University of California
Santa Barbara

Facilitating Emerging Non-volatile Memories in Next-Generation Memory System Design: Architecture-Level and Application-Level Perspectives

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

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Ping Chi

Committee in charge:

Professor Yuan Xie, Chair
Professor Kwang-Ting (Tim) Cheng
Professor Malgorzata Marek-Sadowska
Professor Dmitri Strukov
Professor Mary Jane Irwin, Pennsylvania State University
Professor Wang-Chien Lee, Pennsylvania State University

June 2016

The Dissertation of Ping Chi is approved.

Professor Kwang-Ting (Tim) Cheng

Professor Malgorzata Marek-Sadowska

Professor Dmitri Strukov

Professor Mary Jane Irwin, Pennsylvania State University

Professor Wang-Chien Lee, Pennsylvania State University

Professor Yuan Xie, Committee Chair

June 2016

Facilitating Emerging Non-volatile Memories in Next-Generation Memory System Design: Architecture-Level and Application-Level Perspectives

by

Ping Chi

This dissertation is dedicated to my parents Yuqin Qiu and Yijiang Chi, for their love, encouragement, and support throughout my life.

Acknowledgements

Many people have offered me great help during my journey to pursue a Ph.D. degree.

First of all, I thank Prof. Yuan Xie, my Ph.D. advisor, for his guidance and support that lead to my academic progress and achievement. He is my role model of a researcher. I could not show more respect for his enthusiasm and perseverance in academic pursuits, his broad and insightful vision on our research areas, and his diligent and energetic working style. Moreover, he is always helpful throughout my Ph.D. study. He provided me hands-on help when I was in my early years, and encouraged me when I was not confident of my research ideas. It will always be my honor to have worked with him.

Many professors from University of California Santa Barbara (UCSB) and the Pennsylvania State University (Penn State) have given me helpful advice and instructions. I am grateful to Prof. Kwang-Ting (Tim) Cheng, Prof. Malgorzata Marek-Sadowska, and Prof. Dmitri Strukov for serving on my dissertation committee at UCSB and providing me with insightful suggestions on my research work. I reserve my sincere gratitude for Prof. Wang-Chien Lee and Prof. Mary Jane Irwin who continue serving as my dissertation committee members even after I transferred from Penn State to UCSB. I learned a lot from Prof. Lee about how to do research since he co-advised my first research project and revised my first paper submission word by word. Prof. Irwin is my role model of a successful female researcher. I appreciate her constructive advice on my research work and also on my career plan, and her encouragement and support that brought me self-confidence during my entire Ph.D. study. I also thank Prof. Vijaykrishnan Narayanan, Prof. John Sampson, and Prof. Zhiwen Liu who used to serve on my dissertation committee at Penn State and gave me useful feedbacks at my early stage. Prof. Narayanan was always friendly to me. He welcomed me to be seated among his students and encouraged us to learn from each other, offering me the opportunity to learn various interesting

research topics from his research group.

Moreover, I sincerely thank Dr. Paolo Faraboschi and Dr. Qiong Cai for hosting my internship at Hewlett-Packard Labs. Their rich research experience and in-depth knowledge taught me a memorable lesson.

Furthermore, I wish to thank my friends and colleagues at UCSB and Penn State who have made my life vivid and beautiful over these years. Many senior group members helped me a lot when I was new to the U.S. and our lab. I am grateful to Guangyu Sun, Jin Ouyang, Jing Xie, Hsiang-yun Cheng, Qiaosha Zou, Jishen Zhao, Jue Wang, Xiangyu Dong, Dimin Niu, Yibo Chen, and Matt Poremba. I especially thank Tao Zhang and Cong Xu, who are my good "mentors" and friends, always generous with their time and knowledge, helping me on my research projects as well as in my personal life. Also, I have been very fortunate to work with my nice and brilliant group comrades, Jia Zhan, Hang Zhang, Kaisheng Ma, Shuangchen Li, Ziyang Qi, Itir Akgun, Peng Gu, Maohua Zhu, Liu Liu, Dylan Stow, Linuo Xue, and Russell Barnes.

Most importantly, I would like to thank my parents, Yuqin Qiu and Yijiang Chi, who raised and educated me, and always love me. I also thank my brother, Cheng Chi, and my entire extended family, for their continuous support.

Curriculum Vitæ
Ping Chi

Education

2016	Ph.D. in Computer Engineering (Expected), University of California, Santa Barbara, USA.
2011	M.S.E. in Electronic Science and Technology, Tsinghua University, Beijing, China.
2008	B.Eng. in Information Electronics and Engineering, Tsinghua University, Beijing, China.

Publications

[C1] **Ping Chi**, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, Yuan Xie. "A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory", in *Proc. of the 43rd International Symposium on Computer Architecture (ISCA)*, June 2016, Seoul, Korea. *(To appear.)*

[C2] **Ping Chi**, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H. Kang, Yuan Xie. "Architecture Design with STT-RAM: Opportunities and Challenges", in *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2016, Macao, China. Invited paper.

[C3] Shuangchen Li, **Ping Chi**, Jishen Zhao, Kwang-Ting Cheng, Yuan Xie. "Leveraging Nonvolatility for Architecture Design with Emerging NVM", in *Proc. of Non-Volatile Memory System and Applications Symposium (NVMSA)*, Aug. 2015, Hong Kong, China. Invited paper.

