains the same for cache inter-set wear-leveling, but we should reduce the performance overhead during address

re-mapping since caches are accessed more frequently than main memories.

Using Data Movement: Main memory wear-leveling techniques must use data movement during a re-mapping because any data loss in main memories is unrecoverable. Moving cache lines from one set to another is costly. First, data movement requires temporary data storage. Second, one cache set movement involves multiple reads and writes, it blocks the cache port, and thus it might cause significant system performance degradation. *Start-Gap* [8] is a recently proposed technique for NVM main memory wear-leveling. If we directly extend *Start-Gap* to handle the cache inter-set wear-leveling, it falls into this category.

Using Data Invalidation: Another option to implement set address re-mapping for NVM caches is data invalidation. We can use cache line invalidation because we can always restore the cache data later from lower-level memories as long as they are clean. This unique feature of caches provides us a new opportunity to design a low-overhead cache inter-set wear-leveling technique.

Data invalidation saves the temporary storage and the data movement latency as well. To quantify the performance difference between data movement and invalidation, we use L3 cache inter-set wear-leveling as an example. Equation 5.6 lists the timing overhead of a cache set movement operation, in which M is cache associativity and t_{L3} is the L3 access latency (we assume symmetric read/write latency for the sake of simplicity).

$$t_{move} = M \times (2 \times t_{L3}) \tag{5.6}$$

The timing overhead of a cache set invalidation operation is hard to predict precisely since it highly depends on workloads. Equation 5.7 gives a first-order estimation, where $HitR$ is the L3 hit rate, $WriteR$ is the L3 write ratio, and t_{MM} is the main memory latency. $t_{invalid}$ consists of two parts: writing back the dirty data in this set to main memory; restoring data from main memory to L3 which should be hit later. In the first part, $HitR \times WriteR$ is used to estimate the percentage of dirty blocks, and we assume that the write back buffer can hide the main memory write latency; In the second part, we assume that the data returned from main

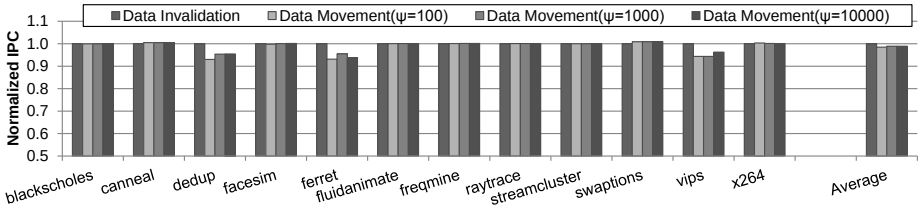

Figure 5.4. The performance comparison between data invalidation and data movement.

memory can be forwarded to L2 before being written into L3.

$$t_{invalid} = M \times HitR \times WriteR \times t_{L3} + M \times HitR \times t_{MM} \tag{5.7}$$

We generally have $t_{invalid} < t_{move}$ since $WriteR$ is usually small. To further quantify the performance difference between these two, we also simulate two systems, respectively (see Section 5.6 for detailed simulation settings). In the data movement system, we extend the Start-gap technique [8] and trigger a cache set movement after every ψ cache writes. In the second system, we do not move the cache set but only invalidate it (write back if dirty). Figure 5.4 shows the performance comparison under different ψ settings (i.e. 100, 1,000 and 10,000). Compared to the data invalidation system, the data movement system has worse performance (i.e. 2% on average and up to 7% when ψ equals to 100). For the data invalidation system, the performance overhead comes from writing back the dirty data to main memory and restoring data which should be hit later from main memory. The widely-used MSHR technique [95] can effective hide these latencies. However, On the other hand, the performance overhead in the data movement system is always there since we cannot move data in a non-blocking way.

5.3.2 Swap-Shift (SwS)

Considering data invalidation is more favorable in cache inter-set wear-leveling, we modify the existing main memory wear-leveling technique and devise a new technique called *Swap-Shift* (SwS).

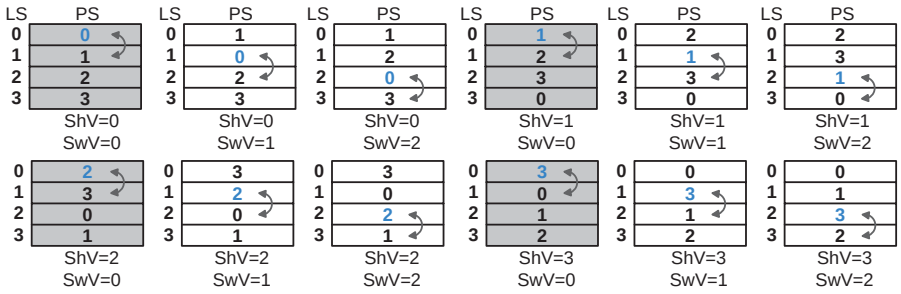

Figure 5.5. One SwS shift round in a cache with 4 sets.

5.3.2.1 SwS Architecture

The concept of SwS is to periodically shift cache set locations. Instead of shifting all the cache sets at once which hits performance significantly, SwS only swaps the data of two neighboring sets at once. SwS can eventually shift all the cache sets by one offset after N-1 swaps, where N is the number of cache sets.

SwS uses a global counter to store the number of cache writes, and we annotate it as Wr. It also uses two registers, SwV (changing from 0 to N-2) and ShV (changing from 0 to N-1), to track the current status of swaps and shifts, respectively. SwS has swap round and shift round::

- **Swap Round (SwapR)**: Every time Wr reaches a specific threshold (Swap Threshold, ST), a swap between cache set $[SwV]$ and set $[SwV+1]$ is triggered. Note that this swap operation only exchanges the set IDs, and invalidates the data stored in these two sets (needs write-back if the data are dirty). After that, SwV is incremented by 1. One swap round (SwapR) consists of N-1 swaps and indicates that all the cache set IDs are shifted by 1.

- **Shift Round (ShiftR)**: ShV is incremented by 1 after each SwapR. At the same time, SwV is reset to 0. One shift round (ShiftR) consists of N shifts (i.e. SwapR).

Figure 5.5 is an example of how SwS shifts the entire cache by multiple swaps. It shows the SwV and ShV values during a complete ShiftR, which consists of 4 SwapR; it also shows that one SwapR consists of 3 swaps and all cache sets are shifted by 1 after each SwapR. In addition, after one ShiftR, all cache sets are

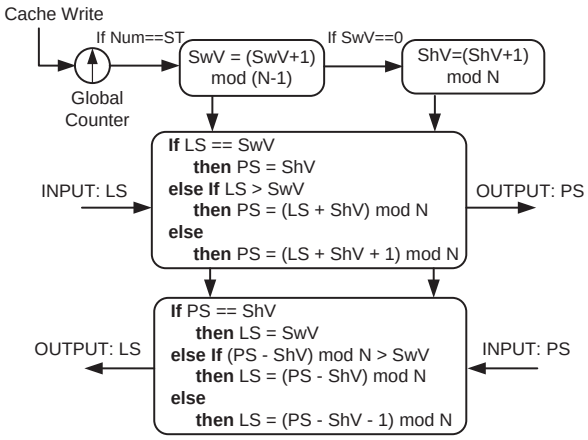

Figure 5.6. The mapping between logical (LS) and physical set index (PS) in SwS.

shifted to the original position and all the logical set indices are the same as the physical ones.

The performance penalty of SwS is small because only two sets are swapped at once and the swap interval period can be long enough (e.g. million cycles) by adjusting ST. The performance analysis of SwS is in Section 5.7.1.

5.3.2.2 SwS Implementation

Figure 5.6 shows the SwS implementation. When a logical set number (LS) arrives, the physical set number (PS) can be computed based on three different situations:

1. If $LS = SwV$, it means that this logical set is exactly the cache set should be swapped in this ShiftR. Therefore, PS is mapped to the current shift value (ShV).

2. If $LS > SwV$, it means that this set has not been shifted in this ShiftR. Therefore, PS is mapped to $LS + ShV$.

3. If $LS < SwV$, it means that this set has been already shifted. Therefore, PS is mapped to $LS + ShV + 1$.

When a dirty cache line is written back to the next level of cache, the logical set address needs to be re-generated. The mapping from PS to LS is symmetrical and is also given in Figure 5.6. This mapping policy can be verified by the simple

example in Figure 5.5. Because SwV and ShV are changed along with cache writes, the mapping between LS and PS change all the time. This scheme balances the writes to different physical sets, reducing cache InterV.

Compared to a conventional cache architecture, the set index translation step in SwS only adds a simple arithmetic operation and can be merged into the row decoder. We synthesize the LS-to-PS address translation circuit in a 45nm technology, and the circuit can handle a LS-to-PS translation within one cycle under a 3GHz clock frequency.

5.4 Intra-Set Variation: A More Severe Issue

SwS only reduces cache inter-set write variations. Our experiment later in Section 5.6.2 shows that SwS alone cannot reduce intra-set variations. In this section, we start with two straightforward techniques and then follow with a much improved technique, called PoLF, to tackle the cache intra-set variation problem.

5.4.1 Set Line Flush

Intra-set write variations are mainly caused by hot data being written more frequently than others. For example, if a cache line is frequently accessed by cache write hits, the corresponding cache set must have a highly unbalanced write distribution.

Traditionally, caches use LRU replacement policy to avoid evicting useful cache lines by marking every accessed block marked towards the MRU (most recently used) position. The LRU policy rarely replaces the hot data that are frequently accessed by cache write hits. This increases the write count of one block and the intra-set write variation of the corresponding set.

To solve this problem, we first consider a *set line flush* (LF) scheme. When there is a cache write hit, LF puts the new data into the write-back buffer directly instead of writing it to the hit data block, and then marks the cache line as *INVALID*. This process is called *set line flush*. Using LF, the block containing the hot data has the opportunity to be replaced by other cold data, and the hot data can be reloaded to other cache lines. We invalidate the hot data line instead of moving it

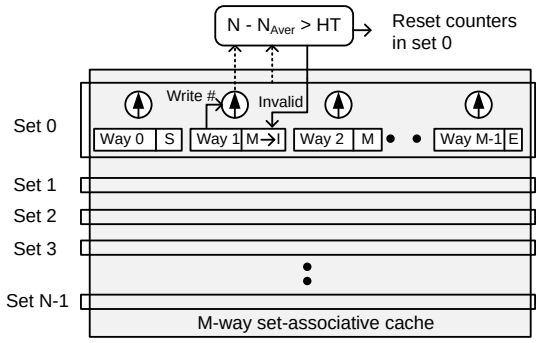

Figure 5.7. The cache architecture of HoLF. The counters to store the write count are added to every cache line.

to other positions due to the same performance concern explained in Section 5.3.1.

LF balances the intra-set write count, but it flushes data on every cache write hit regardless of data hotness. Obviously, LF greatly harms performance as it evicts useful cache line every time. Instead, we need a scheme that only flushes hot cache lines that have been heavily written.

5.4.2 Hot Set Line Flush

We can improve the LF scheme by tracking the write count of each cache line and storing this counter in cache tags. We call this enhanced scheme *hot set line flush* (HoLF). We can detect a hot cache line if its counter is greater than the average value of that cache set by a predetermined threshold, and thus we should flush it. In this way, we can load another data into this cache line, and reload the hot data into a relatively cold cache line. Figure 5.7 shows the HoLf architecture.

However, HoLF is still impractical. HoLF adds a large area overhead since it requires one counter for every cache line. Considering the typical cache line is 64-byte wide and assuming the write counter is 20-bit, the hardware overhead is more than 3.7%. HoLF also degrades performance because it updates both maximum and average write counter values in every cache set. It is infeasible to initiate multiple arithmetic calculations for every cache write.

Due to these reasons, we stop the HoLF discussion and switch to a further improved solution called PoLF.

5.4.3 Probabilistic Set Line Flush

The key of *probabilistic set line flush* (PoLF) is to flush hot data probabilistically instead of deterministically.

5.4.3.1 Probabilistic Invalidation

Unlike HoLF, PoLF only maintains one global counter to count the number of write hits to the entire cache, and it flushes a cache line when the counter saturates regardless of the cache line hotness. Although we cannot guarantee that the hottest data would be flushed, the probability of PoLF selecting a hot data line is high: the hotter the data is, the more likely it will be selected when the global counter saturates. Theoretically, PoLF can still flush the hottest cache line with only one global counter.

Maintaining LRU: Normal LRU policy marks the age bits of the evicted cache line as LRU during a cache line invalidation. However, PoLF should not modify age bits during a probabilistic invalidation. Otherwise it is possible that after invalidating a single hot cache line, the same data will be reinstalled in the very same line on a subsequent miss. Therefore, in our design, when PoLF flushes a cache line in response to a probabilistic invalidation, the cache line age bits are unchanged. Later, a subsequent miss will invalidate the actual LRU line and reinstall the hot data in that line. The cache line evicted by probabilistic invalidation remains invalid until it becomes the actual LRU line.

Comparison with Other Policies: Figure 5.8 shows the behavior of a 4-way cache set managed by LRU, LF, and PoLF polices under an exemplary access pattern. We observe that:

1. For LRU, the hot data a_0 is moved to the MRU position (age bits=3) after each write hit and is never replaced by other data. Thus, the intra-set variation using LRU is the largest one among all the polices.

2. For LF, each write hit causes its corresponding cache line to be flushed. The age bits are not changed during write hits. The intra-set variation is reduced compared to the LRU policy because the hot data a_0 is reloaded into another cache line. However, data a_1 is also flushed since every write hit causes one cache line flush, and it brings one additional access miss.

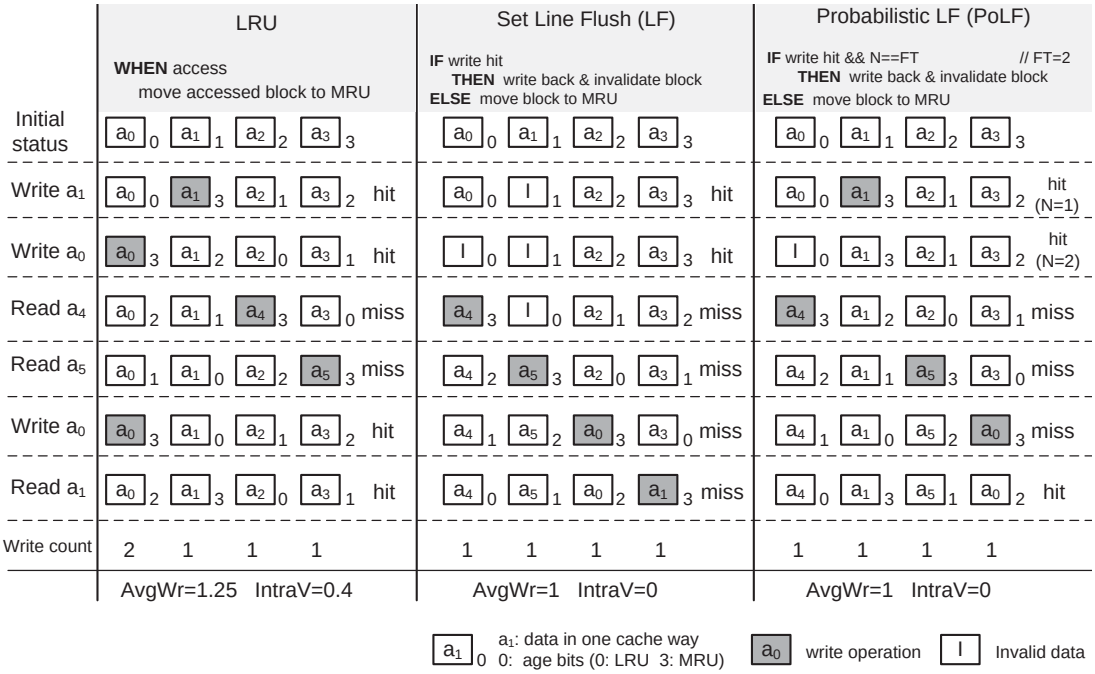

Figure 5.8. The behavior of one cache set composed of 4 ways under LRU, LF, and PoLF polices for the same access pattern. The total write count of each cache way, the average write count and the intra-set variation are marked, respectively.

3. For PoLF, we let every other write hit cause a cache line flush (i.e. line-flush threshold FT=2)[5]. Compared to the LRU policy, its intra-set variation is reduced because the hot data a_0 is moved to another cache line. In addition, compared to LF, cache miss rate is reduced because a_1 is not flushed.

From this example, we can see that PoLF maintains a high probability of replacing a hot cache data and thus reduces IntraV.

5.4.3.2 PoLF Implementation

Figure 5.9 shows the PoLF implementation. The only hardware overhead of PoLF is a global counter (one counter for the entire cache) that tracks the total number of write hits to the cache. The counter is only incremented at each write hit event. If the counter saturates at one threshold, then the cache will record the write operation that causes the counter saturation, and invalidate the line corresponding

[5] FT is set as 2 for this illustration. The realistic value is much larger, and we discuss it in Section 5.6.

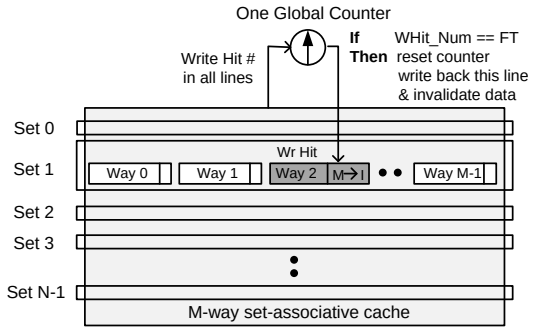

Figure 5.9. The cache architecture of PoLF. Only one global write hit counter is added to the entire cache.

to that write hit. The tunable parameter of PoLF is the line-flush threshold FT.

5.5 i^{2}WAP: Putting Them Together

We combine SwS and PoLF together to form our inter- and intra-set write variation -aware policy, i^{2}WAP. In i^{2}WAP, SwS and PoLF work independently: SwS reduces InterV, and PoLF reduces IntraV.

The total write variations can be reduced significantly and the product lifetime can be improved. The implementation overhead of i^{2}WAP is:

- One global counter to store the number of write accesses (for SwS);
- Two registers to store the current cache set swapping and shifting values (for SwS);
- One global counter to store the number of write hits (for PoLF).

5.6 Experiments

In this section, we first describe our experiment methodology, then we demonstrate how SwS and PoLF reduce InterV and IntraV, respectively. Finally, we show how i^{2}WAP improves the NVM cache lifetime.

Table 5.1. Workload characteristics in L2 an L3 caches under our baseline configuration.

	L2 cache		L3 cache	
Workload	WPKI	TPKI	WPKI	TPKI
blackscholes	0.07	0.4	0.04	0.3
canneal	0.04	23	0.01	15
dedup	1.1	4.8	0.4	0.8
facesim	3.3	4.7	1.1	1.4
ferret	1.8	6.3	0.2	0.5
fluidanimate	0.4	1.4	0.3	0.8
freqmine	1.3	6.7	0.2	0.4
raytrace	0.56	0.62	0.03	0.25
streamcluster	3.7	4.2	0.9	1.1
swaptions	1.4	2.9	0.02	0.06
vips	1.1	4.4	0.6	1.0
x264	0.7	16.1	0.2	0.5

5.6.1 Baseline Configuration

Our baseline is a 4-core CMP system. Each core consists of private L1 and L2 caches, and all the cores share an L3 cache. Our experiment makes use of a 4-thread OpenMP version of the PARSEC 2.1 [72] benchmark workloads[6]. We run single applications since they generally represent the worst case. The native inputs are used for the PARSEC benchmark to generate realistic program behavior. We modify the gem5 full-system simulator [87] to implement our proposed techniques and use it to collect cache accesses. Each gem5 simulation run is fast forwarded to the pre-defined breakpoint at the code region of interest, warmed-up by 100 million instructions, and then simulated for at least 10 billion instructions. The characteristics of workloads are listed in Table 5.1, in which WPKI and TPKI are writes and transactions per kilo-instructions, respectively.

In this work, we use ReRAM L2 and L3 caches as an example. Our techniques and evaluations are also applicable to other NVM technologies. Table 5.2 lists the simulation parameters, and the circuit-level cache parameters (e.g. access latency) are obtained from NVSim [79].

[6]A supplementary experiment on multi-programm workloads is given in Section 5.6.7.

Table 5.2. Baseline configurations.

System	4-core, 3GHz, out-of-order CPU model based on ALPHA 21264
SRAM* I-L1/D-L1 caches	private, 32KB/32KB, 8-way, 64-Byte cache line, LRU & write-back, write allocate, 2-cycle
ReRAM L2 cache	private, 1MB, 8-way, 64-Byte cache line, LRU & write-back, write allocate, 30-cycle
ReRAM L3 cache	shared, 8MB, 8-way, 64-Byte cache line, LRU & write-back, write allocate, 100-cycle
DRAM main memory	4GB, x16, 8 banks, tRCD-tRP-CL: 11-11-11

5.6.2 Effect of SwS on Inter-Set Variation

The SwS effectiveness in InterV reduction is related to the number of shift rounds (ShiftR). For a cache with N sets, one ShiftR includes N swap rounds (SwapR) and one SwapR has $N-1$ swaps. One ShiftR shifts every cache set through all the possible locations. More ShiftR means better inter-set wear-leveling.

We annotate the round number of ShiftR as RRN,

$$RRN = \frac{W_{total}}{ST \times N \times (N-1)} = \frac{WPI \times I_n}{ST \times N \times (N-1)} \tag{5.8}$$

in which ST is the swap threshold, W_{total} is the product of WPI (write access per instruction) and I_n (the number of simulation instructions). For the same application, if the execution time is longer, which means I_n is larger, we can use a larger ST value to get the same RRN.

To illustrate the relationship between InterV reduction and RRN, we run simulations with different configurations and execution lengths. Figure 5.10 shows the result. As RRN increases, the cache InterV reduces significantly. When RRN is larger than 100, InterV becomes 95% smaller. According to Eqn. 5.8, this means if we want to make an effective inter-set wear-leveling every month, we can use a relaxed ST value (e.g. 100,000 in this case).

However, simulating a system within 1-month wall clock time is never realistic. To evaluate the effectiveness of SwS, we use a smaller ST (e.g. ST=10) in a relatively shorter simulation (e.g. 100 billion instructions) to get a similar RRN.

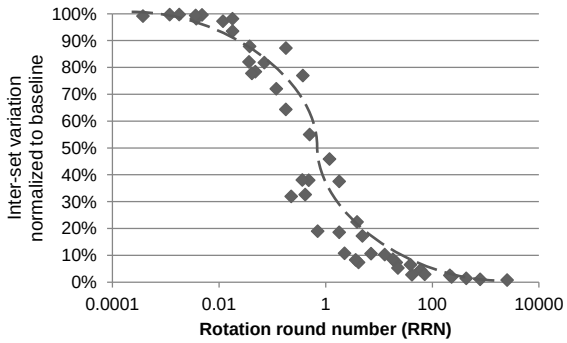

Figure 5.10. Inter-set variations normalized to baseline when RRN increases in SwS scheme. The zoom-in sub-figure shows the detailed L2 inter-set variation of different workloads after adopting the SwS scheme when RRN equals to 100.

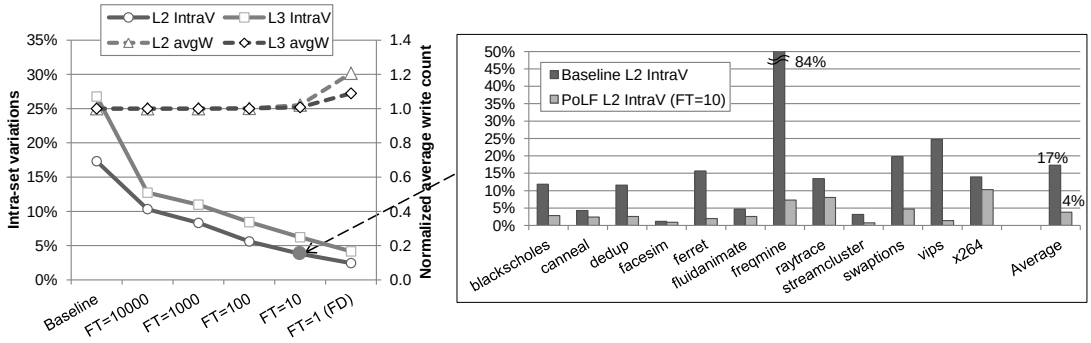

Figure 5.11. The average intra-set variation and the average write count normalized to baseline for L2 and L3 caches after adopting a PoLF scheme. The zoom-in sub-figure shows the detailed L2 intra-set variation for different workloads after adopting an PoLF scheme with line-flush threshold (FT) of 10.

Figure 5.10(b) shows an L2 InterV reduction after adopting SwS when RRN equals to 100. The average InterV is significantly reduced from 66% to 1.2%.

In practice, ST can be scaled along with the entire product lifespan since our wear-leveling goal is to balance the cache line write count in the scale of several months if not years. Thus, the swap operation in SwS is infrequent enough to hide its performance impact.

5.6.3 Effect of PoLF on Intra-Set Variation

Figure 5.11 shows how PoLF affects IntraV and average write counts for L2 and L3 caches. We can see that PoLF can reduce IntraV significantly and the strength

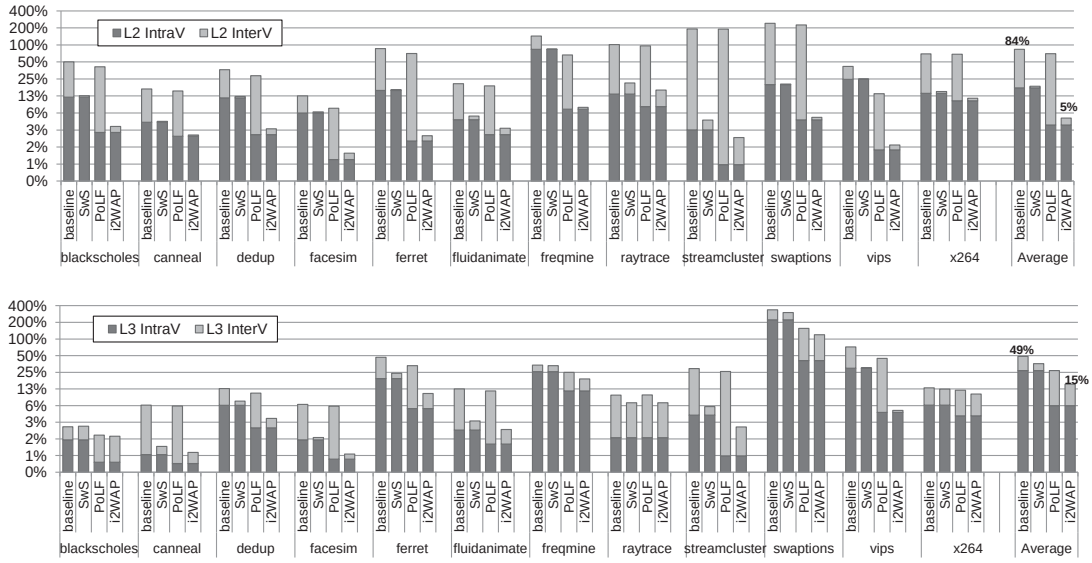

Figure 5.12. The total variation for L2 and L3 caches under the baseline configuration, SwS scheme (RRN=100), PoLF scheme (FT=10) and i^{2}WAP policy. Each value is broken down to InterV and IntraV. Note that a log scale is used to cover a large range of variations.

of PoLF can be changed with different FT values.

When FT equals to 1, the PoLF scheme flushes every write hit and it is equivalent to the LF scheme. Figure 5.11 shows that LF can further reduce IntraV compared to PoLF. However, the average write count of LF is increased significantly as well. Thus, considering the impact on both IntraV and average write counts, we choose PoLF with an FT that equals to 10.

The results show that PoLF reduces the average L2 IntraV from 17% to 4% and the average L3 IntraV from 27% to 6%. The average write count is increased by less than 2% compared to the baseline.

5.6.4 Effect of i^{2}WAP on Total Variation and Lifetime Improvement

Figure 5.12 shows the total variations of L2 and L3 caches under different policies. Compared to the baseline, SwS reduces InterV across all the workloads, but it does not reduce IntraV. On the other hand, PoLF reduces IntraV and has a small impact on InterV. By combining SwS and PoLF, i^2WAP is able to reduce both

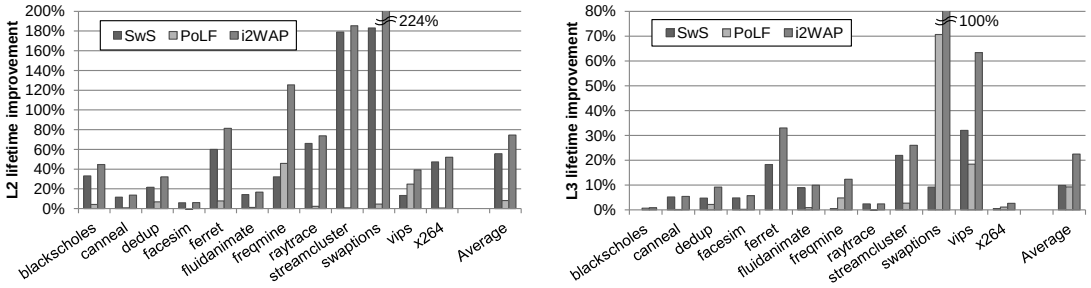

Figure 5.13. The lifetime improvement after adopting i^{2}WAP using Eqn. 5.5. (Left: L2, Right: L3)

InterV and IntraV, evenly distributing writes to every cache line. Figure 5.12 shows that on average the total variation is reduced from 84% to 5% for L2 caches and from 49% to 15% for L3 caches.

Reduced InterV and IntraV mean an improved NVM cache lifetime. Figure 5.13 shows the lifetime improvement of L2 and L3 caches after adopting SwS only, PoLF only, and the combined i^{2}WAP policy, respectively. The lifetime improvement varies based on the workload. Basically, the larger the original variation value is, the bigger the improvement a workload has. The overall lifetime improvement is 75% (up to 224%) for L2 caches and 23% (up to 100%) for L3 caches.

5.6.5 Sensitivity to Cache Associativity

As shown in Table 5.2, we use 8-way associative L2 and L3 caches in the baseline system. To study on the i^{2}WAP effectiveness on different cache configurations, we evaluate its sensitivity to different associativity numbers ranging from 4 to 32. All the other system parameters remain the same.

Figure 5.14 shows the total L2 and L3 variations under different policies when we change the cache associativity. For both L2 and L3 caches in the baseline system, with the increase of the cache associativity, InterV is decreased and IntraV is increased. The reason is that when the cache capacity is fixed, the number of cache sets decreases as the associativity increases. Thus, more writes are merged into one cache set and the write variation from one set to another becomes smaller. Furthermore, IntraV is amplified since the number of cache lines in a set is increased and the intra-set write imbalance becomes worse.

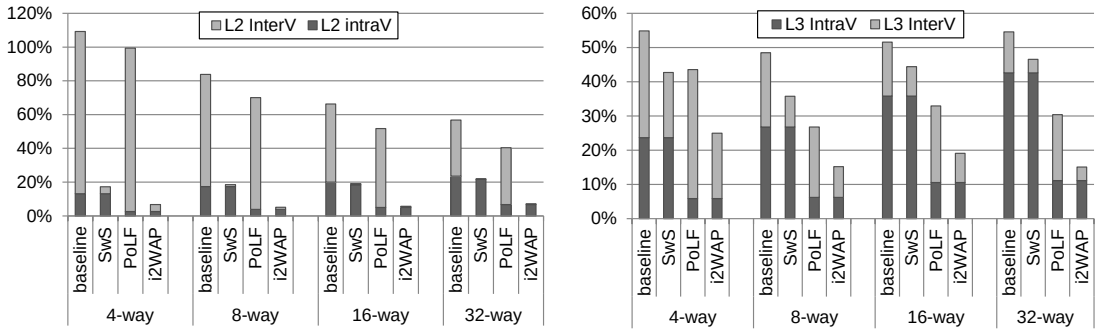

Figure 5.14. The total variation for L2 and L3 caches with 4-, 8-, 16-, and 32-way under the baseline configuration, SwS scheme (RRN=100), PoLF scheme (FT=10) and i^{2}WAP policy. Each value is broken down to the inter-set variation and the intra-set variation.

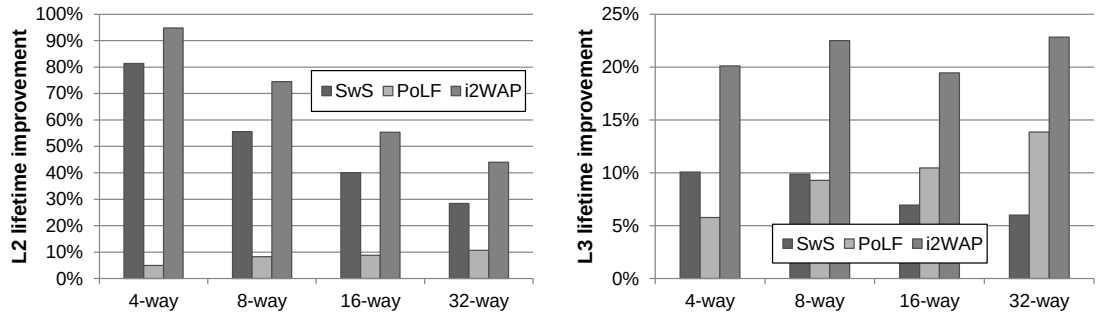

Figure 5.15. The lifetime improvement from i^{2}WAP with different cache associativity. (Left: L2, Right: L3)

Regardless of how the associativity changes, adopting i^{2}WAP reduces the total variations significantly by combining SwS and PoLF. Figure 5.14 shows that for the 4-way, 16-way and 32-way systems, on average the total variation is reduced from 109% to 17%, from 66% to 6%, from 57% to 7% for L2 caches, respectively; for L3 caches, it is reduced from 55% to 25%, from 52% to 19%, from 55% to 15%, respectively.

Accordingly, Figure 5.15 shows the lifetime improvement. On average, the lifetime improvement is 95% and 20% for L2 and L3, respectively, in a 4-way system; it is 55% and 20% for L2 and L3, respectively, in a 8-way system; it is 44% and 23% for L2 and L3, repsectively, in a 16-way system.

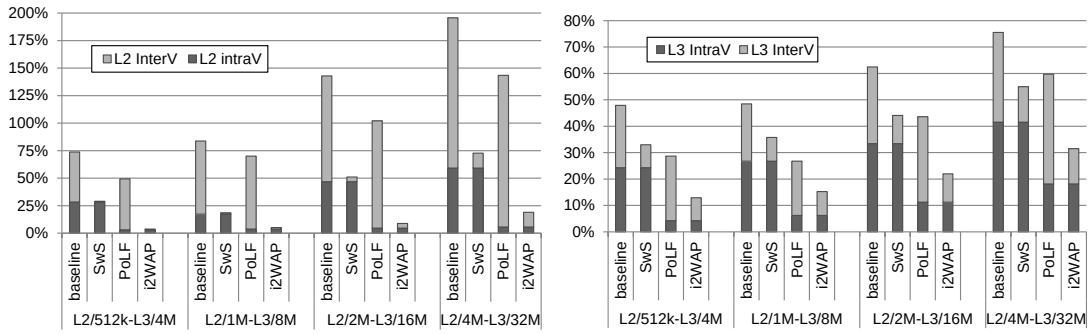

Figure 5.16. The total variation for L2 and L3 caches with different capacities under the baseline configuration, SwS scheme (RRN=100), PoLF scheme (FT=10) and i^{2}WAP policy. Each value is broken down to the inter-set variation and the intra-set variation.

5.6.6 Sensitivity to Cache Capacity

We run another sensitivity study on cache capacities. In Section 5.6, we use 1MB L2 and 8MB L3 caches as shown in Table 5.2. We expect that i^{2}WAP also works effectively on different cache capacities. We conduct experiments on different L2 capacity ranging from 512 kB to 4 MB and different L3 capacity ranging from 4 MB to 32 MB. Figure 5.16 shows the result. On average, the total variation is reduced by 90%-95% for L2 caches and 58%-73% for L3 caches, respectively.

Accordingly, Figure 5.17 shows the lifetime improvement. On average, the lifetime improvement is 66%-153% and 22%-26%, respectively. These results validate that i^{2}WAP works effectively regardless of the cache capacity. For L2 caches, we can see that as capacities increase, the value of lifetime improvement also increases. The reason is that the write imbalance is worse in larger capacity caches causing a larger variation. Thus, i^{2}WAP is more important to large L2 caches. For L3 caches, the variation growth is much flatter than the ones in L2 caches, and IntraV occupies a larger proportion in the baseline. Thus, the L3 cache lifetime improvement is smaller than the one of L2.

5.6.7 Sensitivity to Multi-Program Applications

All the previous simulations are based on multi-thread workloads. To study the i^{2}WAP effectiveness on multi-program applications, we simulate workload mixtures from SPEC CPU2006 benchmark suite [90]. The other simulation configurations

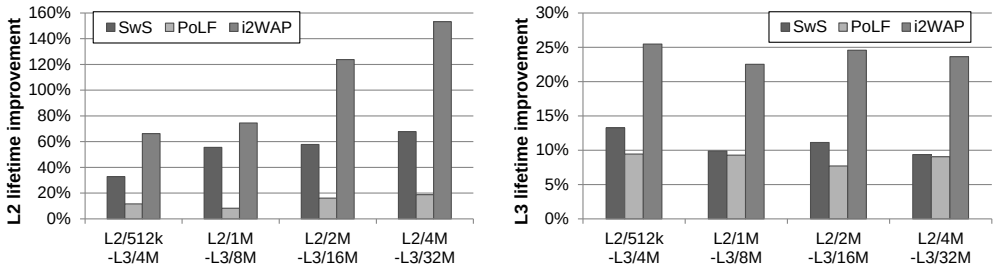

Figure 5.17. The lifetime improvement from i^{2}WAP with different cache capacities. (Left: L2, Right: L3)

remain the same as described in Section 5.6.

Figure 5.18 shows the variations of L2 and L3 caches for multi-program workloads in a 4-core system. Table 5.3 lists the workload mixtures that we synthesize. Intuitively, multiple cores share one L3 caches and run different programs, and the access traffic to the L3 cache should be well mixed and thus balanced. However, Figure 5.18 shows that both InterV and IntraV in the shared L3 cache can still be a problem in some cases. On average, InterV and IntraV of L3 caches are 28% (up to 100%) and 26% (up to 64%), respectively. Similar to the results of multi-thread experiments, the variations in L2 caches is larger than the ones in L3 caches since L2 only serves one program and has more unbalanced write. On average, the total variation is reduced from 132% to 24% for L2 caches and from 54% to 15% for L3 caches, respectively. Figure 5.19 shows the lifetime improvement for the multi-program workloads. For L2 and L3 caches, the overall lifetime improvement is 88% (up to 387%) and 33% (up to 136%), respectively. For the workload mixtures that initially have large InterV, there is a larger space for i^{2}WAP to work and improve the lifetime. On the other hand, IntraV is more difficult to reduce since PoLF is based on a probablistic mechanism, but i^{2}WAP still works fairly well for the workload mixtures that initially have large IntraV.

In general, i^{2}WAP is effective in reducing cache write variations and improving cache lifetime under multi-program workloads.

Table 5.3. The workload list in the mixed groups.

Mixed Group	workloads
Mix 1	astar+bwaves+bzip2+gcc
Mix 2	bzip2+astar+gobmk+h264ref
Mix 3	gromacs+bzip2+gcc+gobmk
Mix 4	gcc+gromacs+hmmer+namd
Mix 5	gobmk+h264ref+hmmer+gromacs
Mix 6	h264ref+hmmer+milc+namd
Mix 7	milc+namd+omnetpp+wrf
Mix 8	namd+h264ref+gcc+astar
Mix 9	omnetpp+wrf+astar+bwaves
Mix 10	wrf+milc+bwaves+gromac

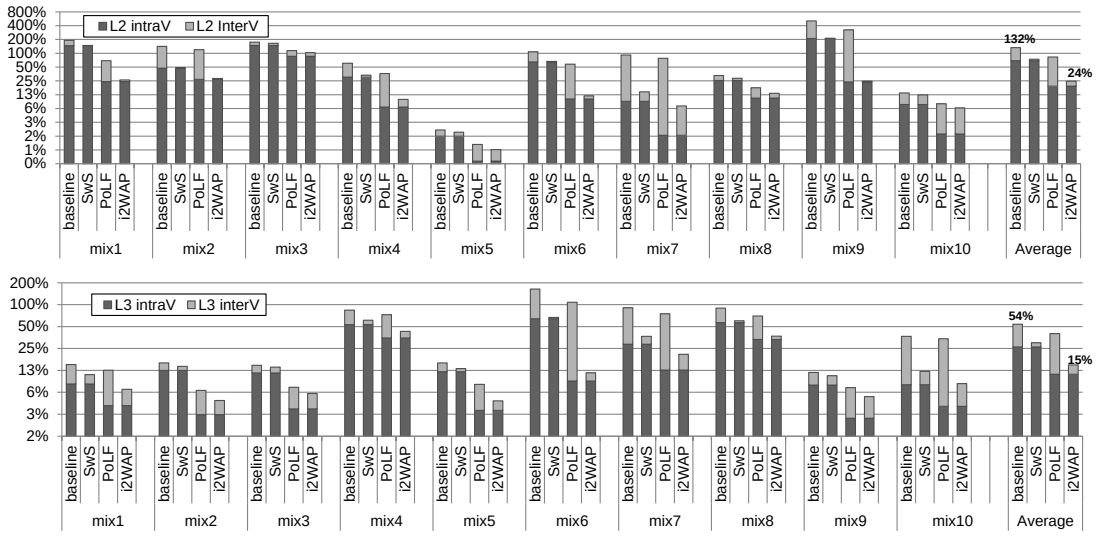

Figure 5.18. The total variation for L2 and L3 caches for multi-program applications using mixed SPEC CPU2006 workloads under the baseline configuration, SwS scheme (RRN=100), PoLF scheme (FT=10) and i^{2}WAP policy. Each value is broken down to the inter-set variation and the intra-set variation.