[C4]. **Ping Chi**, Cong Xu, Tao Zhang, Xiangyu Dong, Yuan Xie. "Using Multi-Level Cell STT-RAM for Fast and Energy-Efficient Local Checkpointing", in *Proc. of International Conference on Computer-Aided Design (ICCAD)*, Nov. 2014, San Jose, USA. **(Best Paper Award - Front End.)**

[C5] **Ping Chi**, Wang-Chien Lee, Yuan Xie. "Making B^+-tree Efficient in PCM-Based Main Memory", in *Proc. of International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2014, La Jolla, USA.

[C6] **Ping Chi**, Cong Xu, Xiaochun Zhu, Yuan Xie. "Building Energy-Efficient Multi-Level Cell STT-MRAM Based Cache Through Dynamic Data-Resistance Encoding", in *Proc. of International Symposium on Quality Electronic Design (ISQED)*, Mar. 2014, Santa Clara, USA.

[J1] **Ping Chi**, Wang-Chien Lee, Yuan Xie. "Adapting B+-Tree for Emerging Nonvolatile Memory Based Main Memory", in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD). (To appear.)*

[J2] Jishen Zhao, Cong Xu, **Ping Chi**, Yuan Xie. "Memory and storage system design with nonvolatile memory technologies", in *IPSJ Transactions on System LSI Design Methodology (TSLDM)*, 8:2-11, 2015. Invited paper.

Abstract

Facilitating Emerging Non-volatile Memories in Next-Generation Memory System Design: Architecture-Level and Application-Level Perspectives

by

Ping Chi

As technology scales, the increasing leakage power dissipation and the degraded reliability of conventional SRAM and DRAM technologies cause growing concern. Emerging non-volatile memory (NVM) technologies have shown great potential to build next-generation memory systems, by combining low leakage power, good scalability, fast speed, and high density. As they are maturing, it is important for computer architects to understand their pros and cons and leverage them in future computer system design.

This dissertation focuses on three types of emerging NVMs, spin-transfer torque RAM (STT-RAM), phase change memory (PCM), and metal-oxide resistive RAM (ReRAM). STT-RAM has been identified as the best replacement of SRAM to build large-scale and low-power on-chip caches and also an energy-efficient alternative to DRAM as main memory. PCM and ReRAM have been considered to be promising technologies for building future large-scale and low-power main memory systems. This dissertation investigates two aspects to facilitate them in next-generation memory system design, architecture-level and application-level perspectives. First, multi-level cell (MLC) STT-RAM based cache design is optimized by using data encoding and data compression. Second, MLC STT-RAM is utilized as persistent main memory for fast and energy-efficient local checkpointing. Third, the commonly used database indexing algorithm, B^+tree, is redesigned to be NVM-friendly. Forth, a novel processing-in-memory architecture built on ReRAM based main memory is proposed to accelerate neural network applications.

Contents

Chapter 1

Introduction

In contemporary computer architecture design, SRAM based on-chip caches, DRAM based off-chip main memory, and solid state drive (SSD) and/or hard disk drive (HDD) based storage, are the major components of the memory hierarchy. The importance of the memory embodiments increases with the rapid advance in microprocessor performance, especially for many-core and large-scale computing systems. However, as technology scales, the conventional SRAM and DRAM technologies suffer from increasing leakage power dissipation and degraded reliability. Also, as DRAM density increases, the refresh penalty becomes nontrivial and can result in significant performance degradation [1]. Moreover, NAND flash SSD suffers from the notorious endurance issue.

In recent years, we have seen many emerging non-volatile memory (NVM) technologies which have better scalability and zero leakage power and have been considered as promising alternatives to conventional memory technologies. A lot of effort has been made on the research and development of these NVM technologies by both academia and industry. Recently, Intel and Micron announced their 3D XPoint technology which is a new class of NVM and promises 1000× lower latency and higher endurance than NAND flash. As these emerging memory technologies are maturing, it is important

for computer architects to understand their pros and cons and leverage them in next-generation memory system design to improve the performance, power, and reliability of future computing systems.

This dissertation is one of such efforts. It is focused on three types of emerging NVM technologies, *Spin-Transfer Torque Random Access Memory (STT-RAM)*, *Phase Change Memory (PCM)*, and *metal-oxide Resistive RAM (ReRAM)*. It tries to facilitate these emerging NVM technologies in next-generation memory system design from two perspectives: architecture-level and application-level. STT-RAM, PCM, and ReRAM, all have non-volatility, low leakage power, asymmetric read and write latency, and asymmetric read and write energy. STT-RAM can achieve SRAM-similar or DRAM-similar read latency and read energy under the same capacity, with its cell size between DRAM's and SRAM's. Therefore, it has been identified as a promising replacement of SRAM to build large-size and low-power on-chip caches and also as an energy-efficient alternative to DRAM. PCM and ReRAM have higher densities than DRAM, and better endurance than NAND flash, and can achieve DRAM-similar read latency and read energy. Therefore, they have been considered as potential candidates to build large-size and low-power main memory and high-performance storage. The multi-level cell (MLC) technology can improve their densities. Currently, STT-RAM can achieve 2 bits/cell, while PCM and ReRAM can achieve 4 or more bits/cell. Besides their nice features, they also have disadvantages. For example, they all suffer from long write latency and high write energy. Although they have better endurance than NAND flash, they still have limited lifetimes.