5.7 Analysis of Other Issues

5.7.1 Performance Overhead and Impact on Main Memory

Since i^{2}WAP causes extra cache invalidations and extra write backs on main memory, it is necessary to compare its performance to a baseline system without wear-

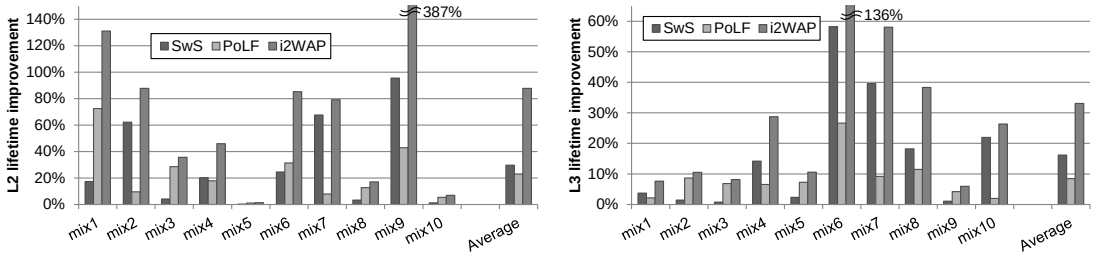

Figure 5.19. The lifetime improvement after adopting i^{2}WAP for multi-program applications using Eqn. 5.5. (Left: L2, Right: L3)

leveling[7]. To model the contention on the main memory bus, we integrate DRAMSim2 [96], in which open-page policy with FR-FCFS [97] scheduling is accurately modeled. DRAM timing information was obtained and modified from Micron datasheets [98].

Figure 5.20 shows the performance overhead of a system in which L2 and L3 caches using i^{2}WAP with $ST = 100,000$ and $FT = 10$ compared to a baseline system in which an LRU policy is adopted. Both of the results of the multi-thread and multi-program workloads are provided. As shown in Figure 5.20, on average, the IPC of the system using i^{2}WAP is reduced only by 0.15% compared to the baseline. The performance penalty of i^{2}WAP is very small because of two reasons:

- In SwS, the interval of swap operations is long (e.g. 10 million instructions), and only two cache sets are re-mapped for each operation.
- In PoLF, write hit accesses are infrequent enough to ensure the frequency of set line flush operations is low (e.g. once per 10^5 instructions), and only one cache line is flushed each time. In addition, designers can trade-off between the number of flush operations and the variation value by adjusting FT.

Figure 5.21 shows the write count to the main memory compared to the baseline system after adopting the i^{2}WAP policy. The result shows that its impact on the write count is very small, only increasing about 1.7% on average. For most workloads, the write count is increased by less than 1%. Thus, the impact on memory bus contention is very small. In addition, because most writes can be

[7]The simulator has a protocol to ensure cache coherency when invalidation occur, thus the performance overhead of this part is included.

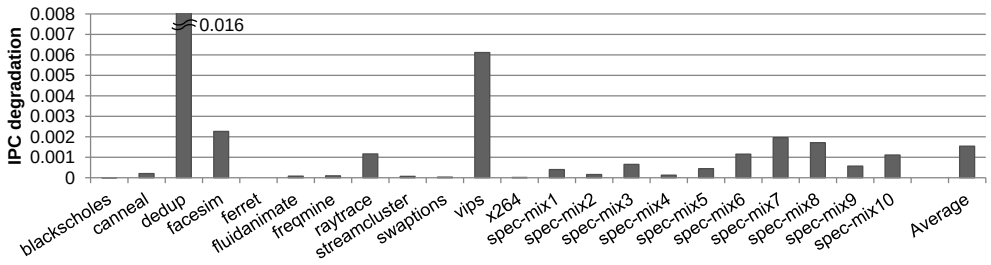

Figure 5.20. The system IPC degradation compared to the baseline system after adopting the i^{2}WAP policy.

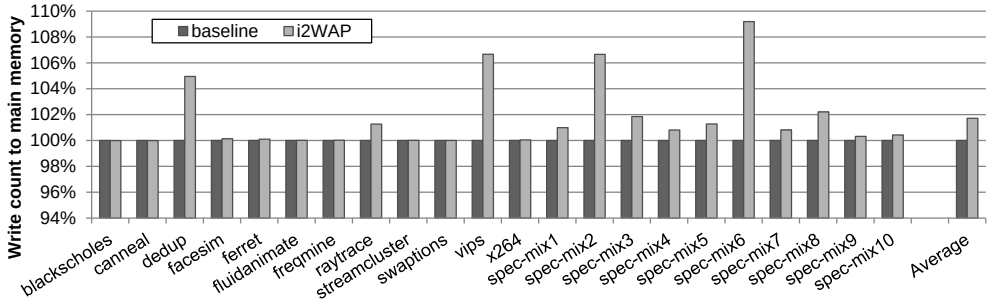

Figure 5.21. The write count to the main memory normalized to the baseline system after adopting i^{2}WAP.

filtered by caches and the write count of main memory is much smaller than that of caches, the endurance requirement for non-volatile main memory is much looser.

5.7.2 Error-Tolerant Lifetime

While our analysis is all focused on the raw cache lifetime, this lifetime can be easily extended by tolerating partial cell failures. There are two factors causing the different failure time of cells. The first one is the variation of write counts, which is addressed mainly in this work. The second one is the inherent variation of the cell's lifetime due to process variations, which needs another type of techniques to solve. For both factors, the system lifetime can be extended by tolerating a small number of cell failures.

It is much simpler to extend i^{2}WAP and tolerate the failed cache lines comparing to tolerating main memory failures [10, 9, 11, 12, 13]. We can force the failed cache lines to be tagged *INVALID*, so that no further data would be written to the failed cache lines. In this case, the number of ways in the corresponding cache

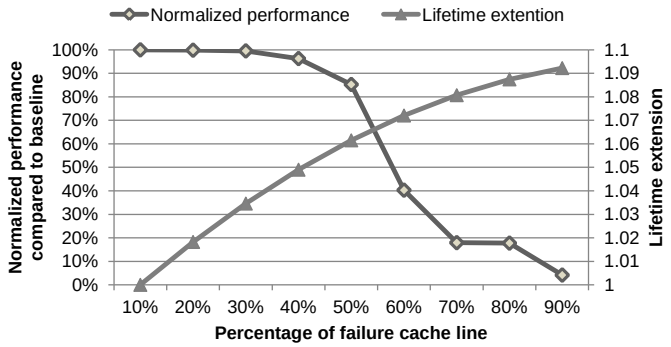

Figure 5.22. The performance degradation and lifetime extension during gradual cache line failure on a non-volatile cache hierarchy.

set is only reduced by 1, e.g. from 8-way associative to 7-way associative.

The error-tolerant lifetime is at least the same as the raw lifetime and may be much longer. However, the performance is degraded because the cache associativity is reduced. Figure 5.22 shows an analysis of an ReRAM-based cache hierarchy with 32KB L1 caches, 1MB L2 caches, and 8MB L3 caches. It shows that if the system can tolerate the failure of 50% of the cache lines at all levels, the lifetime can be extended by 6% and the performance penalty is 15%[8].

5.7.3 Security Threat Analysis

Thus far, we only consider typical workloads. However, memory technologies with limited write endurance always pose a security threat. An adversary might design an attack that stresses few cache lines to reach their endurance limit and then cause a system failure.

One of the common attacks to wear out NVM is Repeated Address Attack (RAA) [8]. Using a simple RAA, an attacker can write a cache line repeatedly and then cause a write endurance failure. It is easy to attack a cache without any endurance-aware policy. A malicious code can attack an L1 data caches by repeatedly write to one address. Since a cache under normal LRU management never replaces a cache line under a cache hit, the same cache line is continuously overwritten under such an attack. The attacking mechanism is similar for L2 or L3 caches. The only difference is that the attacker would use the higher-level

[8]The value of performance degradation and the lifetime extension depend on the cache hierarchy and capacity, but the trends for different configurations are similar.

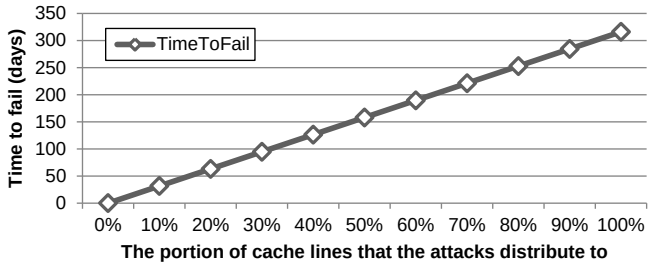

Figure 5.23. The time to fail when the portion of cache lines covered by the distributed attacks is changed.

cache's write-back to repeatedly write the lower-level cache. For example, circularly writing $M + 1$ (M is the cache associativity) of data in one L1 cache set would trigger a repeated write pattern to a single L2 cache line. Assuming it takes 10-50 cycles to write back to one L2 cache line, the time required to make a L2 cache line failure is:

$$Time\ to\ Fail = \frac{Cycle\ per\ write \times Cell\ endurance}{Cycles\ per\ second} = 6 - 30\ minutes \qquad (5.9)$$

Thus, the traditional cache policy opens a serious security problem to NVM caches with limited write endurance.

To defend RAA, we can enhance i^{2}WAP by introducing some randomness. The randomized i^{2}WAP requires several modification:

1. In SwS, we can randomize the swap threshold ST within a pre-determined range. Such randomization makes the mapping relationship between physical and logical set IDs unpredictable, therefore increasing the difficulty of RAA.

2. In PoLF, we can also randomize the line-flush threshold FT within a pre-determined range. Cache line invalidation is quickly triggered by repeatedly write hits. PoLF guarantee a high probability to invalidate the attacked cache line, and data is then loaded in another random location. The threshold randomization makes it more difficult for attackers to predict the new location.

Random number generators have been widely studied [99, 100, 8]. The latency and storage overhead of these generators are small (e.g. 1 cycle delay with 80bytes storage). They can be easily integrated into the i^{2}WAP implementation.

The randomized i^{2}WAP makes the address re-mapping layer unpredictable from outside, and it distributes RAA accesses to different cache lines, thus destroying their repetition feature. Figure 5.23 shows the time to fail for a 1MB L2 cache when the portion of cache lines mapped from the distributed attacks is increased. The time to make a cache line fail can be extended to several months if an RAA access pattern is distributed to more cache lines. Such a long duration is sufficient to detect an abnormal attack.

5.8 Summary

Modern computers require large on-chip caches, but the scalability of traditional SRAM and eDRAM caches is constrained by leakage power and cell density. NVM is a promising alternative to build large on-chip caches. However, NVM usually has limited write endurance, and the existing wear-leveling techniques cannot effectively improve the NVM cache lifetime because caches have IntraV in addition to InterV. In this work, we propose i^{2}WAP, a new endurance-aware cache management policy. i^{2}WAP uses SwS to reduce InterV and PoLF to reduce IntraV thus improving NVM cache lifetime. To implement i^{2}WAP, we only need two global counters and two global registers. In one of our experiments, i^{2}WAP improves the NVM cache lifetime by 75% on average and up to 224%.

Chapter 6

NVM Caches: Hard Error Tolerance

In Chapter 5, we studied on the wear-leveling techniques to balance the write traffic to cache lines, and their basic assumption is all the memory cells have the same quality and the same write endurance. However, this assumption is not true. Due to process imperfection, a more severe problem to non-volatile cache lifetime is that there might be a portion of cells with worse quality and much shorter write endurance. This means the actual lifetime might be much shorter than the expected one even with perfect wear leveling if there is no other error-tolerant technique to be combined and the cell with the worst quality might determine the entire cache lifetime.[1]

State-of-the-art error-tolerant techniques are designed to handle transient soft errors instead of permanent hard errors, such as Error Correcting Codes (ECC). If we use ECC to tolerate permanent wear-out errors, we have to use stronger ECC protection by paying larger hardware overhead. The overhead of *DEC-TED (double-error correction and triple-error detection)* for 64-bit data could be as high as 23%. Moreover, if the bit error count exceeds the ECC protection, we have to discard the affected cache line and waste the healthy bits within it. By doing this, the number of available cache ways in a set might decrease rapidly followed by a quick cache capacity reduction and a significant performance degradation. Therefore, ECC cannot effectively tolerate hard errors from limited write endurance and process imperfection, and new cache architectures for hard error tolerance must

[1]This work is published as "Point and Discard: A Hard-Error-Tolerant Architecture for Non-Volatile Last Level Caches" on DAC2012.

be explored.

Previous work proposed several hard-error-tolerate architecture for non-volatile main memory [10, 11, 12, 13] based on error correction, but the problem is different here since main memory and caches are in different levels of hierarchy. In this chapter, we propose *Point-and-Discard* (PAD), a hard-error-tolerant architecture for non-volatile caches [101]. The principle of PAD is to discard hard errors instead of repairing them. Moreover, it only discards affected bytes instead of the entire cache line when a hard error happens and uses unaffected bytes in the affected cache line to store the locations of affected bytes.

6.1 Error Detection and Distribution Model

Write endurance is defined as the number of times that a memory cell can be overwritten. Limited write endurance is a common problem for non-volatile memories. For example, PCRAM cells are only expected to sustain 10^8 writes before experiencing permanent errors [102]. The write endurance of ReRAM is recently improved but is still at the level of 10^{11} [36]. While STTRAM is usually predicted to have the write endurance of 10^{15}, the current best test result for STTRAM devices is still less than 4×10^{12} cycles [37].

For last-level caches, some of write endurance values (e.g. 10^{11} for ReRAM and 10^{12} for STTRAM) seem to be sufficiently high. However, the real problem is that each cell's lifetime is different due to process imperfection and some weaker cells might wear out much earlier than expected. When a hard error occurs during the runtime of non-volatile caches, a cell would stuck at a value and never change after that. In non-volatile caches, hard errors can be detected by a "read-write-read" pattern for write operations. This mechanism is common in all the non-volatile memory systems to reduce write energy [11]. When there is a write hit, the old data in the cache line is read out at first. Then, only the changed bits are written to the cache line. Finally, the newly-stored data is read out again. If the data is different from the writing data, a hard error is then detected.

In this work, we assume the cell write endurance is under a normal distribution like previous work does [10, 11, 13] and wear-leveling techniques are adopted into the system to get a uniform write distribution across all cache blocks. In the

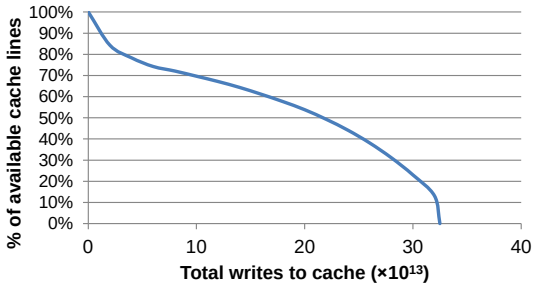

Figure 6.1. The available cache line percentage of a 1MB cache with CoV=0.3 under naive mechanism during runtime.

rest part of this work, we study the non-volatile cache lifetime under a process variation[2] with CoV=0.3. In addition, we assume hard errors are independent and identically distributed.

6.2 Motivation

The simplest method to handle hard errors is to force the affected cache lines tagged as *INVALID*, so that no further data would be written to the cache lines containing hard errors. But, the problem of this mechanism is that even one wear-out cell can cause the waste of the entire cache line, which usually contains 512 bits. In addition, since hard errors are distributed randomly, it is highly possible that almost every cache line has some weak cells. Therefore, the number of unaffected cache lines would rapidly decrease causing significant performance degradation.

We simulate a 1MB ReRAM cache with 8-way associativity and 64-byte cache lines as an example, in which the mean write endurance is 10^{11} and the CoV is 0.3. Figure 6.1 shows the percentage of unaffected cache lines as the write count increases. The result shows the percentage of unaffected cache lines is quickly reduced to 10% only after 3.25×10^{14} writes to the cache. Assuming there are 400 times/second writes to the cache on average, the lifetime is about 1.5 years, which cannot satisfy the requirement of using ReRAM in last-level caches.

This naive mechanism does not work because it over-discards the entire cache line for every single wear-out bit. In order to solve this problem, the basic idea of PAD is to reduce the discard granularity from cache lines to bytes. In PAD, when

[2]CoV is coefficient of variation. A sensitivity analysis of CoV from 0.2 to 0.4 is in Section 6.4.

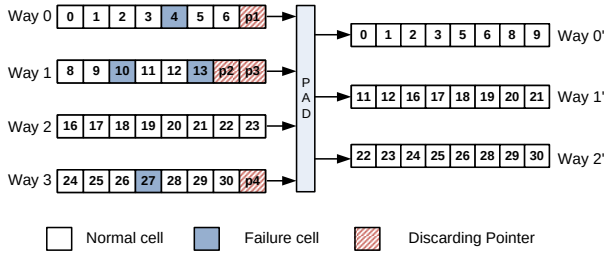

Figure 6.2. A brief example of the PAD architecture.

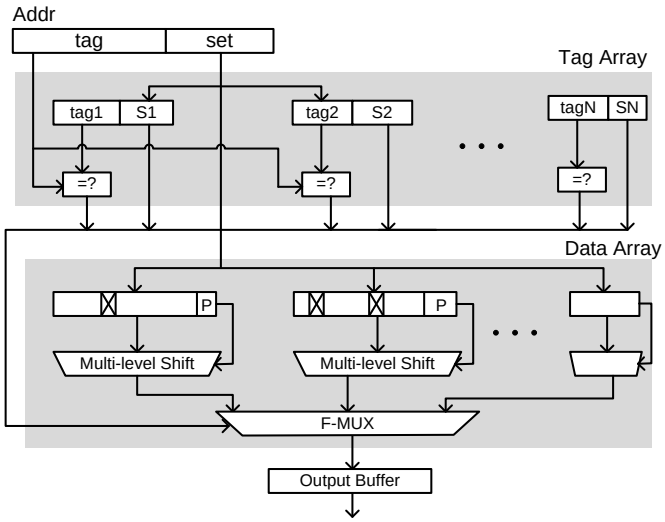

Figure 6.3. The architecture of PAD mechanism.

there is a hard error, only the affected byte is discarded instead of the entire cache line. The remaining healthy bytes from different cache lines are then reorganized to compose new complete cache lines.

Figure 6.2 shows a brief example of the PAD architecture, in which we use a simplified cache structure with 4-way associativity and 8-byte cache lines. The shadow bytes are the ones affected by hard errors. The last few healthy bytes in each way are used to store the positions of the hard-error-affected bytes. By knowing which bytes have hard error, we can use the remaining bytes to compose three complete healthy ways.

6.3 Implementation

In a cache with 64-byte cache lines (Byte[0] to Byte[63]), we allow PAD to discard up to 32 bytes in one cache line. Thus, 5 bits are added to store the number of pointers for each cache line, and their initial values are zeros. Since these cells are only need to be written 31 times at most, it is not possible for them to wear out.

Figure 6.3 shows the diagram of the PAD architecture. In every cache line, if one cell wears out, the address of the affected byte is stored in a pointer and the number of pointers is incremented by 1. Assuming the current number of pointers in a cache line is N, if a new pointer needs to be added, the position of this new pointer is set as Byte[$63 - N$]. In PAD, we use one byte to store one pointer entry although only 6 bits are needed. The remaining 2 bits are used as $VALID\_INFO$. It is set to "00" if this pointer works normally, otherwise it means some bits in this pointer have worn out and this pointer should be ignored. The $VALID\_INFO$ bits themselves are protected by the 2-bit redundancy.

After a byte becomes a pointer, it is impossible for this pointer to wear out because the pointer is read-only once its value is assigned. However, it is possible that the byte to be discarded happens to be the pointer to be designated (i.e. Byte[$63-N$]), which means the new pointer position is just the same as the affected byte. PAD can also tolerate this type of errors: First, PAD uses $VALID\_INFO$ to indicate this pointer is invalid and should be ignored; Second, if $VALID\_INFO$ itself is stuck at "00", another pointer can be used to discard this pointer. In the worst case, if the designated pointer is healthy at the first place but fails at the same time as the one it points to, PAD can invalid it by setting the $VALID\_INFO$ bits and use another pointer. However, it should be noticed that the possibility of such worst-case scenario is extremely low since it is rare for multiple bits to wear out at the same time.

Unlike conventional cache architectures, it is required for PAD to reconstruct healthy bytes into new cache lines when hard errors occur in a cache set. A multi-level shift component is added for this purpose. To access a cache line, data bytes are first loaded into the shift component, then the information stored in the pointers is used to shift out (discard) the hard-error-affected bytes.

For demonstration purpose, the structure of the multi-level shift component

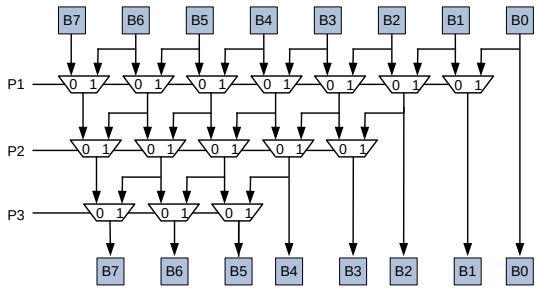

Figure 6.4. Architecture of the single cycle multi-level shifter using 8-byte cache line as an example.

for 8-byte cache lines is shown in Figure 6.4. Every pointer implies to discard one hard-error-affected byte and to shift that byte out. Assuming the hard-error-affected bytes in this example are B6 and B3, then B1 and B0 are used to store the pointers for discarding B6 and B3, respectively. When this cache line is accessed, the pointers stored in B1 and B0 are decoded to control the multiplexer signals on each level. In this example: the control bits on Level 1 are "0111111", and the output data is B7 B5 B4 B3 B2 B1 B0 B0; the control bits on Level 2 are "00011", and the data becomes B7 B5 B4 B2 B1 B1 B0 B0; the control bits on Level 3 are "000", and the final output data is B7 B5 B4 B2. Thus, we remove two hard-error-affected bytes (i.e. B6 and B3) by using two pointer (i.e. B1 and B0) and get the remaining healthy bytes (i.e. B7, B5, B4, and B2).