First, it is necessary to propose solutions to mitigate their disadvantages, such as high write energy. MLC STT-RAM have even higher write energy than single level cell (SLC). Based on the observation that the write energy consumption has great value dependency, a dynamic data-resistance encoding mechanism is proposed to reduce the write energy of MLC STT-RAM based last level cache (LLC), which encodes more frequent data values

to more energy-efficient resistance states at runtime. However, this dynamic encoding technique incurs a slight performance degradation due to the encoding overhead. To eliminate the performance degradation while to save the write energy, the MLC STT-RAM cache design is optimized by using data compression. By compressing a cache line data to half the size, it can be fit into the fast-read and fast-write soft-bit region of series MLC STT-RAM cells, which requires only one-step read and write operations, avoiding the high energy consumption and long latency of two-step read and write accesses. Furthermore, by exploiting the saved space from data compression to store more data, the increased effective cache capacity can improve system performance. The techniques proposed above are architectural solutions. This dissertation also addresses the challenges from application-level perspectives. For main memory databases (MMDBs) built on emerging NVMs, the nice features of NVMs, such as non-volatility, low power, and high density of PCM and ReRAM, will bring great benefits to the system. However, the assumptions that have served as the basis to design conventional database algorithms are changed by the unique characteristics of NVMs such as asymmetric read and write latency and energy. The widely used database indexing algorithm, B^+tree, is redesigned with the new design goal to reduce more expensive write accesses, even at the cost of increasing read accesses.

Second, it is beneficial to take advantage of their unique features in system design. For one example, MLC provides an in-cell multi-versioning opportunity for local checkpointing, since each cell can store two or more bits. An efficient local checkpointing scheme is proposed which leverages MLC STT-RAM as main memory. Different from MLC PCM or MLC ReRAM which adopts the program-and-verify scheme in write operations, MLC STT-RAM has write schemes that contain two steps at most. In the MLC STT-RAM main memory design, the working data are mapped to the soft-bits and the checkpoint data to the hard-bits, so that only one-step write operations are incurred during the en-

tire execution time including the checkpoint and recovery periods. By replacing the data transfer between the main memory and the backup storage with the data transfer between two bits of the memory cells which only requires one-step writes, the proposed scheme efficiently reduces the performance and energy overhead of local checkpointing. For another example, ReRAM has the analog computation capability with a crossbar structure besides the data storage capability. It can execute matrix-vector multiplications very efficiently, and has been widely studied to accelerate neural network (NN) computations that involve a large amount of matrix-vector multiplications. A processing-in-memory (PIM) architecture built on ReRAM crossbar main memory is proposed to accelerate NN computation. In the ReRAM main memory design, a small portion of ReRAM crossbar arrays are enabled to perform NN computation by additional peripheral circuit support, and they can work as NN accelerators and also as normal memory. By reducing the cost of the data movement between the processing units and the off-chip memory in NN computation, the proposed PIM architecture can improve system performance and energy efficiency significantly for NN applications. The above two proposals are both novel architecture design for specific applications.

The remainder of this dissertation is organized as follows. Chapter 2 introduces the technology background of STT-RAM, PCM and ReRAM. Then, Chapter 3 presents the proposals to optimize MLC STT-RAM cache design using data encoding and data compression, and Chapter 4 discusses the project to use MLC STT-RAM for efficient local checkpointing. Next, Chapter 5 introduces the redesign of the B^+-tree algorithm for emerging NVM based main memory systems. Then, Chapter 6 presents the proposal to accelerate NN computation in ReRAM based main memory. Finally, Chapter 7 concludes this dissertation.

Chapter 2

Technology Background

This chapter introduces the basics of STT-RAM, PCM, and ReRAM. They are all resistive random access memories, a type of NVM that uses the cell resistance to store the information by changing it between a high-resistance state (HRS) and a low-resistance state (LRS). It is known as ReRAM or RRAM for short. However, ReRAM typically refers to metal-oxide ReRAM, a subset using metal oxides as resistive switches. In this dissertation, the term "ReRAM" is used for metal-oxide ReRAM in a typical way.

STT-RAM is a new generation of magnetic RAM, using spin-transfer torque to switch memory states. Figure 2.1 (a) shows a magnetic tunnel junction (MTJ), the key component to store the bit information in a STT-RAM cell. An MTJ consists of two ferromagnetic layers and one oxide barrier layer (*e.g.* MgO) that separates them. The magnetization direction of one ferromagnetic layer (*i.e.* reference layer) is fixed, called reference layer; while that of the other ferromagnetic layer (*i.e.* free layer) can be changed by applying a large enough spin polarized current through MTJ, either in parallel or anti-parallel with the fixed direction of the reference layer. "Parallel" indicates MTJ in an LRS; in contrast, "anti-parallel" means HRS. The magnetization directions of the MTJs in Figure 2.1 (a) are in-plane, and they can also be made perpendicular [2]. As the

technology node scales, perpendicular MTJ based STT-RAM demonstrates better characteristics than in-plane MTJ based STT-RAM. It requires a lower switching current and can maintain high thermal stability for a longer retention time. STT-RAM adopts the one-transistor-one-resistor (1T1R) cell structure, as shown in Figure 2.1 (d). It is called one-transistor-one-MTJ (or 1T1J) cell structure for STT-RAM. Like a DRAM cell, each STT-RAM cell has an access transistor to prevent the disturbance with other cells.

PCM exploits the unique behavior of phase change material, *e.g.*, chalcogenide glass, which enters two different states under different heating temperatures and durations [3]. These two states, termed as amorphous state (HRS) and crystalline state (LRS), have significantly different electrical resistivities, and thus they can represent "0" and "1", respectively. Moreover, as this material can achieve several distinct intermediate states, it has the ability to represent multiple bits in a single cell. Figure 2.1 (b) depicts a PCM cell. PCM also adopts the 1T1R cell structure as shown in Figure 2.1 (d).