We call this architecture of multi-level shifter as Single-Cycle-MS since all levels of shifters are connected together and the result can be output in one cycle. However, for conventional 64-byte cache lines, using Single-Cycle-MS architecture brings large area and latency overhead. Thus, we propose an improved architecture called as Multi-Cycle-MS which is shown in Figure 6.5. In each cycle, the data is processed by a n-stage Single-Cycle-MS, in which n is a small number and can be changed by designers according to different target frequencies. In this work, we choose n to be 3. Initially when there is no hard-error-affected bytes, cache lines can be accessed directly and there is no latency overhead. During runtime, when the number of hard-error-affected bytes in one cache line increases, the Multi-Cycle-MS circuit is enabled to reconstruct the correct data by paying latency overhead in an incremental way. Multi-Cycle-MS has two advantages over Single-Cycle-MS:

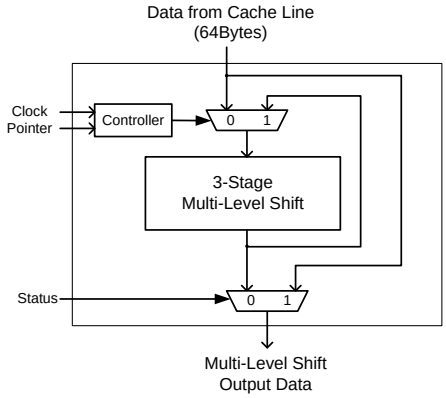

Figure 6.5. Architecture of the multi-cycle multi-level shifter for 64-byte cache lines.

- *Smaller area*: For Single-Cycle-MS, it needs 1023 2-to-1 multiplexers in every multi-level shift. For Multi-Cycle-MS, if we choose n as 3, it only needs 183 2-to-1 multiplexers. It saves about 82.1% areas.

- *Faster data reconstruction speed*: The latency overhead of Single-Cycle-MS is fixed for every cache access and equals to the sum of 31 2-to-1 multiplexers (more than 10 cycles). On the other hand, the latency overhead of Multi-Cycle-MS only increases gradually as the number of hard-error-affected bytes increases. Although the worst-case latency of Multi-Cycle-MS is the same as that of Single-Cycle-MS, the average latency of Multi-Cycle-MS is much shorter. The analysis of the Multi-Cycle-MS latency overhead is in Section 6.4.4.

In PAD, two status bits are added to every cache line to represent 1 out of 4 possible states of the corresponding cache line:

- *Normal*: This cache line does not have any hard-error-affected bytes and can be treated as a normal line, and shift operation is not needed for accessing this cache line.

- *Fixable*: This cache line has some hard-error-affected bytes but can be fixed by using the valid data bytes in the next cache line of the same set.

- *Patch*: All the healthy bytes in this cache line are used to fix the previous *Fixable* line of the same set. *Patch* cache lines never get a hit because they do not store actual data anymore.

- *Discarded*: After the number of the hard-error-affected bytes in a cache line is more than 31, the cache line is set as *Discarded* and is not used anymore.

In every cache access, the result of comparators in the tag array shows which cache line is hit in the set[3]. Besides the tag, PAD stores a *Start* information in the tag. For *Normal* ways, *Start* always equals to 0. For *Fixable* ways, *Start* denotes which byte is the actual first byte in this cache line, which means that the healthy bytes in Byte[0]–Byte[$Start - 1$] are used to shift into previous ways. When the first hard error occurs in a 8-way set, the last cache line in this set is tagged as *Patch* because this set can only store 7 cache lines instead of 8 lines at most. The affected cache line and its following ones are tagged as *Fixable*. When a new hard-error-affected byte occurs in a *Fixable* line, the number of pointers (N) in this cache line is added by 1. If *Start* equals to $64 - N$, which means all healthy bytes in this cache line are used to fix the previous line, then the status of this cache line is changed from *Fixable* to *Patch*. During runtime, more ways are tagged to be *Patch* as more weak cells wear out, and the percentage of *Normal* and *Fixable* ways in the set is reduced.

The hit and *Start* information is loaded into a final MUX (F-MUX) to get the output data. F-MUX outputs the data based on the status of the hit cache line, which must be *Normal* or *Fixable* since hits never occur to *Patch* and *Discarded* lines. If the status of the hit cache line is *Normal*, F-MUX chooses this line and outputs the complete cache line. If its status is *Fixable*, F-MUX chooses this *Fixable* line and all the following lines until the next *Fixable* or *Normal* line or the last line of this set. Assuming *Start* of a hit *fixable* line is S_n, then F-MUX would choose all the bytes from Byte[S_n] to Byte[63] in this line, and the remaining bytes from the followed *Patched* or *Fixable* ways. This part is easy to implement since the Multi-Cycle-MS has already discarded the hard-error-affected bytes and F-MUX only needs to assemble them together[4].

[3]For caches, the size of tag array is very small compared to the data array. Thus we can use simple redundant encoding to tolerate the hard errors in tag cells.

[4]Although we focus on reads in the discussion, reads and writes are symmetric. Also, Data-Comparison Write [103] is adopted to avoid the unnecessary writes.

Table 6.1. Experiment settings

Set Number	2048
Way Number	8
Cache Line Size	64 Byte
Cache Size	1MB
Mean Cell Lifetime	10^{11}
Lifetime Variation	0.3

6.4 Experimental Results

In this section, we describe our experiment methodology and evaluate the lifetime improvement and the overhead after applying PAD.

6.4.1 Experiment Methodology

While it is intractable to simulate the cache behavior over its entire lifetime, we follow the same assumptions used in previous work [10, 11], in which we assume that existing wear-leveling techniques already spread writes evenly over cache lines and every write randomly hits one cache line. In addition, the bit flipping probability is assumed to be 0.5. In this work, we use ReRAM caches as the analysis target. For each simulation run, the simulator assigns a random lifetime to each cell using a normal distribution with a mean of 10^{11} writes [36]. Our experiment tracks the number of available cache lines after each write operation. If a new hard error occurs, the available cache line count is recalculated.

We use the architectural parameters shown in Table 6.1. The baseline architecture is set to a naive cache architecture in which an entire hard-error-affected cache line (i.e. 64 bytes) has to be disabled for a single hard error. To evaluate the effectiveness of PAD, we also simulate the cache architecture adopted 64-bit *Single Error Correction* (SEC_{64}) which is a common type of ECC in today's memory. SEC_{64} corrects up to one error bit for each 64-bit data blocks by adding 7 check bits. The storage overhead of SEC_{64} is 7 bits per 64-bit data (11%).

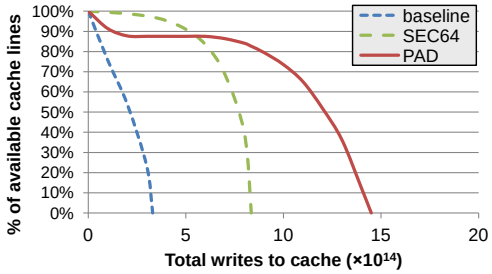

Figure 6.6. The percentage of available lines of a 1MB cache with CoV=0.3 in PAD compared to baseline and ECC (SEC64).

6.4.2 Lifetime Improvement

Figure 6.6 shows the percentage of available cache lines (i.e. *Normal* and *Fixable* cache lines) in PAD compared to baseline and SEC_{64}. Under the assumed write endurance CoV of 0.3, the percentage of available cache lines in baseline quickly decreases to 10% only after 3.25×10^{14} writes to the cache. On the other hand, after adopting PAD architecture, the percentage of available cache lines is reduced to 10% after 1.41×10^{15} writes. Assuming there are 400 times/second writes to the cache on average, the lifetime is about 1.5 years for the baseline system and about 6.8 years after adopting PAD. Thus, PAD can improve the simulated ReRAM cache lifetime by 4.6X.

For SEC_{64}, the percentage of available cache lines is higher than PAD in the system's early life. It is because SEC_{64} pays much larger storage overhead (11% compared to 1.4%) to correct the wear-out bits while PAD uses unaffected bytes to fix the affected lines. However, PAD has larger error-tolerant capability than SEC_{64}. The result shows that PAD improves the cache lifetime by 1.7X compared to SEC_{64}.

6.4.3 Sensitivity Analysis of CoV

It is expected that the lifetime of cache architecture is very sensitive to the write endurance distribution. Therefore, we use the same model to simulate a 1MB ReRAM cache with different CoV (higher CoV means more severe process variation, and lower means less). Figure 6.7 and Figure 6.8 show the percentage of available cache lines in PAD compared to baseline and SEC_{64} systems when CoV

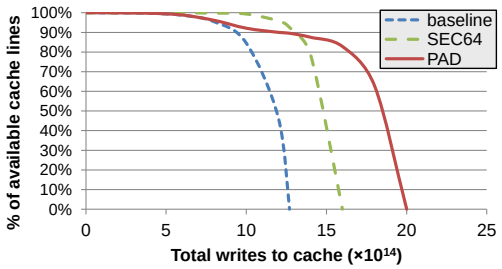

Figure 6.7. The percentage of available lines of a 1MB cache with CoV=0.2 in PAD compared to baseline and ECC (SEC64).

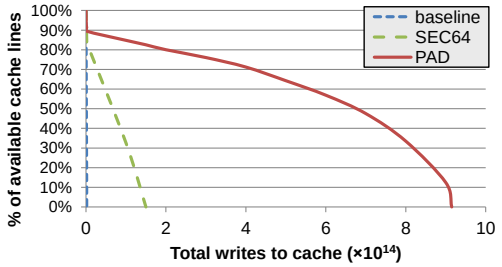

Figure 6.8. The percentage of available lines of a 1MB cache with CoV=0.4 in PAD compared to baseline and ECC (SEC64).

is 0.2 and 0.4, respectively. The results show the 1MB ReRAM cache lifetime is increased by 1.6X and 440X compared to baseline, respectively. Compared to SEC_{64}, PAD can also get 1.3X and 6.1X lifetime improvement. It implies that PAD is more effective under more severe process variation, which means it would be more useful as the process node advances.

6.4.4 Overhead Analysis

PAD has the capability of tolerating hard errors. However, it is also important to evaluate the area and performance overhead of PAD when we deploy it in low-power high-density non-volatile caches.

For each cache line, PAD only needs to add a 5-bit counter (for pointers) and a 2-bit status label. Thus, the storage overhead is about 1.4% for a 64-byte cache line. The major area overhead comes from the multi-level shifters. For a 8-way cache, it needs 8 multi-level shifters, and there are 183 2-to-1 multiplexers in every Multi-Cycle-MS. Thus, the overall area overhead is 1,464 2-to-1 multiplexers. Compared to common error-tolerant schemes, such as SEC_{64}, which has 11% storage overhead

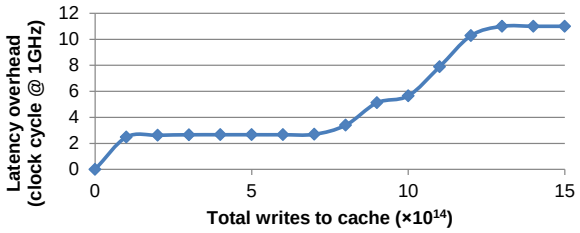

Figure 6.9. The read latency of PAD for a 1MB ReRAM cache. It is small in the early life and increases during the running time.

and additional encoder/decoder, PAD has much smaller overhead.

The latency overhead of PAD is zero in the system's early life when there is no hard error in the cache. But, the latency increases over time when more cells wear out because Multi-Cycle-MS needs more cycles to reconstruct a valid data. Figure 6.9 shows the simulation result of the latency overhead. It shows that the access latency overhead is smaller than 3 cycles during half of the cache's lifespan, and the maximum latency is 11 cycles which only occurs near the end of the lifetime.

6.4.5 Performance Analysis

We use Instructions-Per-Cycle (IPC) to compare the performance degradation between PAD and the other cache architectures. In baseline and SEC_{64}, there is no latency overhead[5] but the percentage of available cache lines is reduced rapidly. On the other hand, the latency overhead is increased gradually in PAD, but the percentage of available lines reduces much more slowly as shown in Figure 6.6.

We used the gem5 simulator for our performance evaluations [87]. Figure 6.10 shows the normalized IPC of PAD compared to baseline and SEC_{64} of simulating EEMBC 2.0 benchmark [91]. As more write operation conducted to caches, Figure 6.10 shows the baseline performance is degraded quickly from 100% to 50% after only 3×10^{14} writes. For PAD, the write count is about 1.4×10^{15} when IPC degrades 50%, which is increased by 4.7X. Compared to SEC_{64}, the IPC of PAD is reduced slightly (smaller than 2%) in the cache's early life. However, due to the higher error-toleration, PAD achieves better performance than SEC_{64} with

[5]We use zero latency overhead for SEC_{64} in the simulation. However, depending on the implementation, the encoder and decoder could bring some latency overhead.

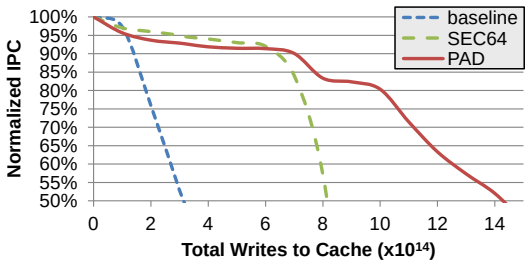

Figure 6.10. The IPC degradation of PAD architecture compared to baseline and ECC (SEC64).

the number of writes increasing. Therefore, compared to baseline and SEC_{64}, PAD architecture can improve the performance of non-volatile caches with longer lifetime.

6.5 Summary

Adopting non-volatile memory technologies in last-level caches is attractive because they has lower energy consumption and higher density compared to the traditional SRAM and embedded DRAM technologies. However, limited write endurance is one obstacle before adopting non-volatile caches widely. Although the write endurance values of some non-volatile memories seem sufficient for last-level caches, a more severe problem is lifetime variations due to process imperfection. It causes the actual cache lifetime much shorter than expectation because the weakest cell might determine the overall lifetime. However, the current error-tolerant techniques for caches are designed to handle transient faults and cannot effectively tolerate hard errors in non-volatile caches.

This work presents *Point-and-Discard* (PAD), a hard-error-tolerant architecture for non-volatile caches. PAD has better error-tolerant capacity than some common schemes (such as SEC_{64}) with smaller storage overhead. The experimental results show that PAD improves the lifetime of non-volatile caches by 1.6X-440X under different process variations. Moreover, PAD incurs no performance overhead in the system's early life and ensures a gradual performance degradation as the cells continuously wear out.

Chapter 7

NVM Caches: Write Optimizations

In addition to the limited write endurance issue discussed in Chapter 5 and Chapter 6, another problem of NVM caches is expensive write operations. In this chapter, we use STTRAM last-level caches as an example to show the impact of expensive writes on system performance and propose our solutions.[1]

We define a type of process characteristic called LLC-obstructive. Processes with this characteristic can cause significant performance degradation in STTRAM LLC based system, and they also negatively affect the performance of other processes running in parallel. We design an obstruction-aware monitoring mechanism (OAM) and an obstruction-aware cache management (OAP), which differentiates the accesses generated by LLC-obstructive processes from others [104, 105]. OAP can significantly improve system performance and reduces energy consumption for STTRAM last-level cache design.

7.1 Background and Motivation

In this section, we briefly review the STTRAM LLC, and use a motivational example to demonstrate why the port obstruction problem can cause huge performance loss.

[1] This work is published as "OAP: An Obstruction-Aware Cache Management Policy for STTRAM Last-Level Caches" on DATE2013.

Table 7.1. Characteristics of 8MB SRAM and STTRAM caches (32nm).

Memory type	SRAM	STTRAM
Cell factor (F^2)	146	40
Read latency (ns)	11.1	11.4
Write latency (ns)	11.1	22.5
Read energy (pJ)	15.8	17.5
Write energy (pJ)	13	172
Leakage power (mW)	14.1	0.14
Area (mm^2)	11.5	3.16

7.1.1 Using STTRAM LLCs

Compared to conventional SRAM cache, the advantages of STTRAM cache are smaller area and lower leakage power. The cell size of STTRAM is currently in the range of $13F^2$ [106] to $100F^2$ [41] where F is the feature size. This cell size is much smaller than the SRAM size (e.g. $146F^2$ [78]). The small cell size of STTRAM can greatly reduce the silicon area occupied by caches. In addition, STTRAM has zero leakage power consumption from memory cells due to its non-volatility. Previous work [35] shows the leakage energy can be as high as 80% of total energy consumption for an L2 cache in 130nm process. Thus, using STTRAM last-level caches to eliminate standby leakage power is an attractive choice.

Table 7.1 shows the characteristics of a 4-bank 8-way 8MB SRAM cache and its STTRAM counterpart. The estimation is given by NVSim [79], a performance, energy, and area model based on CACTI [78]. It shows that STTRAM cache has much smaller area and leakage power than the one of SRAM. However, as also shown in Table 7.1, the write latency of STTRAM is around 2X longer than SRAM. This long write latency becomes a sensitive parameter in STTRAM L3 cache and it causes the L3 cache port obstruction problem.

7.1.2 Motivation 1: Port Obstruction

As the last-level cache, L3 is usually implemented using single-port memory bitcell due to the infeasible cost associated with multi-port bitcell designs. For example, building dual-port STTRAM cell requires at least 4 transistor [107] (a 4X larger cell layout area), which is not affordable in large-capacity L3 designs. For single-

Table 7.2. Cache hierarchy of 3 different systems.

System name	Cache hierarchy
2-LC	4-core, private SRAM 32KB L1, SRAM 256KB L2
3-LC-SRAM	2-LC + SRAM 8MB shared L3
3-LC-STTRAM	2-LC + STTRAM 8MB shared L3

port STTRAM caches, an ongoing write operation can cause a long L3 cache port obstruction and delay all the following read operations that are on the critical path from the performance aspect. When the number of pending read operations exceeds the number of MSHR entries of the requester (e.g. CPU or upper-level cache) [108], the system must stall. Because of this port obstruction, while some workloads can benefit from the larger capacity of STTRAM LLCs, it is possible that the system performance is not improved but degraded when running some write-intensive workloads.

A common practice to hide write latency is to enlarge the write buffer size. Write buffer can hold the incoming writing data and fill them into the destination cache line only when the cache becomes idle or the write buffer reaches high water mark. Write buffering is usually effective in traditional SRAM LLCs since SRAM write speed is sufficiently fast to drain all the write buffer entries during the cache idle period. However, write buffering becomes less effective for STTRAM cache as its write is too long to hide. In addition, the practical write buffer size is limited by design complexity and fully-associative look-up overhead.

To quantify this write-induced port obstruction problem, we simulate 3 different systems as configured according to Table 7.2 (see Section 7.3 for details on our simulation methodology): one with only 2 levels of caches, one with an additional SRAM L3 cache, and one with an additional STTRAM L3 cache. Figure 7.1 shows the normalized IPC comparison among these 3 systems. Compared to *2-LC*, the performance of *3-LC-SRAM* is improved by 3% on average, but the performance of *3-LC-STTRAM* is degraded 14% on average (up to 35%). While adding STTRAM L3 cache improves the performance for some workloads (e.g. *hmmer*), it might heavily degrade the performance for write-intensive workloads (e.g. *lbm*) as well.

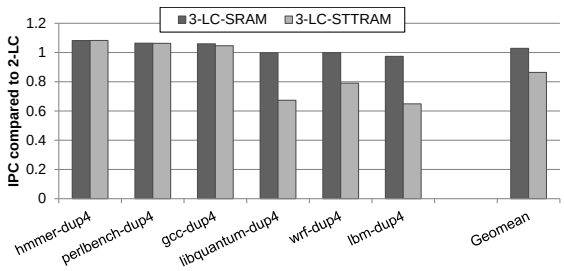

Figure 7.1. Compared to the system *2-LC*, the normalized IPC of *3-LC-SRAM* and *3-LC-STTRAM* using 4-duplicate workloads.

7.1.3 Motivation 2: Process Interference

Besides the performance degradation caused by a workload's own write intensity, we must notice that the LLC is commonly a shared resource, and one process that obstructs the cache port can block the normal cache accesses generated by other concurrent processes in a multi-core system.

To illustrate this problem, we simulate 4 different workloads running simultaneously on a 4-core system. We use two different mixed group: *mix1* and *mix2*. Figure 7.2 shows the normalized IPC of each workload in these two mixed groups, and the names of running workloads are also listed.

In our experiment design, we assign 4 different workloads to Group 1, and copy the same configuration to Group 2 but intentionally replacing the workload running on the first core with a write-intensive workload *cactusADM* in Group 2. The result shows that all the workload IPCs in Group 1 are only reduced by 1%. However, on contrary, not only the IPC of the first core but all the IPCs in Group 2 are degrades significantly (10%-25%). This demonstrates that the write-intensive process not only causes performance loss to itself but also interferes the normal cache accesses initiated by other processes.