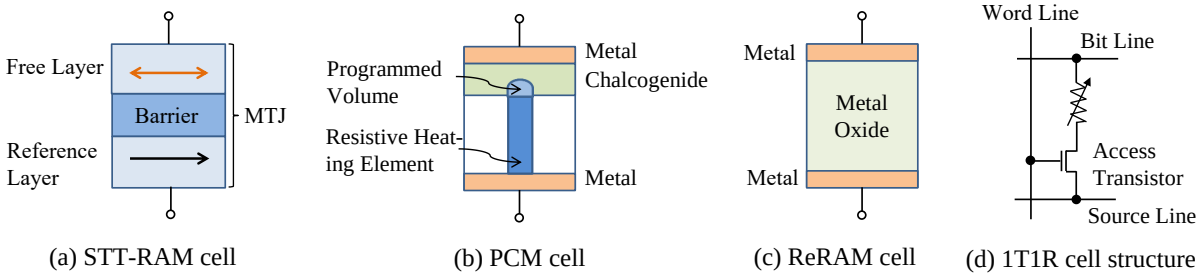


Figure 2.1: The basics of NVM cell structures (not to scale). (a) STT-RAM cell; (b) PCM cell; (c) ReRAM cell; (d) 1T1R cell structure.

Figure 2.1 (c) presents the metal-insulator-metal structure of a ReRAM cell. It contains a top metal layer, a bottom metal layer, and a metal-oxide layer in the middle as resistive switches. There are various oxide materials and metal choices, and the cell switching behavior depends on their interfacial properties [4]. According to the widely accepted filamentary model, during a SET operation, nanoscale conductive filaments (CFs) are formed and the cell is in an LRS; during a RESET operation, the CFs are

cut off and the cell is in a HRS. Like PCM, ReRAM can achieve the MLC characteristic by controlling the switching current/voltage to obtain several intermediate resistance levels. [5]. ReRAM has the 1T1R cell structure as shown in Figure 2.1 (d). To reduce the cost, ReRAM also supports a cross-point architecture, which eliminates the access transistor, using a bidirectional diode as a selector or even no selector at all to achieve the theoretical smallest cell size of $4F^2$.

Despite of different materials and technologies, PCM, STT-RAM, and ReRAM have some common characteristics. First, all have asymmetric read and write performance and energy. The reason is that the READ current/voltage they use is smaller and with a shorter duration than WRITE so as not to perturb the cell state. Table 2.1 compares PCM, STT-RAM, and ReRAM with DRAM. Their write latency is several times longer than their read latency, and their write energy consumption is several times higher than their read energy consumption. Second, they all have the write endurance problem, *i.e.*, each cell will be worn out after a limited number of writes. PCM suffers from the most severe endurance issue. ReRAM is two or three orders of magnitude better than PCM in endurance; Although STT-RAM has the best endurance among these three emerging NVMs, it is not free of this issue. Moreover, they are all non-volatile and have low idle power.

Table 2.1: Compare STT-RAM, PCM, and ReRAM with DRAM [6, 7, 8, 9, 10]

	DRAM	STT-RAM	PCM	ReRAM
Retention	refresh	non-volatile	non-volatile	non-volatile
Density	$1\times$	$0.4-0.8\times$	$2-4\times$	$2-4\times$
Endurance	10^{16}	$10^{12}-10^{15}$	10^{8}	10^{12}
Idle power	high	low	low	low
Read latency	$1\times$	$0.5-2\times$	$2-8\times$	$0.5-10\times$
Write latency	$1\times$	$1-8\times$	$10-100\times$	$1-100\times$
Read energy	$1\times$	$0.5-2\times$	$2-6\times$	$0.5-5\times$
Write energy	$1\times$	$0.5-10\times$	$5-50\times$	$0.2-100\times$

A lot of research studies has been conducted to tackle their challenges and make them more feasible memory replacements. Some studies improve their lifetime by using wear leveling strategies [3, 11, 12]. Some deal with the long write latency by using a small buffer at the architecture level [3]. Reducing redundant writes [13, 11, 14] can improve both system performance and NVM lifetime.

Chapter 3

Optimize MLC STT-RAM Cache Design Using Data Encoding and Data Compression

This chapter presents two projects that optimize MLC STT-RAM cache design, one using dynamic data-resistance encoding and the other using data compression. Before introducing these two projects, the basics of MLC STT-RAM are given first.

3.1 The Basics of MLC STT-RAM

The cell in Figure 2.1 (a) stores a single logic bit, called single level cell (SLC). To enhance the density of STT-RAM, multi-level cell (MLC) structures have been proposed, holding multiple logic bits in a single cell.

There are two categories of MLC MTJ designs, parallel [15] and series [16]. In parallel MLC MTJs, the free layer has two domains to achieve four resistance states by the combinations of their magnetization directions, as shown in Figure 3.1 (a); while series

MLC STT-RAM stacks two MTJs to represent two logic bits, as shown in Figure 3.2 (a). Parallel MLC requires relatively smaller switching current density and owns higher tunneling magneto resistance (TMR) ratio. Series MLC is easier to be fabricated. A recent study demonstrates that series MLC STT-RAM is more feasible than parallel, because parallel MLC STT-RAM design is only applicable to in-plane MTJ technology while series design is compatible with advanced MTJ technologies such as perpendicular MTJ and has overwhelming advantage in read and write reliability [17].

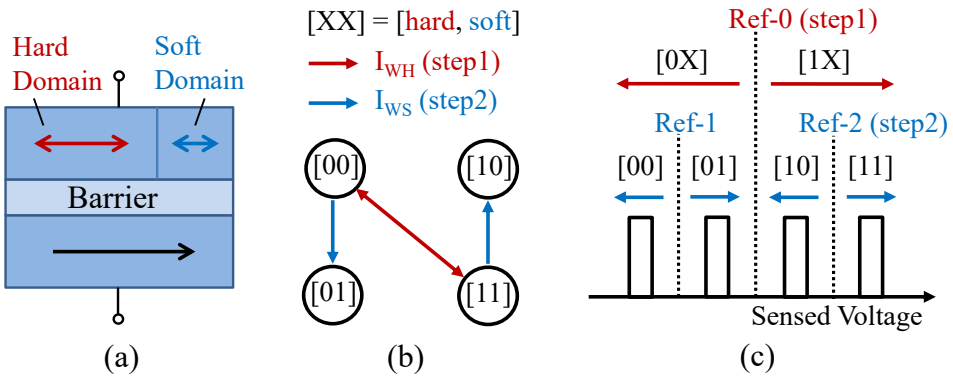


Figure 3.1: (a) Parallel MLC MTJ structure; (b) 2-step write operation; (c) 2-step read operation.

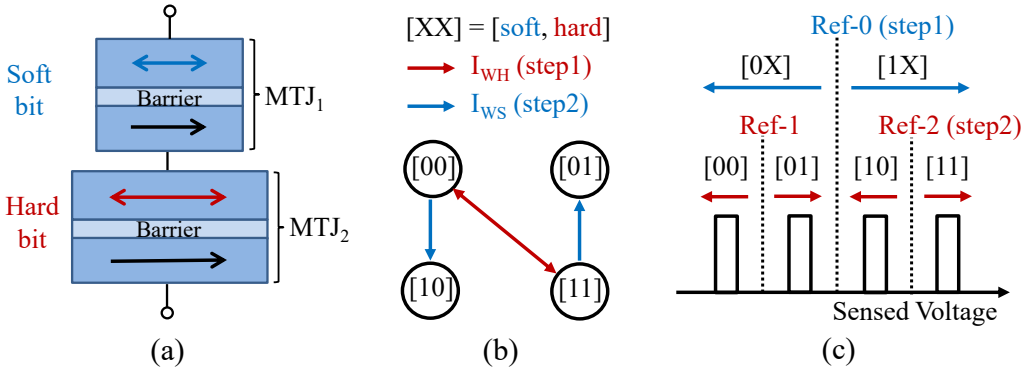


Figure 3.2: (a) Series MLC MTJ structure; (b) 2-step write operation; (c) 2-step read operation.

Figure 3.1 (a) depicts the parallel MLC MTJ structure. The two domains in the free layer have different areas so as to represent two logic bits. The bit stored in the

soft domain is called soft-bit, and the bit stored in the hard domain is called hard-bit. For a write operation, the write current passes through both the hard domain and the soft domain at runtime. However, the hard domain requires a larger switching current than the soft domain ($I_{C,hard} > I_{C,soft}$). By encoding LRS as "0" and HRS as "1", in a 2-bit cell, there are four different levels: "00", "01", "10", and "11". In parallel MLC, the most significant bit (MSB) is the hard-bit, and the least significant bit (LSB) is the soft-bit. Figure 3.1 (b) and (c) illustrate the write and read operations of parallel MLC STT-RAM. They both have two steps. As shown in Figure 3.1 (b), in a write operation, first a large current I_{WH} ($I_{WH} > I_{C,hard}$) is applied to switch both the hard-bit and the soft-bit; next a smaller current I_{WS} ($I_{C,soft} < I_{WS} < I_{C,hard}$) is used to flip only the soft-bit. As shown in Figure 3.1 (c), a read operation based on voltage sensing requires three reference voltages (Ref-0, Ref-1, and Ref-2) and two comparisons. The MSB (hard-bit) is first detected by comparing the sensing voltage with Ref-0; then based on the result, the LSB (soft-bit) is read by comparing the sensing voltage with either Ref-1 or Ref-2.

Figure 3.2 (a) presents the series MLC MTJ structure. The two MTJs have different areas in order to distinguish two logic bits. The bit stored in the smaller MTJ_1 is called soft-bit, and the bit stored in the bigger MTJ_2 is called hard-bit. The same with the parallel MLC MTJ structure, given a constant critical switching current density, the hard-bit requires a larger switching current than the soft-bit, i.e. $I_{C,hard} < I_{C,soft}$. However, different from the parallel MLC MTJ structure, the soft-bit in series MLC has a larger resistance, and it is the MSB while the hard-bit is the LSB. Figure 3.2 (b) and (c) demonstrate the write and read operations of series MLC STT-RAM. They both have two steps, similar to those of parallel MLC.

Since MLC STT-RAM can promise a higher density than SLC STT-RAM, we have seen a lot of studies that use it to build large-scale on-chip caches recently [18, 19, 20, 21]. Compared with SLC STT-RAM, MLC STT-RAM has more complex read and write

operations, and thus has even longer read and write latencies and consumes even higher read and write energy, which may degrade system performance and energy efficiency despite its capacity benefits.

3.2 Improving Write Energy through Dynamic Data-Resistance Encoding

This section presents a dynamic data-resistance encoding technique for MLC STT-RAM based cache [22]. First, the write energy model and the existing static data-resistance encoding algorithm are introduced, and then the dynamic encoding algorithm is proposed that aims to efficiently reduce the energy consumption in MLC STT-RAM write operations. Furthermore, the encoder and decoder designs and the modified architecture of MLC STT-RAM cache bank are presented. Next, after discussing the hardware overhead, the experimental setup and results are shown before this project is summarized.