Based on these observations, we define a type of process characteristic called "LLC-obstructive". If a workload is LLC-obstructive, the performance of itself can be degraded significantly by adding an STTRAM LLC, and worse, it might also affect the performance of other workloads in a negative way.

Therefore, we need a new STTRAM L3 cache management policy, and its task is to first identify any potential LLC-obstructive processes on-the-fly and then avoid the performance degradation caused by such LLC-obstructive processes.

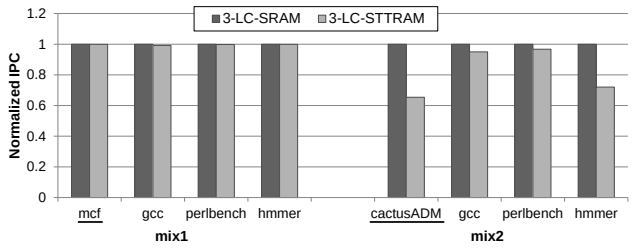

Figure 7.2. The normalized IPC of *3-LC-SRAM* and *3-LC-STTRAM* of two workload groups: the first core runs different workloads; the other three run the same workloads.

7.2 Obstruction-Aware Cache Management

In this chapter, our basic concept is to enhance the existing cache management policy so that it can detect those LCC-obstructive processes, block their cache accesses, and reserve the LLC capacity for well-behaved processes. In this section, we describe how to implement such a policy.

7.2.1 A Naive Approach: Using Static Threshold

To design an obstruction-aware cache management policy, the most critical problem is how to find LLC-obstructive processes during runtime. We start from a relatively straightforward method called "static threshold obstruction-aware management policy" (SOAP), which uses a predetermined threshold to determine whether a process is LLC-obstructive. Based on our motivational experiments in Section 7.1, we can observe that LLC-obstructive processes commonly have two features:

1. **High utilization**: The LLC port occupancy is high. It is usually manifested in the respect of high TPKI (transactions per kilo instruction). Given a certain read/write ratio, write operations obstruct the LLC port most of the time. This causes the delay of read requests from itself and other concurrent processes.

2. **High miss rate**: The cache miss rate is high. The consequence is that most of the data written into the LLC will are useless. Thus, both the time and the energy spent on those LLC writes are wasted.

If one process has these two characteristics, it means that this process introduces heavy writes to the LLC but most of them are unnecessary. This type of processes obstruct the LLC and degrade the system performance.

Thus, we can set two static thresholds, $Util_{th}$ and $MissR_{th}$, monitor the cache access statistics of each process, and determine which one is likely to be LLC-obstructive. First, we can define the utilization of each process according to Equation 7.1,

$$Utilization = \frac{N_{RD} \times T_{Rd} + N_{WR} \times T_{Wr}}{Execution\,Time} \tag{7.1}$$

where N_{RD} and N_{WR} are the read and the write counts, respectively, and T_{Rd} and T_{Wr} represent the latencies of the LLC read and the write operation, respectively. If the calculated value is greater than the threshold $Util_{th}$, this workload is labeled as *high utilization*.

Similarly, we can also monitor the cache miss rate of a process. If it is greater than $MissR_{th}$, we then label that process as *high miss rate*. If one process is both *high utilization* and *high miss rate*, we treat it as a potential *LLC-obstructive* process.

It is relatively simple to implement the SOAP idea, but SOAP has its disadvantages. First, it is highly possible that misjudgment can happen since we only use two simplified metrics. For example, given the criteria used in SOAP, such a "LLC-obstructive" process might not actually block the LLC port for a long time if most of its cache accesses are reads. Though it is possible to overcome this problem by adding more metrics and conditions (e.g. read/write ratio), the scheme itself only becomes marginally better but with more and more hardware complexity.

Second, although the value of $Util_{th}$ and $MissR_{th}$ can be adjusted upfront by the architects, the working thresholds for different processes vary from one case to another, and it becomes very impractical to predetermine a set of universal values. To show the impact of selecting different threshold values, we show a sensitivity study on these two parameters. The details of simulation methodology are described in Section 7.3. Figure 7.3 shows how the performance speedup of SOAP[2] varies when we change the value of $MissR_{th}$ from 0.05 to 0.95 only. This

[2]Each speedup value is the average result among all the benchmarks

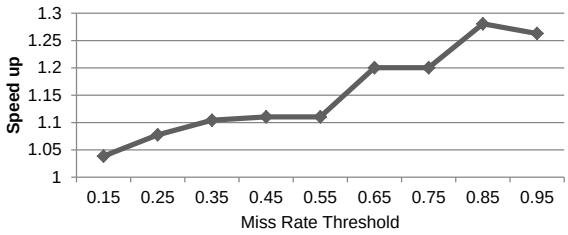

Figure 7.3. The performance speedup from SOAP and changing $MissR_{th}$ from 0.05 to 0.95.

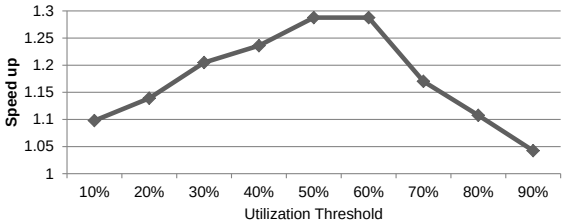

Figure 7.4. The performance speedup from SOAP and changing the value of $Util_{th}$ from 10% to 90%.

result shows that the speedup can be small if this threshold is set too low. It is because some normal processes would be incorrectly labeled as LLC-obstructive when $MissR_{th}$ is low. Their performance drops since they are blocked by the cache management policy and cannot benefit from the existence of the LLC.

On contrary, Figure 7.4 shows the speedup of SOAP when $Util_{th}$ is changed from 10% to 90%. This result shows that the speedup can be small if $Util_{th}$ is either too low or too high. If $Util_{th}$ is too low, normal processes can be incorrectly treated as LLC-obstructive; vice versa, if $Util_{th}$ is too high, LLC-obstructive processes cannot be effectively captured. We see a sub-optimal system performance in both of these two cases.

In short, the performance speedup of SOAP is very sensitive to the setting of these thresholds (i.e. $Util_{th}$ and $MissR_{th}$), and it is difficult to predefine the optimal values. Therefore, we stop discussing the details of SOAP any further, instead we focus the rest of this chapter on an improved solution called dynamic obstruction-aware management policy (OAP), which does not require any predefined thresholds.

7.2.2 A More Practical Approach: Using Heuristic Metric

To make OAP adaptive to different workloads, a more practical way is to use heuristic metrics. The rationale behind adding another level of cache (e.g. L3) into the memory hierarchy is to provide fast-access memory resources so that workloads with good spacial and temporal locality can leverage these memory resources and avoid the time-consuming access to the off-chip DRAM main memory. However, not all the workloads can benefit from this feature.

Assuming that the behavior of a process does not change within a short period of time, if we write data of this process into the LLC, the expected execution time for one of following accesses can be estimated as Equation 7.2,

$$T = P_{Rd} \times T_{Rd} + (1 - P_{Rd}) \times T_{Wr} + P_{Miss} \times (T_{Mem} + T_{Wr}) \tag{7.2}$$

where P_{Rd} is the LLC read/write ratio, P_{Miss} is the LLC miss rate. In addition, T_{Rd}, T_{Wr}, T_{Mem} are the latency of LLC read, LLC write, and the average latency of main memory, respectively. Note that we assume that this cache uses *fetch-on-write* policy, a widely-used policy in modern cache designs.

For each cache miss, new data need to be fetched from main memory and be written to LLC at first. Thus, the additional latency is added to the total delay T.

On the other hand, if the data of this process are not written into LLC, then the following operations have to access main memory directly. Thus, the expected execution time of one access can be estimated as:

$$T' = T_{Mem} \tag{7.3}$$

If writing the data from one process into the LLC cannot get any benefit in terms of system performance, which means $T > T'$, then we can get:

$$P_{Miss} > \frac{T_{Mem} - P_{Rd} \times T_{Rd} - (1 - P_{Rd}) \times T_{Wr}}{T_{Mem} + T_{Wr}} \tag{7.4}$$

When the process characteristic satisfies Equation 7.4 during a period of time, it might cause intensive write operations to the LLC so that writing its data into LLC

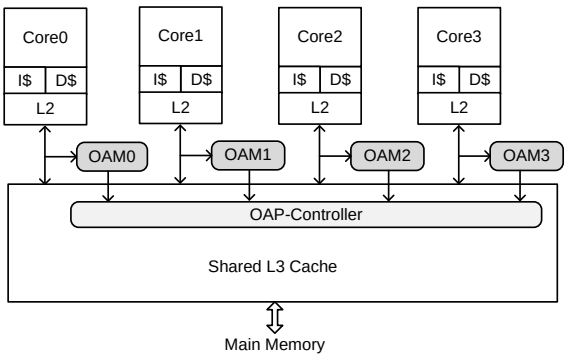

Figure 7.5. A 3-level cache hierarchy enhanced by OAP. Newly added structures are highlighted in gray.

will only extend the execution time and hence degrade the system performance. We then can define this type of processes as *LLC-obstructive processes*.

We can also observe from Equation 7.4 that the possibility of LLC obstruction is higher when an STTRAM LLC is used because the miss rate threshold becomes smaller when the LLC write latency is longer. This is also the reason why the performance degradation is more severe when we use STTRAM LLCs instead of their SRAM counterparts.

7.2.3 LLC Obstruction Monitor

To detect LLC-obstructive processes during runtime, we design an obstruction-aware monitor (OAM) and places it before LLC (i.e. between the L2 and L3 caches). The OAM circuit is separated from the cache hierarchy. This allows OAM to independently obtain the L3 cache access information from each core.

An adjustable parameter *period* is available in OAM, and it presents the timing granularity of the LLC-obstructive detection. Every *period* is further divided to two parts:

1. *sampPeriod*, in which the cache works under the normal policy, and OAMs collect cache access statistics;

2. *exePeriod*, in which L3 is managed by OAP using the detection result collected during the last sample period.

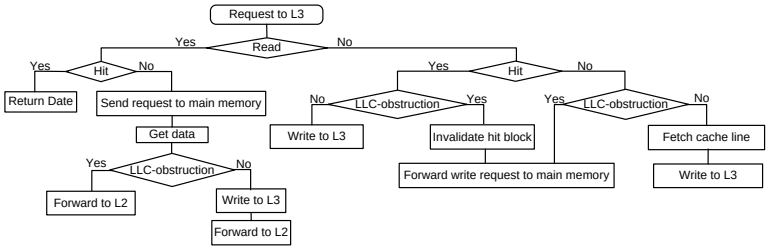

Figure 7.6. The control flow of an OAP controller. The "LLC obstruction" detection is made by the OAM associated with each core.

In *sampPeriod*, all processes are labeled as *Non-LLC-obstructive*. A per-core OAM collects the statistics including: the execution time *currentTime*, the number of read accesses *RD*, the number of write accesses *WR*, and the number of cache misses *Miss*.

At the end of *sampPeriod*, OAM evaluates two metrics: the actual miss rate *MissR* and the obstruction threshold OAP_{th}.

$$MissR = Miss/(RD + WR) \tag{7.5}$$

$$OAP_{th} = \frac{T_{Mem} - (RD \cdot T_{Rd} + WR \cdot T_{Wr})/(RD + WR)}{T_{Mem} + T_{Wr}} \tag{7.6}$$

According to Equation 7.4, if *MissR* is larger than OAP_{th}, it means that the process is not benefiting from LLC. Therefore, OAM labels this process as an *LLC-obstructive* process. Otherwise, OAM labels it as a *Non-LLC-obstructive* one. During the following *exePeriod*, depending on whether OAM considers this process is LLC-obstructive or not, OAP handles the cache accesses from this process accordingly until the next *period* starts. The monitor algorithm can be described in Algorithm 3.

7.2.4 OAP Architecture

Figure 7.5 shows a 4-core system with a 3-level cache hierarchy enhanced by OAP. Similar to previous work [109, 110], we assume processes are bound to cores. Each core has its own OAM to track L3 cache accesses requested by its private L2 cache and find out the LLC-obstructive processes. After the potential LLC-obstructive processes are detected, an OAP-controller in the shared L3 cache is responsible

Algorithm 3 The algorithm of runtime OAM

Input: The new *access* sent to LLC.
Output: The detection result.
Parameters: *period*, *sampPeriod*, $Util_{th}$, $MissR_{th}$.

```
1: if currentTime − startTime == period then
2:    reset all parameters
3:    label as Non-LLC-obstructive
4:    startTime ← currentTime
5: end if
6: if currentTime − startTime < sampPeriod then
7:    if accessIsRead then
8:       RD++ // Update read access count
9:    else
10:      WR++ // Update write access count
11:   end if
12:   if accessIsMiss then
13:      Miss++ // Update miss count
14:   end if
15: end if
16: if currentTime − startTime == sampPeriod then
17:   Calculate OAP_th, MissR
18:   if MissR > OAP_th then
19:      label as LLC-obstructive
20:   else
21:      label as Non-LLC-obstructive
22:   end if
23: end if
```

to prevent LLC-obstructive processes from accessing L3 cache and also make sure data coherence. The control flow of the OAP-controller is shown as Figure 7.6, and its functionality is described as follows:

- *L3 read hits*: The L3 cache returns the data to the corresponding L2 cache.
- *L3 read misses*: The L3 cache asks the data from the main memory at first. When the main memory returns the data, the OAP-controller checks weather the read requester is LLC-obstructive. If it is, the returning data bypasses the L3 and is directly forwarded to the L2. Otherwise, the data is written into the L3 as normal.
- *L3 write hits*: The OAP-controller checks if the write requester is LLC-

obstructive at first. If it is, the L3 invalidates the hit data block, and the write request bypasses the L3 and is directly forwarded to the main memory. Otherwise, the data is written into the L3 as normal.

- *L3 write misses*: At first, the OAP-controller checks if the requester is LLC-obstructive. If it is, the write request bypasses the L3 and is forwarded to the main memory directly. Otherwise, the cache line is fetched and allocated at first and then the new data is written into the L3 cache.

Note that we assume the modification is applied on the top of a non-inclusive or exclusive cache management work flow. If we need to implement OAP atop of an inclusive LLC, we should include other techniques, such as adding a bypass buffer [111] to implement cache bypassing.

7.3 Experimental Results

In this section, we describe our experiment methodology, and we evaluate the performance improvement and the energy reduction achieved by OAP.

7.3.1 Experiment Methodology

We model a 1.5GHz 4-core out-of-order ARMv7 microprocessor using a modified version of gem5 [87]. Our modification to gem5 includes an asymmetric cache read/write latency model, a banked cache model, and a sophisticated cache write buffer scheme. Write buffers are commonly used in conventional cache architecture to hide the performance penalty due to write operations, but the write buffer size cannot be too large because of its fully-associative look-up overhead [17]. Thus, the write buffer technique alone is not sufficient to deal with the performance degradation due to the STTRAM long write latency. We use an 8-entry write buffer in this work.

Simulation setting details are listed in Table 7.3. All the circuit-level cache module parameters (e.g. read latency and write latency) are obtained from NVSim [79] and are consistent with Table 7.1. The simulation workloads are from SPEC CPU2006 benchmark suite [90]. Workloads are compiled using gcc version 4.5.2

Table 7.3. Simulation settings.

Core		4-core, 1.5GHz out-of-order ARM cores
SRAM I-L1/D-L1 caches		private, 32KB/32KB, 16-way, LRU, 64B cache line, write-back, write allocate 2-cycle read, 2-cycle write
SRAM L2 cache		private, 256kB, 8-way, LRU, 64B cache line, write-back, write allocate 8-cycle read, 8-cycle write
L3 cache	Common	shared, 8MB, 8-way, LRU, 4 banks, 8-entry write buffer per bank, 64B cache line, write-back, write allocate
	OAP STTRAM	17-cycle read, 34-cycle write, w/ OAP
	Baseline STTRAM	17-cycle read, 34-cycle write, w/o OAP
	Baseline SRAM	17-cycle read, 17-cycle write
DRAM main memory		4GB, 128-entry write buffer, 200-cycle

(Sourcery G++ Lite 2011.03-41) without any optimization setting. We simulate each experiment for at least 2 billion instructions. We set the OAM sampling period to be 0.1 million cycles and OAM launching period to be 10 million cycles.

We use IPC (instructions per cycle) as the performance metric and use geometric mean to average the performance of cores and derive the speedup metric ($speedup = geomean(\frac{IPC_i}{IPC_i^{baseline}})$) which accounts for both fairness and performance [110].

7.3.2 Performance Speedup

Figure 7.7 illustrates the speedup over conventional STTRAM L3 based system after adopting the OAP architecture. The first 8 groups run 4 duplicated processes, and the other 8 groups are workload mixtures as listed in Table 7.4. It shows that after adopting OAP on the STTRAM L3 cache, the system performance is improved by 14% on average (up to 42%) compared to the conventional cache management policy.

Furthermore, we list several characteristics over workloads in Table 7.5: 1) the percentage of reads and writes to L3 cache; 2) the miss rate of L3; 3) the port utilization calculated as Eqn 7.1. In order to understand where the performance

Table 7.4. The workload list in the mixed groups.

Mixed group	workloads
mix1	lbm+libquantum+sjeng+cactusADM
mix2	povray+mcf+namd+lbm
mix3	bwaves+cactusADM+astar+hmmer
mix4	gcc+lbm+libquantum+bwaves
mix5	cactusADM+gcc+mcf+lbm
mix6	libquantum+bzip2+bwaves+cactusADM
mix7	lbm+sjeng+bzip2+libquantum
mix8	mcf+bwaves+sjeng+lbm

saving comes from, the frequency analysis for the different paths outlined in the control flow of the OAP controller can be calculated based on the data in Table 7.5. Generally, the workloads with high write miss rate and high port utilization are more likely to be detected as LLC-obstructive processes, and the groups which have more LLC-obstructive processes in more periods benefit more from OAP architecture. Because OAM can quickly detect such processes during sampling periods, and OAP-controller skips their unnecessary writes to the STTRAM L3 cache.

For groups with mixed workload, the performance improvement comes from two respects. First, the performance of LLC-obstructive processes is improved because OAP skips the unnecessary writes in these processes and mitigates the penalty coming from the longer write latency. Second, the performance of concurrent processes is also improved since the obstruction on the L3 port is reduced and their requests to L3 can be satisfied more quickly. In addition, the available cache lines of L3 is increased for these well-behaviored processes because the number of write operations from LLC-obstructive processes is reduced, thus it increases their hit rate and performance.

7.3.3 Compare to SRAM LLC

Adopting OAP architecture in L3 cache makes STTRAM more competitive compared to SRAM. To evaluate its benefits, we simulate another baseline system with a shared SRAM L3 cache, which is configured as Table 7.3.

Figure 7.8 shows the normalized IPC of the systems with normal SRAM L3,

Table 7.5. The characteristics of each workload.

Workloads	L3 read%	L3 write%	L3 miss rate	L3 utilization
bwaves-dup4	32%	68%	42%	0.49
bzip2-dup4	39%	61%	36%	0.30
cactusADM-dup4	42%	58%	95%	0.44
libquantum-dup4	60%	40%	99%	0.48
gcc-dup4	74%	26%	73%	0.09
lbm-dup4	23%	77%	89%	0.63
mcf-dup4	72%	28%	74%	0.34
sjeng-dup4	46%	54%	94%	0.16
mix1	43%	57%	96%	0.56
mix2	44%	56%	69%	0.28
mix3	36%	64%	73%	0.37
mix4	42%	58%	78%	0.57
mix5	45%	55%	80%	0.46
mix6	42%	58%	73%	0.51
mix7	43%	57%	83%	0.47
mix8	40%	60%	76%	0.45

STTRAM L3 and OAP STTRAM L3. The result shows that compared to the SRAM L3 based system, a straightforward STTRAM L3 replacement degrades the system performance by 13.2% on average due to its longer write latency. However, after adopting OAP, the performance of STTRAM L3 based system can be improved significantly. Compared to SRAM L3, the performance degradation in OAP STTRAM based system is only 0.7% on average.

Besides the performance improvement achieved by OAP, using STTRAM L3 itself can save energy as well. This is because the leakage power is dominant in L3 cache and the leakage power of STTRAM is only about 1% of SRAM's. In addition, adopting OAP can reduce the total energy consumption further because it reduces the number of writes, which is a major source of the dynamic energy in STTRAM caches. Figure 7.9 shows the normalized energy consumption of 3 different systems and each value is broken down to leakage energy and dynamic energy. Due to the leakage energy reduction, an 8Mb STTRAM L3 can save around 90% total energy compared to an SRAM L3 cache with the same capacity. And after adopting OAP architecture, the energy of STTRAM L3 is further reduced by 64% on average.

The final advantage of STTRAM L3 cache is the silicon area reduction. It is

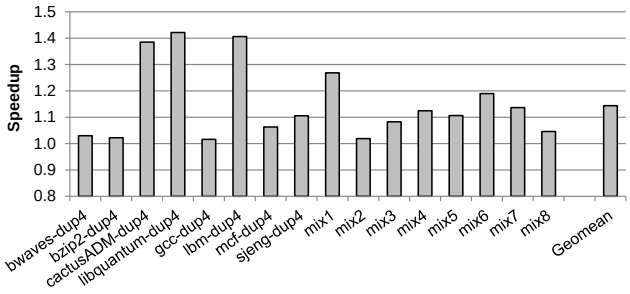

Figure 7.7. The performance speedup after adopting OAP with duplicated and mixed workloads in a 4-core system: *benchmark-dup4* means executing *benchmark* on each core.