This project is focused on energy-efficient parallel MLC STT-RAM, with the assumption that the basic direct data mapping is used for MLC STT-RAM based cache.

3.2.1 Write Energy Model and Data-Resistance Encoding

As described in Section 3.1 and Figure 3.1 (b), a typical write operation of a parallel MLC STT-RAM cell takes two steps. Figure 3.3 demonstrates the switching currents required for resistive state transitions of a parallel MLC at 45 nm technology node. The sign of the current value (in μA) denotes its direction: "positive" means from the free layer to the reference layer while "negative" means the reverse direction. It is obvious that changing states or writing data has significantly value-dependent energy consumption, classified by the following four categories:

- *A:* Neither bit is changed. No write current is required.
- *B:* Only LSB is flipped, *i.e.* "00"↔"01" and "10"↔"11". In this case only a small write current is needed.
- *C:* MSB is switched and in the new value MSB and LSB are the same, *i.e.* "11"(or "10")→"00" and "00"(or "01")→"11". In this case, a large write current is required to flip MSB.
- *D:* MSB is flipped and in the new value MSB and LSB are different, *i.e.* "00"(or "01")→"10" and "11"(or "10")→"01". This is the worst case in terms of both write latency and energy since direct switch is infeasible and a two-step write is needed. A large current is applied to flip MSB before a small one is employed to process LSB.

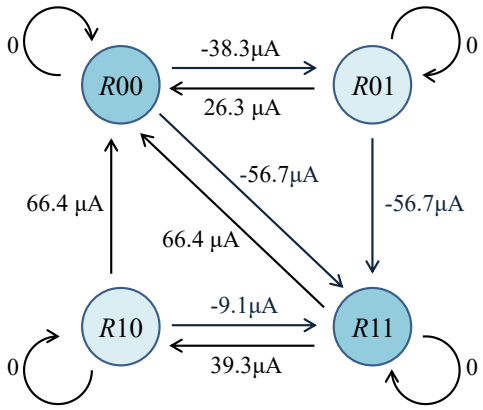


Figure 3.3: The switching currents for parallel MLC STT-RAM state transitions at 45 *nm* technology node. [19]

In the typical two-step write scheme, many unnecessary state transitions are induced. To reduce them, a hybrid write scheme has been proposed [18]. A read operation is conducted first, and based on the values of the original data and new data, the optimized type of write operation is chosen according to Figure 3.3. In *case A* if data are not

changed, the write operation is completely skipped. In *cases B* and *C*, only one step is needed by applying either a small or large current. In *case D*, two steps are unavoidable. This project assumes that the energy-efficient hybrid write scheme is employed in this project.

The write energy consumption of every state transition with the hybrid write scheme in parallel MLC STT-RAM at 45 nm technology node is calculated based on the reported data [15, 18], by assuming that 10 ns pulse duration is applied. The result is shown in Table 3.1. The last row is the average energy to write each resistance state given that the possibilities of the original state to be one of the four options are equal. Therefore, the resistance states "11" and "00" are more energy-efficient in which "11" is the most energy-efficient one, and "01" and "10" are both high energy consumption states.

Table 3.1: Write energy of every state transition with hybrid write scheme at 45 nm technology node (in pJ)

To / From	R00	R01	R10	R11
R00	0	0.045	0.185	0.120
R01	0.021	0	0.194	0.128
R10	0.144	0.189	0	0.001
R11	0.164	0.209	0.065	0
Avg.	0.082	0.111	0.111	0.062

By profiling write data in benchmarks, logic state "00" contributes about 37% of the total write operations, which indicates mapping logic data "$L00$" to resistance level "$R11$" is a beneficial way for saving energy. The optimal encoding is $\{L00 \mapsto R11$, $L01 \mapsto R10$, $L10 \mapsto R01$, $L11 \mapsto R00\}$ [19]. However, their data-resistance encoding scheme is static, in which the mapping is determined by the general write patterns abstracted from a wide range of benchmarks. In this project, a dynamic data-resistance encoding technique is proposed which decides the mapping at runtime and turns out to be more application-specific and energy-efficient.

3.2.2 Dynamic Data-Resistance Encoding Algorithm

Given a 64-byte cacheline, 256 2-bit cells in one bank are activated all together for a write operation. There exists an optimal data-resistance encoding solution that minimizes the energy consumption of switching original data to new values in a cacheline since the total $4! = 24$ encoding schemes can be enumerated. However, to find the optimal one is too costly to implement due to the limited on-chip resources. The proposed encoding algorithm can efficiently reduce write energy with minor overheads.

First, five terms are defined for future discussion, including three resistance states based on the write energy model (Table 3.1) and two logic states based on their frequencies in a cacheline writing:

- *MEES*: the most energy-efficient resistance state, *i. e.* "$R11$";
- *SEES*: the second-most energy-efficient resistance state, *i. e.* "$R00$";
- *LEESs*: the two least energy-efficient resistance states, including "$R01$" and "$R10$";
- *MFLS*: the most frequent logic state when writing a cacheline;
- *SFLS*: the second-most frequent logic state when writing a cacheline.

The key idea of the proposed encoding algorithm is to increase the frequency of energy-efficient states (*i. e.* MEES and SEES) and to decrease that of energy-inefficient states (LEESs) when writing new data to a cacheline. The algorithm ensures that MFLS in new data is mapped to MEES, and SFLS to SEES. Since the two LEESs have little difference in energy consumption, it is unnecessary to distinguish them when mapping the other two logic states to them. Therefore, there are $A_4^2 = 12$ mapping types, and thus four additional bits (that is, two 2-bit cells) are needed to record the mapping type code for each cacheline. The algorithm also avoids using energy-inefficient LEESs in

mapping type codes, so "0101", "0110", "1001" and "1010" are excluded. The mapping type lookup table demonstrates the encoding algorithm, as shown in Table 3.2. The area overhead to store the mapping type code in each cacheline is about 0.78% of the area of MLC STT-RAM arrays, given the cacheline size being 64 bytes. Additionally, the storage overhead of the mapping type lookup table is negligible.