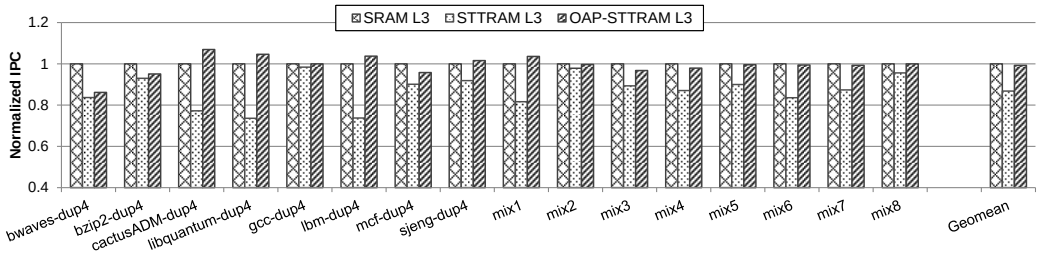

Figure 7.8. The IPC of systems with SRAM L3, conventional STTRAM L3 and OAP STTRAM L3 (normalized to the value of SRAM L3 based system).

not related to OAP but also important. Compared to an 8MB SRAM cache, an 8MB STTRAM cache can save 72.5% area as shown in Table 7.1.

In summary, STTRAM L3 enhanced by OAP makes itself a more attractive option in replacing SRAM L3 cache.

7.4 Overhead Analysis

7.4.1 Hardware Overhead

We evaluate the hardware overhead of the proposed OAP architecture using Synopsys Design Compiler with a 32nm CMOS library. According to the synthesis result, the OAM circuitry only incurs $0.05mm^2$ area overhead, which is negligible compared to the L3 cache area (usually half of the total chip area). According to the energy analysis, the extra energy consumption per access is about 1.7pJ, which is included in our simulations. In addition, it is straightforward to integrate the OAP judgment flow into the conventional cache controller using bypass circuits.

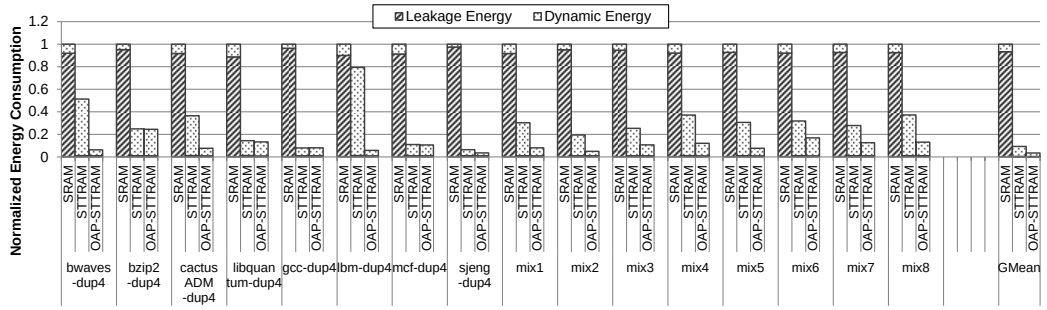

Figure 7.9. The energy consumption of SRAM L3, conventional STTRAM L3 and OAP STTRAM L3 (normalized to the value of SRAM L3). Each value is broken down to leakage energy and dynamic energy.

In terms of latency overhead, considering OAMs are separated from the cache hierarchy and OAP-controller only brings some branch decisions in the control flow, we do not add any latency overhead in our simulation settings.

7.4.2 Impact on L3 Miss Rate

OAP avoids the writes to STTRAM L3 from LLC-obstructive workloads, but it might increase the miss rate since the data of LLC-obstructive processes are not allocated in L3. If L3 miss rate increases significantly, the system performance might be degraded because more accesses to the main memory are needed. However, the OAP scheme is designed to detect LLC-obstructive processes automatically which usually have high miss rate. Figure 7.10 shows the normalized L3 miss rate of an OAP system compared to the baseline. It shows the overall miss rate only increases by 6%. Thus, the OAP scheme has small impact on system miss rates.

7.4.3 Traffic Overhead Analysis

We also need to evaluate the traffic overhead since some writes are bypassed from LLC in the OAP system. Figure 7.11 and Figure 7.12 show the main memory traffic and the L3 traffic of OAP system normalized to the baseline system, respectively.

From Figure 7.11, we can observe that OAP increases the overall read and write accesses to main memory by 39% and 10%, respectively. Although the traffic to the main memory is increased, the number of write accesses to L3 is reduced by 62% on average as shown in Figure 7.12. Therefore, combined these two factors,

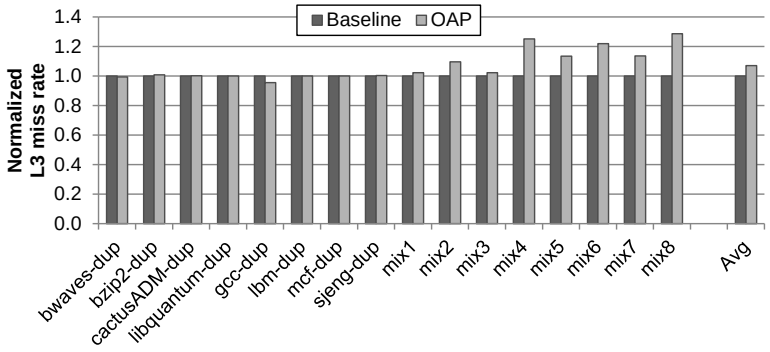

Figure 7.10. The L3 miss rate comparison between a baseline system and an OAP system.

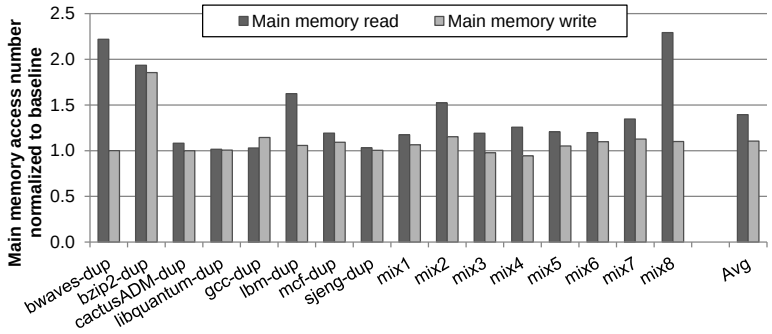

Figure 7.11. Read and write access numbers of the main memory normalized to the baseline system.

the performance of the entire system is increased after adopting OAP since the port obstruction possibility becomes lower, and the write latency of the STTRAM L3 cache is long.

7.5 Sensitivity Study

In this section, we describe our experiments and results to study the performance sensitivity in terms of different write latencies, bank numbers, core numbers, and the period length of OAM when applying OAP technique.

7.5.1 Sensitivity to Cache Write Latency

In Section 7.3, we use the cache configuration shown in Table 7.3, in which the LLC latency is consistent with Table 7.1. We expect that OAP also works effec-

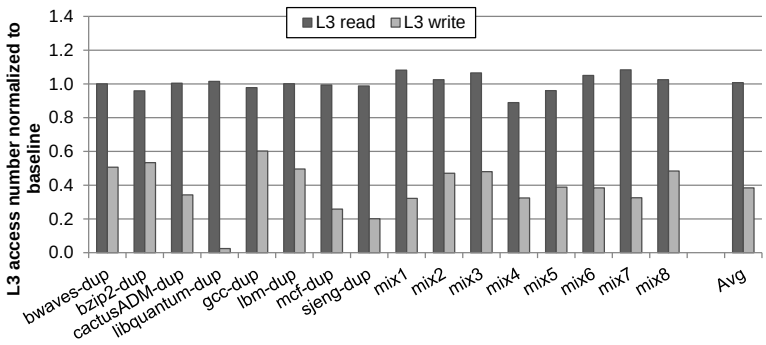

Figure 7.12. Read and write access number of L3 cache normalized to baseline system.

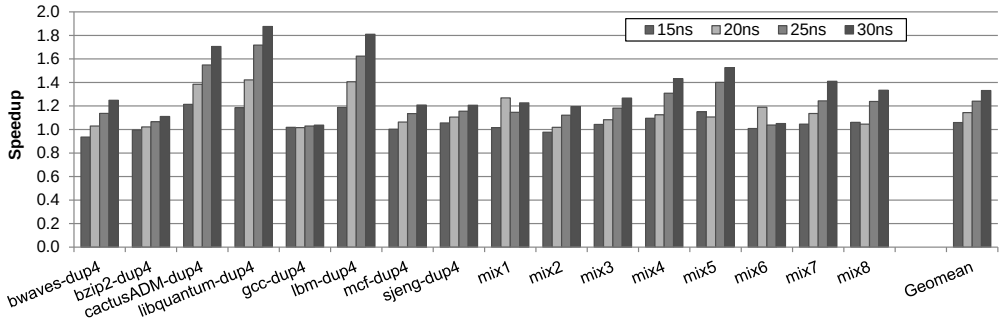

Figure 7.13. The performance speedup after adopting OAP with different cache write latencies.

tively on caches with different cache configurations and can get more benefits when increasing the STTRAM LLC latency.

We run experiments on different LLC write latencies ranging from 15ns to 30ns. Figure 7.13 shows the results. Among all the workloads, the performance speedup is 1.06 on average and up to 1.21 when the LLC write latency is reduced to 15ns. On the other hand, the effectiveness of OAP is more significant when we increase the LLC latency. The system speedup is increased to 1.24 and 1.33 on average when the LLC write latency is increased to 25ns and 30ns, respectively.

Generally, adopting the OAP technique gets more performance benefit when the STTRAM LLC latency is longer. The reason is that the port obstruction problem is more severe when L3 write latency is longer. An ongoing write from one core could have a higher chance to block other reads to the same cache bank. The OAP technique improves the system performance by detecting LLC-obstructive processes and skipping their unnecessary writes to the STTRAM L3 cache.

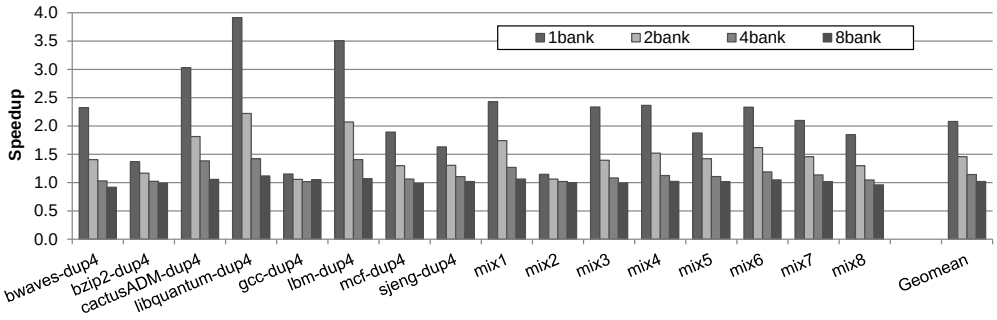

Figure 7.14. The performance speedup after adopting OAP architecture with different numbers of L3 banks.

7.5.2 Sensitivity to Bank Number

Large on-chip caches are usually split to multiple banks to increase access parallelism. If the LLC has more banks, the possibility of port obstruction will be reduced since an ongoing write operation only blocks the port of the corresponding cache bank and does not effect the port of other cache banks. To study on the effect of LLC bank number, we adopt OAP in systems with different L3 configurations as 1 bank, 2 banks, 4 banks and 8 banks, respectively. All the other system parameters are consistent with the ones in Table 7.3.

Figure 7.14 shows the performance speedup result. Among all workloads, the speedup is as high as 2.08 and 1.46 on average when L3 cache is configured as a 1-bank and 2-bank system, respectively. And the average speedup is 1.02 when the bank number is increased to 8.

Therefore, the system with smaller LLC bank number can get more performance improvement by adopting OAP technique. The reason is that a multi-bank organization provides more parallelism and mitigates the port obstruction problem. However, adding bank number is expensive in terms of cost. Thus, adopting OAP to improve the performance for the system with smaller bank number is a better solution.

7.5.3 Sensitivity to Core Number

Another sensitivity study is targeted on the core number. The simulation result given in Section 7.3 is focused on a 4-core system, but OAP can also be adopted

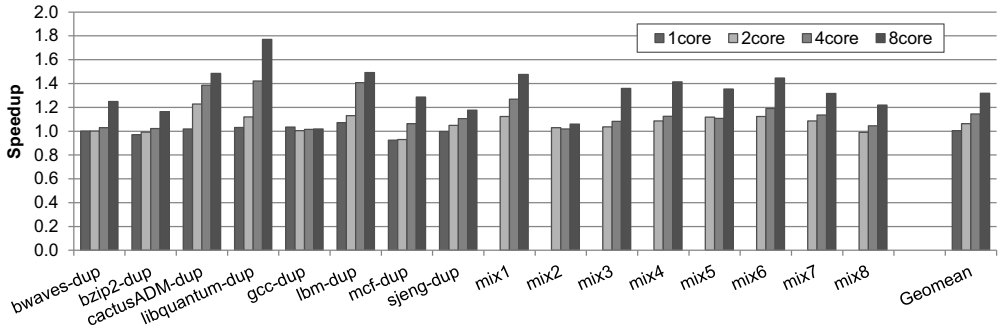

Figure 7.15. The performance speedup after adopting OAP architecture with different numbers of cores.

in single-core or multi-core systems with different core numbers.

We run simulations on systems with 1 core, 2 cores and 8 cores. Other system configurations are not changed and shown in Table 7.3. For the single-core system, we only execute one program without using the mixed groups. For the 2-core system, we use the first two programs in each mixed group as shown in Table 7.4. For the 8-core system, we duplicate each program to two cores for each mixed group.

Figure 7.15 shows the speedup results. The speedup of the single core system is 1.01 on average and up to 1.07. This speedup is not as significant as the 4-core system because there is no negative affect between different applications, and the obstruction problem is not as severe as in multi-core systems.

For the 2-core and 8-core systems, the overall speedup is 1.06 and 1.32 , respectively. It shows that when the core number is increased, the problem of LLC port obstruction is more and more severe since there are more processes sharing the same port of LLC. Thus, adopting OAP technique can get more performance improvement with the core number increasing.

7.5.4 Sensitivity to the Length of Launching Period and Sampling Period

In addition, we run sensitivity study on the length of launching period and sampling period. If the sampling period is too long compared to the launching period, the running time using OAP technique is short and the effectiveness of OAP is reduced. On the other hand, if the sampling period is too short, the LLC-obstructive pro-

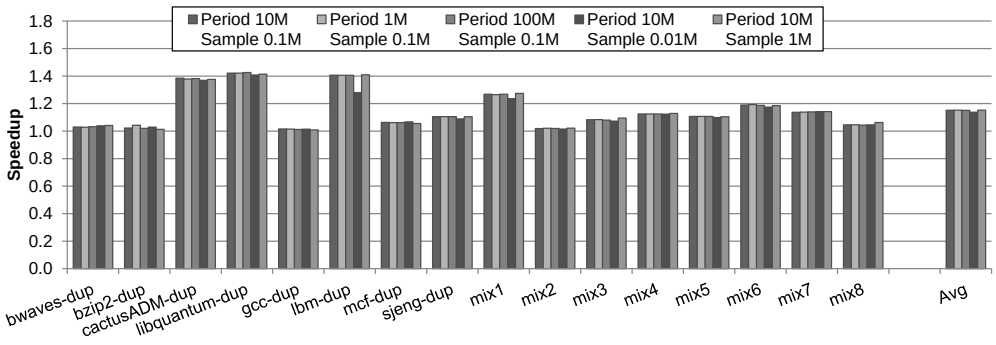

Figure 7.16. The performance speedup after adopting OAP architecture with different launching sampling period length.

cesses might not be captured by OAM and the performance speedup is decreased.

Figure 7.16 shows the performance speedup of OAP architecture with different length of launching period and sampling period. It shows that the overall performance speedup is reduced to 13% when the sampling period is too small (i.e. the launching period is set as 10M cycle and the sampling period is set as 0.01M cycle).

7.6 Summary

While the scaling of SRAM and eDRAM is constrained by cell density and leakage energy, the emerging STTRAM memory technology is explored as one of the potential alternatives of last-level caches. Despite STTRAM has higher density and smaller leakage energy, STTRAM has much longer write latency compared to SRAM, and may cause severe performance degradation in a multi-core system where cache accesses are obstructed by the write operations. In this chapter, we proposed OAP, an obstruction-aware cache management policy, which can significantly improve system performance and reduces energy consumption across different workloads for future STTRAM last-level cache design.

Chapter 8

NVM GPGPU: Write-Aware Register Files

Besides main memories and caches, NVMs are also explored as other memory levels, such as register files in general-purpose computing on graphics processing units (GPGPU). Recently, to enable the acceleration on parallel data processing, GPGPU is studied and developed rapidly. By utilizing traditional GPUs, GPGPUs are targeted for non-graphics workloads which demand high computational performance. In industry, Nvidia and AMD have already released many GPGPU products [112, 113].[1]

Different from CPUs, GPGPUs can achieve teraFLOP peak performance by exploiting thread-level parallelism. For example, Nvidia GTX680 GPU provides 3090 GFLOPS/s with 1536 cores [114]. This high performance of GPGPUs comes from the large number of parallel threads executed in a SIMD (Single-Instruction Multiple-Data) fashion. In addition, GPGPUs typically implement a hardware-based context switch among threads to hide the long latency of memory operations. In order to achieve this, GPGPUs have a large number of registers to hold states and contents of all the active threads. For instance, each streaming multiprocessor (SM) in the Nvidia Fermi architecture is equipped with a 128KB register file (RF) which supports up to 1,536 active threads [112]. Compared to the RF in CPUs (e.g., 2KB in total for a 64-bit CPU with 128 int/128 FP registers), the

[1]This work will be published as "A Write-Aware STTRAM-Based Register File Architecture for GPGPUs" on JETC.

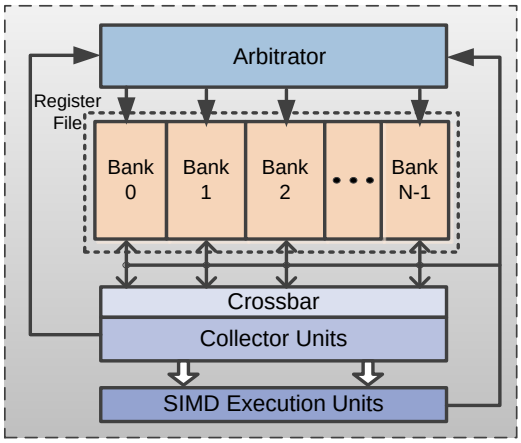

Figure 8.1. The architecture of operand collectors.

RF in GPGPUs is much larger (e.g., 2MB in total for the top-tier Fermi chip). Therefore, the area cost and the energy consumption of RFs should be carefully evaluated in GPGPU design.

Traditionally, SRAM is used to build the RFs in GPGPUs, but it faces challenges in terms of density and leakage power. Thus, STTRAM is being studied as one of the potential alternatives for SRAM to mitigate these two issues. However, as discussed in Chapter 7, the expensive write operations of STTRAM might degrade the system performance and increase the dynamic energy consumption. To solve these two issues, we propose a sophisticated Write-AwaRe Register File design (**WarRF**) in this chapter, which makes STTRAM a more attractive option for RF design in future GPGPU systems.

8.1 Background

8.1.1 GPGPU and Register File Architecture

GPGPUs usually consist of many small cores, and each core includes multiple data processing lanes (e.g. 32 in Fermi), an L1 data/instruction cache, a shared memory, a register file, multiple schedulers, and multiple computing units [112]. There are thousands of threads processing simultaneously in the GPU core to fully utilize the parallelism. The number of threads is determined by the hardware resource.

GPGPUs issue threads in groups, and we call each group a warp (e.g. 32

threads in one warp in Fermi). All the threads in one warp are executed in a SIMD fashion. At each clock cycle, the instruction scheduler selects a ready warp for execution and issues it to the data path. Another special GPGPU feature is to switch from one warp to another without any latency. Because of this, GPGPUs can hide long latency memory operations and keep their pipelines busy even if some warps are stalled. In order to implement this instant context switch, each thread allocates some dedicated registers from the RF. This is the reason why GPGPU requires huge RFs, and GPU RF size is orders of magnitude larger than CPU RF size. For example, each stream multiprocessor (SM) in the Nvidia Fermi architecture has 32,768 32-bit registers, and the RF size can be as large as 2MB for the entire GPGPU chip [115]. This RF size still keeps increasing over product generations to support more threads and provide higher performance.

Moreover, GPGPU RF must support multiple memory operations simultaneously. To achieve this, Nvidia uses a banked RF architecture combing multiple single ported RF to avoid the big hardware overhead of using multi-port memory array. The design of operand collectors is further devised to distribute accesses to different RF banks and avoid bank conflicts since one bank can only handle one access at each cycle [116].

Figure 8.1 shows the detailed architecture of operand collectors. When an instruction is received from the decode stage, an available collector unit is allocated to store the register identifier, operand data, etc. Meanwhile the source operand read requests are queued in the arbitrator. The arbitrator contains read queues for each register bank. It selects a group of non-conflicting accesses and sends them to RF at every cycle. As each operand is read out of the register file and placed into the corresponding collector unit, it is marked as ready. Finally, when all the operands are ready, the instruction is issued to a SIMD execution unit. In addition, each execution unit and memory port has an independent issue port from the operand collector to avoid the stall due to shared writebacks.

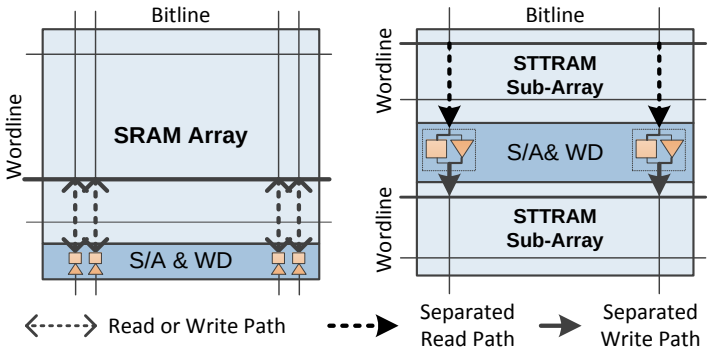

Figure 8.2. The layouts of SRAM and STTRAM arrays. In STTRAM array, separated read and write paths to different sub-arrays can be operated in parallel.

8.2 Split Bank Write

8.2.1 Motivation

For SRAM-based RFs, read and write accesses have similar latency and both can be finished in one cycle. Thus, the arbitrator sends a group of accesses to RF at each cycle. But, for STTRAM-based RFs, write access has much longer latency than read, and it needs multiple cycles to finish. Therefore, when a write access is operated in one RF bank, the read/write accesses to the same bank need to wait until the write operation is finished. If all the requests in the queues conflict with the current writes, the pipeline has to stall.

To improve the parallelism of read and write accesses to RFs, we explore an inherent feature of STTRAM memory array circuitry. SRAM commonly uses voltage sense amplifiers which partially share circuit with write drivers. On contrary, STTRAM uses current sensing, which is generally more complicated and requires larger sense amplifiers. In addition, write drivers and sense amplifiers are separated entirely. As shown in Figure 8.2, different from SRAM, each STTRAM memory array is usually divided to two sub-arrays, which share sense amplifiers placed in the middle. Therefore, separated read and write paths to two different sub-arrays inside one bank can still be operated simultaneously in STTRAM arrays.

Thus, we propose the Split Bank Write technique (**SBW**), in which the arbitrator treats a read access and a write access to the same RF bank but different sub-arrays non-conflicting.

8.2.2 Implementation of SBW

To implement SBW, we add two buffers in the arbitrator as shown in Figure 8.3:

Write_Busy (WBusy) Buffer: it is used to memorize the banks which are occupied by write accesses. Assuming the write cycle count of STTRAM RFs is n times longer compared to the one of SRAM RFs (e.g. n equals to 3 in this work), *WBusy* buffer contains $m = \lfloor log_2 n \rfloor$ bits for each RF bank to store the remaining cycles of the current writes.

Subarray_id Buffer: it is used to store the identifier of the accessed sub-array when a write happens. This buffer contains only one bit for each RF bank since each bank is divided to two sub-arrays. In addition, the input address buffer for each RF bank is decoupled for two sub-arrays.

After adding these two buffers, the work flow of the arbitrator is modified as follows to distribute non-conflicting reads and writes to RF banks:

- First, the arbitrator reads *WBusy* buffer. If the corresponding entry for bank M is 0, it means that bank M is free to assign any access; otherwise, it means bank M is occupied by a write access, and the arbitrator needs to read the corresponding entry in *Subarray_id* buffer in order to get the accessed sub-array identifier.

- Second, the arbitrator uses the same algorithm as the traditional one [116] to decide a group of non-conflicting accesses (G_{access}) and sends them to RF. The input is the status of read queues for each register bank and writeback informations from ALU and memory units. The only difference here is that the accessed sub-arrays in banks with non-zero *WBusy* entry are all avoided.

- Next, for the bank chosen to write in G_{access}, the arbitrator sets the corresponding entry in *WBusy* buffer to $n - 1$ (e.g. 2 in this work). In addition, The arbitrator calculates which sub-array is used for this register address, and the identifier (0 or 1) is stored in *Subarray_id* buffer.

- Then, the arbitrator subtracts all the other non-zero entries in *WBusy* buffer by 1 since the write accesses have just consumed one cycle.

An example is shown in Figure 8.3 to illustrate the work flow. Assuming that at a certain cycle the arbitrator finds there is a read request for RF Bank 1 in the

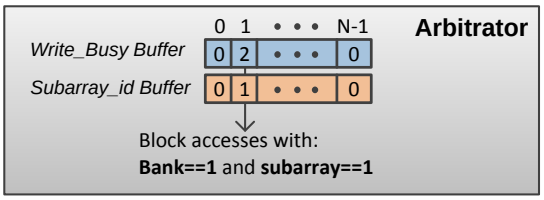

Figure 8.3. The architecture of Split Bank Write technique.

queue. It reads *WBusy* buffer and finds that the data in entry 1 is "2", which means Bank 1 is occupied by a write access that needs two more cycles to finish. Without the SBW technique, this read access needs to be blocked until the write is finished. But, when SBW is adopted, the arbitrator reads *Subarray_id* buffer and finds the write is operated in sub-array 1 of Bank 1. Thus, this read can be issued normally if it accesses sub-array 0.

8.3 Writeback Pool

8.3.1 Motivation

Besides longer write latency, write energy of STTRAM-based RFs is also much larger than SRAM-based ones. SBW is designed to reduce the performance overhead, but it does not reduce the total access count as well as the dynamic energy. Thus, we still need other techniques to solve this problem.

GPGPU usually has many warps running in parallel. Two-level warp scheduler was proposed in previous work to manage the warps and utilize computational resources more effectively [58, 117]. It partitions warps into an active set eligible for execution and an inactive pending set waiting for long latency operations. A warp encountering a stall-inducing event can be demoted to the pending set, and a pending warp can be promoted to the active set when it is ready.

This two-level warp scheduler provides an opportunity to reduce the write access number to RFs. Only active warps require register accesses, and the number of active warps is relatively small compared to the total warp number. Before being moved to the pending set, an active warp might write some registers repeatedly, and it is possible to avoid these repeated write accesses. On the basis of this

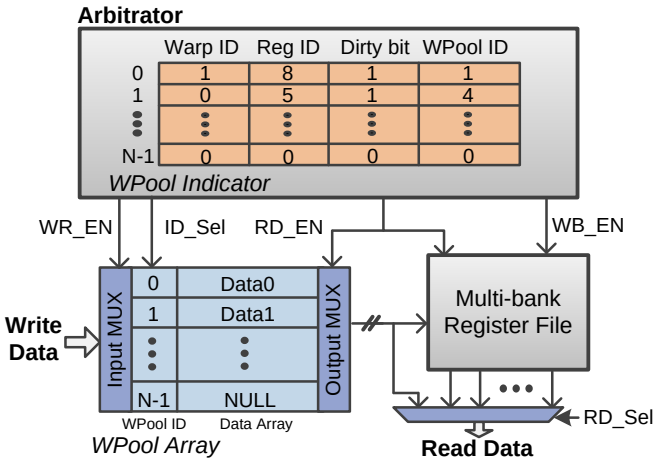

Figure 8.4. The architecture of Write Pool technique.

concept, we propose the Write Pool (**WPool**) technique.

8.3.2 Implementation of WPool

We add a small SRAM-based write buffer between the arbitrator and the RF, so that write accesses initiated by an active warp can be temporarily held in this write buffer and only written back to the STTRAM-based RF when this buffer is full. In addition, the entries containing the registers for pending warps are replaced out firstly.

Architecture of WPool: As shown in Figure 8.4, WPool consists of two components:

- *WPool Indicator:* It is placed in the arbitrator and has N entries, where N is a designer parameter to tune the balance between energy saving and hardware overhead (we use N=60 in this work). Each indicator entry includes 4 fields: *warp id*, *register id*, *dirty bit* and *WPool id*, in which *dirty bit* indicates whether an entry is occupied or not, *warp id* and *register id* store the write access information, *WPool id* points to an entry in the second WPool component which actually stores the register data.

- *WPool Array:* it is a separate small SRAM buffer which also has N entries. Each entry stores two fields: *WPool id* is used to connect with WPool Indicator; *data array* stores the register data, in which the size of one entry

is the same with the size of one register multiplied by the thread number in one warp since all threads in a warp execute same instructions.

Work flow of WPool: The arbitrator decides the access group as described in Section 8.2, and the write pool is considered as a separate bank besides all the other register banks when non-conflicting accesses are selected.

As shown in Figure 8.4, if there is a access to bank M, the arbitrator firstly checks WPool Indicator using *warp id* and *register id*, and then its behavior is as follows:

- If it is a read access, the arbitrator sets *RD_EN* and *RD_Sel* correspondingly to choose between WPool and RF. Besides, if the data is in WPool, *ID_Sel* is set as *WPool id*.

- If it is a write hit, the arbitrator sends *WPool id* to WPool Array by *ID_Sel* and sets *WR_EN* to 1. Thus, the data is written to the corresponding entry in WPool Array.

- If it is a write miss and the arbitrator finds an empty entry in WPool Indicator, its behavior is the same as a write hit. It sends *ID_Sel* to WPool Array and writes the data to the empty entry.

- If it is a write miss and there is no empty entry left, the arbitrator chooses an entry to replace. It sends *ID_Sel* as the selected *WPool id*, and sets *WB_EN* to 1 in order to write the old data into RFs. Then *WR_EN* is set to 1 and the new data is loaded to WPool Array.

Note that SBW is also adopted here to make sure write accesses are operated correctly without blocking other accesses.

Replacement policy: The replacement policy of WPool is modified based on Least Recently Used (LRU) policy, which discards the least recently used items first. Different from LRU, we change the policy as follows. When a warp is demoted to the pending list from the active set, all the entries in WPool Indicator with this *warp id* are shifted to LRU side. It is because register accesses only happen in active threads, and the registers allocated for a pending warp will only be accessed when this warp is active again. The interval between a warp is demoted and

promoted is usually very long since the pending warp is waiting for long latency operations, such as off-chip DRAM accesses. Thus, writing back the dirty data occupied by pending warps can save space for active warps.

8.4 Implementation Overhead

We combine these two techniques to form our WarRF design. The implementation overhead of WarRF is small. For SBW, it does not bring any circuit overhead on memory arrays since it explores an existing special feature of STTRAM array. The only overhead is two small buffers (e.g. 48 bits) located in the arbitrator, and the modification on the arbitrator algorithm is also small. For WPool, the overhead mainly comes from WPool array. In this work, we choose 60 as the entry number, which brings smaller than 6% overhead compared to the RF size. And this overhead can be reduced further by adjusting the number of WPool entries. The energy overhead of one access to WPool array is about 0.08pJ/bit calculated by NVSim [79], and it is also included in our experimentel results.

8.5 Experimental Results

8.5.1 Experiment Methodology

We model a contemporary GPU SIMT processor, which is similar to the Nvidia Fermi GTX480 GPU [115] using our modified version of GPGPU-Sim [118]. Our modification to GPGPU-Sim includes an asymmetric read/write latency model for RFs, a modification to the arbitrator, and the WarRF implementation. We select the simulation workloads from GPGPU-Sim benchmark suite [118] and Rodinia benchmark [119].

Table 8.1 shows the system configurations. Each GPU core is equipped with a 128KB register file which consists of 16 banks. In addition, all the circuit-level parameters (e.g. read/write latency and energy consumption) are obtained from NVSim [79] and are shown in Table 8.2. It shows that STTRAM-based RF has much smaller area and leakage power compared to SRAM-based one. But, the write latency and the write energy of STTRAM-based RF are much larger.

Table 8.1. Simulation Configuration

Parameter	Value
Core Number	15 cluster, 1 core/cluster
Clock (Core/Icnt/DRAM)	0.7/1.4/0.924 GHz
SIMD Width	32
Warp Size	32
Registers Per Core	128KB
Register File Banks	16
Shared Memory	48KB
Shared Memory Banks	32
L1/Cons/Tex Cache	16KB/8KB/12KB
L2 Cache	786KB

Table 8.2. Characteristics of SRAM and STTRAM RFs (22nm)

Memory type	SRAM	STTRAM
Cell factor (F^2)	146	40
Read latency (ns)	0.69	0.88
Write latency (ns)	0.67	4.12
Read energy (pJ/bit)	0.37	0.42
Write energy (pJ/bit)	0.32	0.72
Leakage power (uW/bank)	25	0.3
Area (mm^2)	4.1	1.1

8.5.2 Performance Improvement of WarRF

Figure 8.5 illustrates the normalized IPC of different systems. Compared to SRAM-based RF, STTRAM-based one degrades the system performance due to its longer write latency. Observed from Figure 8.5, the overall performance of STTRAM baseline is degraded by 14% compared to SRAM RF. This is also the problem addressed by WarRF.

First, the SBW technique improves the system performance by issuing reads and writes to different sub-arrays in the same bank. Compared to STTRAM baseline, the overall system performance after adopting SBW is boosted by 7% (up to 11%). Moreover, the system performance is further improved by 6% on average (up to 14%) after adopting the WPool technique because the write traffic to STTRAM RF is reduced.

Combining these two techniques, WarRF can improve the system performance

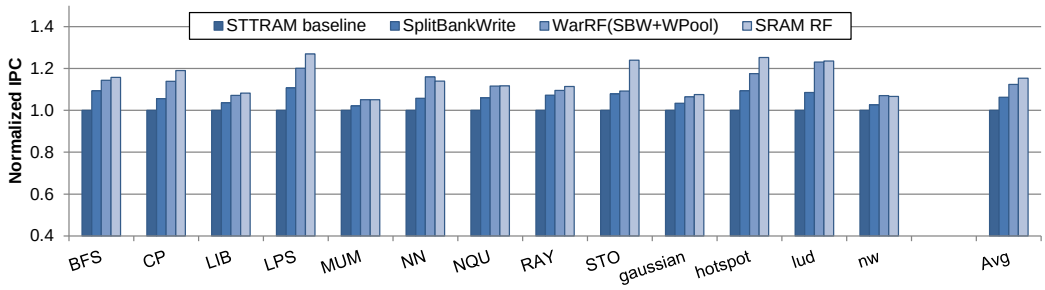

Figure 8.5. The normalized IPC compared between STTRAM baseline RF, SBW, WarRF and SRAM RF based systems.

of STTRAM-based RF by 13% on average and up to 23%. In general, the workload with more RF write traffic benefits more from our WarRF architecture since the long write latency is originally the bottleneck of the baseline system.

Thus, by adopting WarRF, STTRAM-based RF can achieve comparable performance with SRAM-based one, and the difference is only about 2.5% on average.

8.5.3 Write Access Reduction and Energy Consumption Saving

Adopting WPool in WarRF can help to reduce the number of write accesses to RF. Figure 8.6 shows that the write traffic to RF is reduced by 31% on average and up to 83% using WPool. It brings benifits in terms of both performance improvement and energy saving.

Figure 8.7 shows the normalized energy consumption of different systems and each value is broken down to leakage energy and dynamic energy. The energy overhead to access WPool array is also included in the results. Compared to SRAM-based RF, the total energy consumption of STTRAM baseline is reduced by 32% on average because the leakage energy of STTRAM is only about 1% of SRAM. But, we can also observe that the dynamic energy of STTRAM baseline is increased about 49% on aveargе because the write energy of STTRAM is much larger. As a result, for some workloads, the total energy consumption after adopting STTRAM-based RF is not decreased but increased, such as *LPS*, in which the energy consumption is increased by 18%.

After adopting WarRF, the dynamic energy is decreased since the write traffic

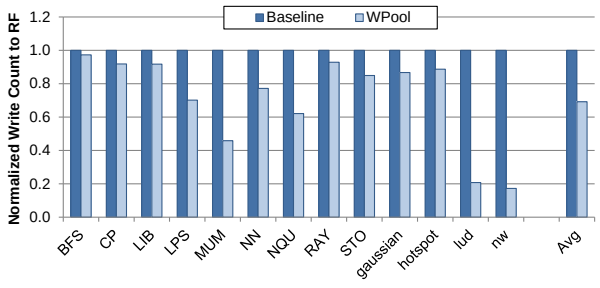

Figure 8.6. The write traffic reduction after adopting the WPool technique.

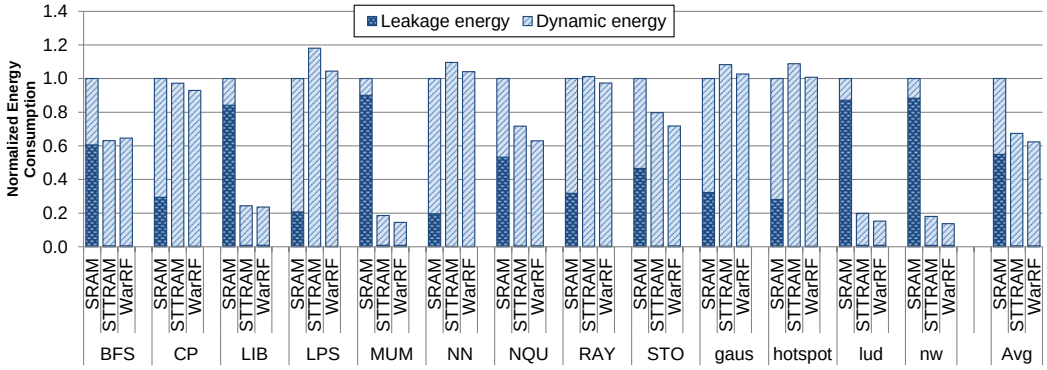

Figure 8.7. The normalized energy consumption compared between SRAM RF, STTRAM RF, and WarRF.

to RF is reduced. As shown in Figure 8.7, the energy consumption of WarRF is further reduced by 7.4% on average and up to 24%. As a result, the overall energy consumption of WarRF is reduced by 38% compared to SRAM-based RF.

8.5.4 Silicon Area Saving

Another advantage of STTRAM-based RF is the silicon area reduction due to its much smaller cell size. Based on Table 8.2, an 2MB STTRAM RF can save about 68% area compared to an SRAM one, in which the overhead of WarRF is also added. Therefore, considering comparable performance and the area/energy savings, STTRAM-based RFs enhanced by WarRF makes itself a more attractive choice in replacing SRAM ones.

8.6 Summary

GPGPUs usually have large size of register files to implement the zero overhead switch between threads. RFs are built with SRAM by default, but the scaling of SRAM is constrained by cell density and leakage energy. STTRAM is a new non-volatile memory technology and is explored as one of the potential alternatives for SRAM because of its smaller cell area and zero standby leakage power. However, STTRAM-based RFs face the problems of performance loss and higher dynamic energy due to longer write latency and higher write energy. In this work, we propose a write-aware STTRAM-based RF (WarRF) for GPGPUs, which contains two techniques: SBW modifies the arbitrator design to increase the parallelism of read and write accesses; WPool reduces the number of repeated write accesses to RFs. Our experiment shows that by adopting WarRF, the STTRAM-based system performance is improved by 13% on average and the energy consumption is reduced by 7.4% further, which makes STTRAM RF a more attactive choice.

Chapter 9

Conclusion

While the scaling of SRAM and DRAM is facing constrains of cell density and leakage energy, people are looking for their replacements. NVMs are promising candidates because of their low leakage power and high density. Therefore, there are many recent works in academic and industry to study on the feasibility of using NVMs as different memory levels.

Although using NVMs brings benefits in terms of lower leakage energy and higher density, there are still obstacles to adopt them in practical products, such as limited write endurance, expensive write operations, etc. It is very difficult to solve these issues on cell-level since they are caused by intrinsic features of non-volatile memory cells. Therefore, we can only mitigate these problems on architecture level, which is also the motivation of our work. In this dissertation, the impacts of NVM features are analyzed on system level, and some architecture designs using NVMs have been investigated.

First, we explore NVMs as main memory system. We first study on how to use multi-bit phase change memory to build the main memory system and propose an energy-efficient architecture based on data encoding to reduce the writing energy overhead. Then, we work on STTRAM main memories and propose several techniques to design an LPDDR3-compatible high performance and low power architecture.

Next, we explore NVMs as on-chip caches. To build a more reliable non-volatile caches, we first study on the limited write endurance problem. A wear-levering technique is proposed to balance the write traffic to each cache line, and a hard-

error tolerant technique is proposed to correct hard errors caused by the inherent variation of the cell's lifetime due to process variations. In addition, we also work on the issue of expensive writes, and propose a technique to mitigate the performance overhead caused by the longer write latency of non-volatile last-level caches.

Last but not least, we evaluate NVMs as GPGPU register files and propose a write-aware NVM-based register file design.

Overall, this dissertation intends to build the novel computer memory architecture design that takes the advantages of emerging NVMs and mitigate their disadvantages as well.

Bibliography

[1] Bedeschi, F., R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande (2009) "A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage," *IEEE Journal of Solid-State Circuits*, **44**(1), pp. 217–227.

[2] Condit, J., E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee (2009) "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the Symposium on Operating Systems Principles*, pp. 133–146.