Table 3.2: The mapping type lookup table.

Type Code	Mapping MFLS to MEES	Mapping SFLS to SEES	Mapping the other two logic states to LEESs	
00 00	$L00 \mapsto R11$	$L01 \mapsto R00$	$L10 \mapsto R01$	$L11 \mapsto R10$
00 01	$L00 \mapsto R11$	$L10 \mapsto R00$	$L01 \mapsto R01$	$L11 \mapsto R10$
00 10	$L00 \mapsto R11$	$L11 \mapsto R00$	$L01 \mapsto R01$	$L10 \mapsto R10$
00 11	$L01 \mapsto R11$	$L00 \mapsto R00$	$L10 \mapsto R01$	$L11 \mapsto R10$
01 00	$L01 \mapsto R11$	$L10 \mapsto R00$	$L00 \mapsto R01$	$L11 \mapsto R10$
01 11	$L01 \mapsto R11$	$L11 \mapsto R00$	$L00 \mapsto R01$	$L10 \mapsto R10$
10 00	$L10 \mapsto R11$	$L00 \mapsto R00$	$L01 \mapsto R01$	$L11 \mapsto R10$
10 11	$L10 \mapsto R11$	$L01 \mapsto R00$	$L00 \mapsto R01$	$L11 \mapsto R10$
11 00	$L10 \mapsto R11$	$L11 \mapsto R00$	$L00 \mapsto R01$	$L01 \mapsto R10$
11 01	$L11 \mapsto R11$	$L00 \mapsto R00$	$L01 \mapsto R01$	$L10 \mapsto R10$
11 10	$L11 \mapsto R11$	$L01 \mapsto R00$	$L00 \mapsto R01$	$L10 \mapsto R10$
11 11	$L11 \mapsto R11$	$L10 \mapsto R00$	$L00 \mapsto R01$	$L01 \mapsto R10$

3.2.3 Encoder and Decoder Design

The data flows of the encoder and decoder are shown in Figure 3.4 (a) and (b). To write data to a MLC STT-RAM cacheline, the original data are first stored in the write buffer. Then the number of each 2-bit logic state in the original data is counted by the encoder to determine which mapping type is selected by searching the mapping type lookup table. At last the data are encoded by parallel 2-bit 4-to-1 multiplexers (MUX) whose modes are controlled by the mapping type code, before they are written to the corresponding cacheline. For example, if MFLS in the original data is "10" and SFLS is "00", the mapping type with code "1000" is chosen as the mapping type lookup table

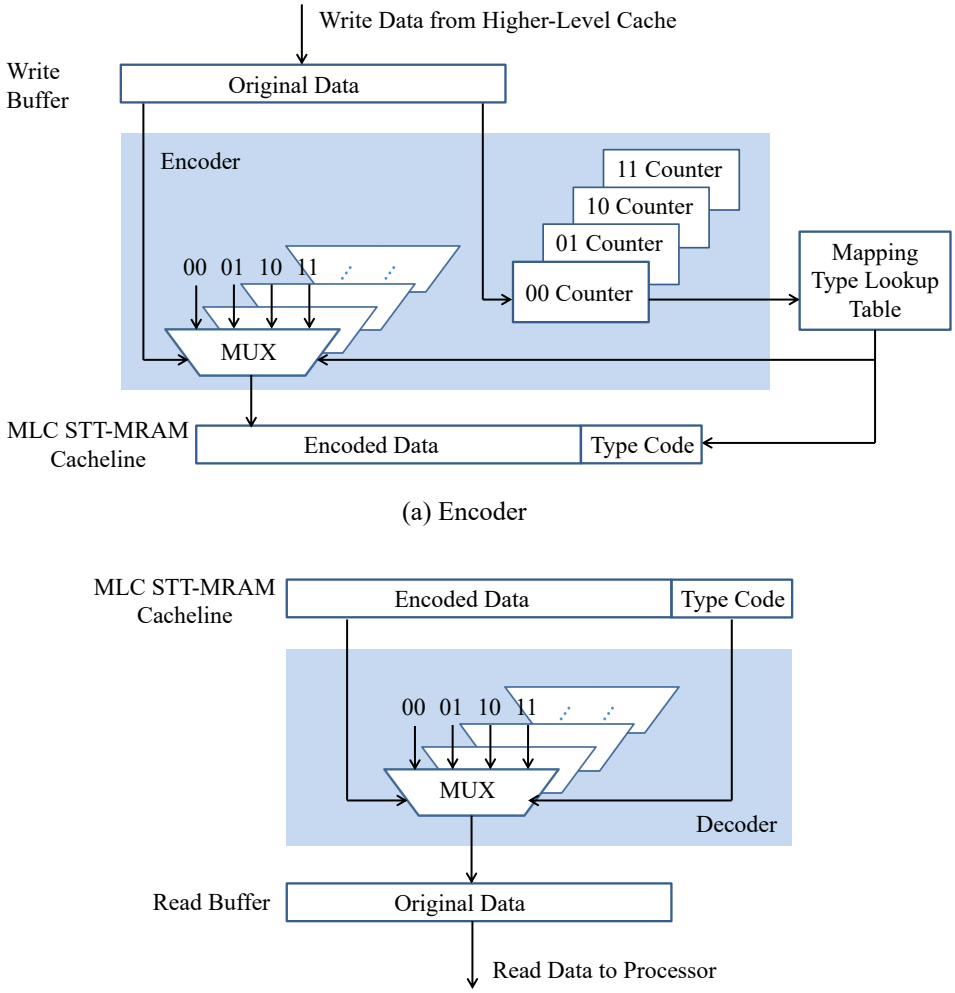


Figure 3.4: The data flow diagrams of the encoder and the decoder: (a) in write operations, data from higher-level cache are encoded before written to MLC STT-RAM cache; (b) in read operations, data in MLC STT-RAM cache are decoded before read out.