[3] Caulfield, A. M., A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson (2010) "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories," in *Proceedings of the International Symposium on Microarchitecture*, pp. 385–395.

[4] Ameen, A., A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson (2011) "Onyx: A Protoype Phase Change Memory Storage Array," in *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems*, pp. 1–5.

[5] 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 International Symposium on Computer Architecture*, pp. 24–33.

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

[7] 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 International Symposium on Computer Architecture, pp. 14–23.

[8] 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 International Symposium on Microarchitecture*, pp. 14–23.

[9] 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.

[10] 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 International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–14.

[11] 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.

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

[13] 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.

[14] Xu, W., H. Sun, X. Wang, Y. Chen, and T. Zhang (2011) "Design of Last-Level On-Chip Cache Using Spin-Torque Transfer RAM (STT RAM)," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **19**(3), pp. 483–493.

[15] Zhou, P., B. Zhao, J. Yang, and Y. Zhang (2009) "Energy reduction for STT-RAM using early write termination," in *Proceedings of the International Conference on Computer-Aided Design*, pp. 264–268.

[16] Smullen, C., V. Mohan, A. Nigam, S. Gurumurthi, and M. Stan (2011) "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 50–61.

[17] 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.

[18] Sun, Z., X. Bi, H. H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu (2011) "Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme," in *Proceedings of the International Symposium on Microarchitecture*, pp. 329–338.

[19] Nomura, K., K. Abe, H. Yoda, and S. Fujita (2012) "Ultra low power processor using perpendicular-STT-MRAM/SRAM based hybrid cache toward next generation normally-off computers," *Journal of Applied Physics*, **111**(7), pp. 07E330–07E330–3.

[20] Chen, E., D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. Wolf, A. W. Ghosh, J. Lu, S. J. Poon, M. Stan, W. Butler, S. Gupta, C. K. A. Mewes, T. Mewes, and P. Visscher (2010) "Advances and Future Prospects of Spin-Transfer Torque Random Access Memory," *IEEE Transactions on Magnetics*, **46**(6), pp. 1873–1878.

[21] Duan, R., M. Bi, and C. Gniady (2011) "Exploring memory energy optimizations in smartphones," in *Proceedings of the International Green Computing Conference and Workshops*, pp. 1–8.

[22] Lebeck, A. R., X. Fan, H. Zeng, and C. Ellis (2000) "Power aware page allocation," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 105–116.

[23] Zhou, P., V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar (2004) "Dynamic tracking of page miss ratio curve for memory management," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 177–188.

[24] Udipi, A. N., N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi (2010) "Rethinking DRAM design and organization for energy-constrained multi-cores," in *Proceedings of the International Symposium on Computer Architecture*, pp. 175–186.

[25] Zheng, H., J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu (2008) "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *Proceedings of the International Symposium on Microarchitecture*, pp. 210–221.

[26] Ahn, J. H., J. Leverich, R. S. S. Schreiber, and N. P. Jouppi (2009) "Multicore DIMM: an energy efficient memory module with independently controlled DRAMs," *Computer Architecture Letters*, **8**(1), pp. 5–8.

[27] Liu, J., B. Jaiyen, R. Veras, and O. Mutlu (2012) "RAIDR: Retention-aware intelligent DRAM refresh," in *Proceedings of the International Symposium on Computer Architecture*, pp. 1–12.

[28] Nair, P., C.-C. Chou, and M. K. Qureshi (2013) "A case for Refresh Pausing in DRAM memory systems," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 627–638.

[29] Mukundan, J., H. Hunter, K.-H. Kim, J. Stuecheli, and J. F. Martínez (2013) "Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems," in *Proceedings of the International Symposium on Computer Architecture*, pp. 48–59.

[30] 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**(4/5), pp. 465–479.

[31] Hellmold, S. (2013) "Delivering Nanosecond-Class Persistent Memory," in *Flash Memory Summit*, p. 1.

[32] Ramos, L. E., E. Gorbatov, and R. Bianchini (2011) "Page Placement in Hybrid Memory Systems," in *Proceedings of the International Conference on Supercomputing*, pp. 85–95.

[33] Rizzo, N. D., D. Houssameddine, J. Janesky, R. Whig, F. B. Mancoff, M. L. Schneider, M. DeHerrera, J. Sun, K. Nagel, S. Deshpande, H.-J. Chia, S. M. Alam, T. Andre, S. Aggarwal, and J. M. Slaughter (2013) "A Fully Functional 64 Mb DDR3 ST-MRAM Built on 90 nm CMOS Technology," *IEEE Transactions on Magnetics*, **49**(7), pp. 4441–4446.

[34] Sheu, S.-S., M.-F. Chang, K.-F. Lin, C.-W. Wu, Y.-S. Chen, P.-F. Chiu, C.-C. Kuo, Y.-S. Yang, P.-C. Chiang, W.-P. Lin, C.-H. Lin, H.-Y. Lee, P.-Y. Gu, S.-M. Wang, F. Chen, K.-L. Su, C.-H. Lien, K.-H. Cheng, H.-T. Wu, T.-K. Ku, M.-J. Kao, and M.-J. Tsai (2011) "A 4Mb embedded SLC resistive-RAM macro with 7.2ns read-write random-access time and 160ns MLC-access capability," in *Proceedings of the International Solid-State Circuits Conference*, pp. 200–201.

[35] Kim, C., J.-J. Kim, S. Mukhopadhyay, and K. Roy (2003) "A forward body-biased low-leakage SRAM cache: device and architecture considerations," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 6–9.

[36] Kim, Y.-B., S. R. Lee, D. Lee, C. B. Lee, M. Chang, J. H. Hur, M.-J. Lee, G.-S. Park, C.-J. Kim, U.-i. 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.

[37] Huai, Y. (2008) "Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects," *Association of Asia Pacific Physical Societies Bulletin*, **18**(6), pp. 33–40.

[38] Dong, X., X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen (2008) "Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement," in *Proceedings of Design Automation Conference*, pp. 554–559.

[39] Kim, K. and S. J. Ahn (2005) "Reliability investigations for manufacturable high density PRAM," in *Proceedings of the International Reliability Physics Symposium*, pp. 157–162.

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

[41] Tsuchida, K., T. Inaba, K. Fujita, Y. Ueda, T. Shimizu, Y. Asao, T. Kajiyama, M. Iwayama, K. Sugiura, S. Ikegawa, T. Kishi, T. Kai, M. Amano, N. Shimomura, H. Yoda, and Y. Watanabe (2010) "A 64Mb MRAM with clamped-reference and adequate-reference schemes," in *Proceedings of the International Solid-State Circuits Conference*, pp. 268–269.

[42] Slaughter, J. M., N. D. Rizzo, F. B. Mancoff, R. Whig, K. Smith, S. Aggarwal, and S. Tehrani (2010) "Toggle and spin-torque MRAM: Status and outlook," *Journal of the Magnetic Society of Japan*, **5**(1), pp. 171–176.

[43] Chen, B., Y. Lu, B. Gao, Y. H. Fu, F. F. Zhang, P. Huang, Y. S. Chen, L. F. Liu, X. Y. Liu, J. F. Kang, Y. Y. Wang, Z. Fang, H. Y. Yu, X. Li, X. P. Wang, N. Singh, G. Q. Lo, and D.-L. Kwong (2011) "Physical mechanisms of endurance degradation in TMO-RRAM," in *Proceedings of the International Electron Devices Meeting*, pp. 12.3.1–12.3.4.

[44] Lee, H., Y. S. Chen, P. Chen, P. Gu, Y. Hsu, S. Wang, W. Liu, C. Tsai, S. Sheu, P.-C. Chiang, W. Lin, C. H. Lin, W.-S. Chen, F. Chen, C. Lien, and M. Tsai (2010) "Evidence and solution of over-RESET problem for HfO_x 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.

[45] Yang, B.-D., J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu (2007) "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 3014–3017.

[46] Joo, Y., D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie (2010) "Energy- and endurance-aware design of phase change memory caches," in *Proceedings of the Design, Automation Test in Europe*, pp. 136–141.

[47] Cho, S. and H. Lee (2009) "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proceedings of the International Symposium on Microarchitecture*, pp. 347–357.

[48] Xu, W., J. Liu, and T. Zhang (2009) "Data manipulation techniques to reduce phase change memory write energy," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 237–242.

[49] Joshi, M., W. Zhang, and T. Li (2011) "Mercury: A Fast and Energy-Efficient Multi-level Cell based Phase Change Memory System," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 345–356.

[50] Qureshi, M. K., M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis (2010) "Morphable memory system: A robust architecture for exploiting multi-level phase change memories," in *Proceedings of the International Symposium on Computer Architecture*, pp. 153–162.

[51] Dong, X. and Y. Xie (2011) "AdaMS: Adaptive MLC/SLC phase-change memory design for file storage," in *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 31–36.

[52] Joo, Y., Y. Cho, D. Shin, and N. Chang (2007) "Energy-Aware Data Compression for Multi-Level Cell (MLC) Flash Memory," in *Proceedings of the Design Automation Conference*, pp. 716–719.

[53] Meza, J., J. Li, and O. Mutlu (2012) "A case for small row buffers in non-volatile main memories," in *Proceedings of the International Conference on Computer Design*, pp. 484–485.

[54] ——— (2012) *Evaluating Row Buffer Locality in Future Non-Volatile Main Memories*, *Tech. rep.*, CMU-2012-002.

[55] Kultursay, E., M. Kandemir, A. Sivasubramaniam, and O. Mutlu (2013) "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 256–267.

[56] Joo, Y., D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie (2010) "Energy- and endurance-aware design of phase change memory caches," in *Proceedings of the Design, Automation Test in Europe*, pp. 136–141.

[57] Seznec, A. (2010) "A Phase Change Memory as a Secure Main Memory," *IEEE Computer Architecture Letters*, pp. 5–8.

[58] Gebhart, M., D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron (2011) "Energy-efficient mechanisms for managing thread context in throughput processors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 235–246.

[59] Gebhart, M., S. W. Keckler, and W. J. Dally (2011) "A compile-time managed multi-level register file hierarchy," in *Proceedings of the International Symposium on Microarchitecture*, pp. 465–476.

[60] Abdel-Majeed, M. and M. Annavaram (2013) "Warped register file: A power efficient register file for GPGPUs," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 412–423.

[61] 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 *Proceedings of the International Symposium on Computer Architecture*, pp. 247–258.

[62] Jing, N., Y. Shen, Y. Lu, S. Ganapathy, Z. Mao, M. Guo, R. Canal, and X. Liang (2013) "An energy-efficient and scalable eDRAM-based register file architecture for GPGPU," in *Proceedings of the International Symposium on Computer Architecture*, pp. 344–355.

[63] Goswami, N., B. Cao, and T. Li (2013) "Power-performance co-optimization of throughput core architecture using resistive memory," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 342–353.

[64] Nirschl, T., J. Philipp, T. Happ, G. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C. Chen, E. Joseph,

M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. Chen, Y. Zhu, R. Bergmann, H. L. Lung, and C. Lam (2007) "Write strategies for 2 and 4-bit multi-level phase-change memory," in *Proceedings of the International Electron Devices Meeting*, pp. 461–464.

[65] Kang, D. H., J. H. Lee, J. H. Kong, D. Ha, J. Yu, C. Y. Um, J. Park, F. Yeung, J.-H. Kim, W. I. Park, Y. J. Jeon, M. Lee, J. Park, Y. Song, J. Oh, H. Jeong, and H. Jeong (2008) "Two-bit cell operation in diode-switch phase change memory cells with 90nm technology," in *Proceedings of the Symposium on VLSI Technology*, pp. 98–99.

[66] Grupp, L., A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel, and J. Wolf (2009) "Characterizing flash memory: anomalies, observations, and applications," in *Proceedings of the International Symposium on Microarchitecture*, pp. 24–33.

[67] Wang, J., X. Dong, G. Sun, D. Niu, and Y. Xie (2011) "Energy-efficient multi-level cell phase-change memory system with data encoding," in *Proceeings of the International Conference on Computer Design*, pp. 175–182.

[68] Wang, J., X. Dong, and Y. Xie (2012) "Data Encoding for Multi-Level Cell Phase-Change Memory," in *Proceedings of the Non-Volatile Memories Workshop*.

[69] Krebs, D., S. Raoux, C. Rettner, G. Burr, R. M. Shelby, M. Salinga, C. M. Jefferson, and M. Wuttig (2009) "Characterization of phase change memory materials using phase change bridge devices," *Journal of Applied Physics*, **106**(5), p. 054308.

[70] Bedeschi, F., E. Bonizzoni, G. Casagrande, R. Gastaldi, C. Resta, G. Torelli, and D. Zella (2005) "SET and RESET pulse characterization in BJT-selected phase-change memories," in *Proceedings of the International Symposium on Circuits and Systems*, pp. 1270–1273.

[71] Chung, H., B.-H. Jeong, B. Min, Y. Choi, B.-H. Cho, J. Shin, J. Kim, J. Sunwoo, J. min Park, Q. Wang, Y.-J. Lee, S. Cha, D. Kwon, S. Kim, S. Kim, Y. Rho, M.-H. Park, J. Kim, I. Song, S. Jun, J. Lee, K. Kim, K. won Lim, W. ryul Chung, C. Choi, H. Cho, I. Shin, W. Jun, S. Hwang, K.-W. Song, K. Lee, S. whan Chang, W.-Y. Cho, J.-H. Yoo, and Y.-H. Jun (2011) "A 58nm 1.8V 1Gb PRAM with 6.4MB/s program BW," in *Proceedings of the International Solid-State Circuits Conference*, pp. 500–502.

[72] 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. 72–81.

[73] Standard Performance Evaluation Corporation, "SPEC OMP (OpenMP Benchmark Suite)," `http://www.spec.org/omp/`.

[74] Magnusson, P., M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner (2002) "Simics: A Full System Simulation Platform," *Computer*, **35**(2), pp. 50–58.

[75] Association, J. S. S. T., "JESD209-2E LPDDR2," .

[76] Wang, J., X. Dong, and Y. Xie (2014) "Enabling High-performance LPDDRx-compatible MRAM," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 339–344.

[77] Association, J. S. S. T., "JESD209-3 LPDDR3," .

[78] Thoziyoor, S., J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi (2008) "A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies," in *Proceedings of the International Symposium on Computer Architecture*, pp. 51–62.

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

[80] Alam, S. M., T. Andre, and D. Gogl, "Memory controller and method for interleaving DRAM and MRAM accesses," US Patent 2012/0155160 A1.

[81] Slaughter, J. M., N. D. Rizzo, J. Janesky, R. Whig, F. B. Mancoff, D. Houssameddine, J. J. Sun, S. Aggarwal, K. Nagel, S. Deshpande, S. M. Alam, T. Andre, and P. LoPresti (2012) "High Density ST-MRAM Technology," in *Proceedings of the International Electron Devices Meeting*, pp. 29.3.1–29.3.4.

[82] Micron, "121-Ball LPDDR2-PCM and LPDDR2 MCP," .

[83] La, O.-G., "SDRAM having posted CAS function of JEDEC standard," US Patent 6,483,769.

[84] Bae, Y.-C., J.-Y. Park, S. J. Rhee, S. B. Ko, Y. Jeong, K.-S. Noh, Y. Son, J. Youn, Y. Chu, H. Cho, M. Kim, D. Yim, H.-C. Kim, S.-H. Jung, H.-I. Choi, S. Yim, J.-B. Lee, J.-S. Choi, and K. On (2012) "A 1.2V 30nm 1.6Gb/s/pin 4Gb LPDDR3 SDRAM with input skew calibration and enhanced control scheme," in *Proceedings of the International Solid-State Circuits Conference*, pp. 44–46.

[85] Hidaka, H., Y. Matsuda, M. Asakura, and K. Fujishima (1990) "The cache DRAM architecture: a DRAM with an on-chip cache memory," *IEEE Micro*, **10**(2), pp. 14–25.

[86] Zhang, Z., Z. Zhu, and X. Zhang (2001) "Cached DRAM for ILP processor memory access latency reduction," *IEEE Micro*, **21**(4), pp. 22–32.

[87] Binkert, N., B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood (2011) "The Gem5 Simulator," *Computer Architecture News*, **39**(2), pp. 1–7.

[88] Rosenfeld, P., E. Cooper-Balis, and B. Jacob (2011) "DRAMSim2: A Cycle Accurate Memory System Simulator," *Computer Architecture Letters*, **10**(1), pp. 16–19.

[89] Rixner, S., W. J. Dally, U. J. Kapasi, M. P., and J. D. Owens (2000) "Memory access scheduling," in *Proceedings of the International Symposium on Computer Architecture*, pp. 128–138.

[90] SPEC CPU, "SPEC CPU2006," `http://www.spec.org/cpu2006/`.

[91] EEMBC, "EEMBC benchmark," `http://www.eembc.org/`.

[92] HPEC, "HPEC benchmark," `http://www.omgwiki.org/hpec/`.

[93] Wang, J., X. Dong, Y. Xie, and N. Jouppi (2013) "i2WAP: Improving non-volatile cache lifetime by reducing inter- and intra-set write variations," in *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 234–245.

[94] Wang, J., X. Dong, Y. Xie, and N. P. Jouppi (2014) "Endurance-aware Cache Line Management for Non-volatile Caches," *ACM Transactions on Architecture and Code Optimization*, **11**(1), pp. 4:1–4:25.

[95] Kroft, D. (1998) "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the International Symposium on Computer Architecture*, pp. 195–201.

[96] Rosenfeld, P., E. Cooper-Balis, and B. Jacob (2011) "DRAMSim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, **10**(1), pp. 16–19.

[97] Rixner, S., W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens (2000) "Memory access scheduling," in *Proceedings of the International Symposium on Computer Architecture*, pp. 128–138.

[98] Micron, "4Gb: x32 Mobile LPDDR2 SDRAM," .

[99] Blum, L., M. Blum, and M. Shub (1986) "A simple unpredictable pseudo-random number generator," *SIAM Journal on computing*, **15**(2), pp. 364–383.

[100] Kirkpatrick, S. and E. P. Stoll (1981) "A very fast shift-register sequence random number generator," *Journal of Computational Physics*, **40**(2), pp. 517–526.

[101] Wang, J., X. Dong, and Y. Xie (2012) "Point and Discard: A Hard-error-tolerant Architecture for Non-volatile Last Level Caches," in *Proceeings of the Design Automation Conference*, pp. 253–258.

[102] 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.

[103] Zhang, W. and T. Li (2009) "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *Proceedings of the International Symposium on Microarchitecture*, pp. 2–13.

[104] Wang, J., X. Dong, and Y. Xie (2013) "OAP: An obstruction-aware cache management policy for STT-RAM last-level caches," in *Proceedings of the Design, Automation Test in Europe*, pp. 847–852.

[105] ——— (2014) "Preventing STT-RAM Last-Level Caches from Port Obstruction," *ACM Transactions on Architecture and Code Optimization*, **11**(3), pp. 23:1–23:19.

[106] Kawahara, T., R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno (2007) "2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read," in *Proceedings of the International Solid-State Circuits Conference*, pp. 480–617.

[107] Rao, H. M. and J. P. Kim, "Multi-port non-volatile memory that includes a resistive memory element," US Patent 8400822 B2.

[108] Kroft, D. (1998) "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the International Symposium on Computer Architecture*, pp. 81–87.

[109] Qureshi, M. K. and Y. N. Patt (2006) "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," in *Proceedings of the International Symposium on Microarchitecture*, pp. 423–432.

[110] Xie, Y. and G. H. Loh (2009) "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the International Symposium on Computer Architecture*, pp. 174–183.

[111] Gupta, S., H. Gao, and H. Zhou (2013) "Adaptive Cache Bypassing for Inclusive Last Level Caches," in *Proceedings of the International Symposium on Parallel Distributed Processing*, pp. 1243–1253.

[112] Corporation, N., "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Nvidia White Paper.

[113] Brookwood, N., "AMD Fusion. Family of APUs: Enabling Superior, Immersive PC Experience," AMD White Paper.

[114] NVIDIA, "Geforce GTX 680," `http://www.geforce.com/hardware`.

[115] ———, "Geforce GTX 480," `http://www.geforce.com/hardware`.

[116] Liu, S., J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand collector architecture," US Patent 7,834,881.

[117] Narasiman, V., M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt (2011) "Improving GPU performance via large warps and two-level warp scheduling," in *Proceedings of the International Symposium on Microarchitecture*, pp. 308–317.

[118] Bakhoda, A., G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt (2009) "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pp. 163–174.

[119] 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 *Proceedings of the International Symposium on Workload Characterization*, pp. 44–54.

Vita

Jue Wang

Jue Wang received B.E. and M.E. degrees from Tsinghua University in China. Currently, she is a Ph.D. candidate in the Computer Science and Engineering Department, Pennsylvania State University. She is a member of Microsystems Design Laboratory (MDL) group working with Professor Yuan Xie. Her research interests are high performance and low power computer architecture design, with a particular emphasis on memory systems. She has published 16 conference and journal papers, as well as 1 US patent. She has also served as a peer reviewer for several journals and conferences in the field of computer architecture, VLSI design, and electronic design automation, including the ACM Transactions on Architecture and Code Optimization (TACO), IEEE Transactions on Very Large Scale Integration Systems (TVLSI), Design Automation Conference (DAC), International Symposium on High Performance Computer Architecture (HPCA).