indicates, and the data are encoded in accordance with the rule: $\{L10 \mapsto R11$, $L00 \mapsto R00$, $L01 \mapsto R01$, $L11 \mapsto R10\}$. Since different mapping types may be chosen at each time based on data values, the data-resistance encoding is dynamic.

The data flow of the decoder is simpler, as Figure 3.4 (b) shows. To read a MLC STT-RAM cacheline, the data are just decoded by parallel 4-to-1 multiplexers under the control of its type code.

With additional components (the encoder and the decoder), the MLC STT-RAM cache bank architecture needs to be modified, illustrated in Figure 3.5. In a write operation, write data must be encoded before enter the write driver; in a read operation, data can be read out only after the decoding.

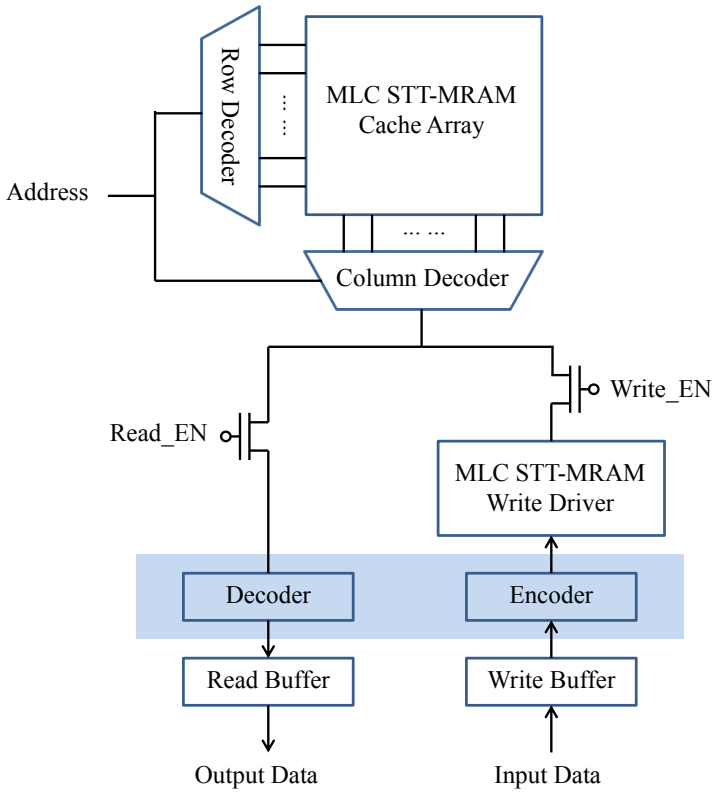


Figure 3.5: MLC STT-RAM cache bank architecture design with encoder and decoder.

3.2.4 Overhead Estimation

The hardware overhead of the encoder and decoder (as shown in Figure 3.4) is evaluated. Mishra and Akashe have designed a high-performance and low-power 4-to-1 MUX by using the CMOS transmission gate logic (TGL) in 45 nm technology, which operates at up to 200 Gb/s with the power dissipation of 1.887 nW and the area of 6.175 μm^2 [23]. Based on their reported data, the encoder and the decoder in the proposed technique

is estimated to consume about 0.000012 pJ and 0.000010 pJ extra energy per cacheline access respectively. The total area overhead is 0.007 mm^2, which is negligible to the area of MLC STT-RAM arrays (it is reported that a $45nm$ 64MB SLC STT-RAM chip can have an area of 3.06 mm^2 with device-architecture co-optimization [24]). The encoding process would induce delay to the original write latency. The decoding process is fairly simple and fast. Later, the experimental results will show that the small extra delay from the encoder and the decoder has insignificant impact on overall performance.

3.2.5 Experimental Methodology

A pin-based multi-core x86 simulator, Sniper [25], is used in the experiments. The core and memory configurations are summarized in Table 3.3. A CMP is modeled with four 4-issue, out-of-order cores running at 2GHz. Each core has private 32KB I-L1 and 32KB D-L1 caches, which are 4-way associative SRAM. Each core has an 8MB L2 cache, which is 16-way associative MLC STT-RAM with asymmetric read and write latencies. In addition, cacheline size is set to 64 bytes and write-back policy is employed. The capacity of DRAM main memory is 4GB in the simulation. There is one channel, one rank and eight banks in the DRAM main memory, and close-page policy is adopted.

Table 3.3: Baseline configurations in the simulation.

Processor	4-core CMP, x86, 2GHz, out-of-order, 4-issue, 128-entry reorder buffer
L1 cache	private, 32KB I-L1 & 32KB D-L1 SRAM, LRU, 64B line, 4-way, 2-cycle access, write back
L2 cache	private, 8MB MLC STT-RAM, LRU, 64B line, 16-way, 5-cycle read, 37-cycle write, write back
Main memory	4GB DRAM, 300-cycle access latency, 1 channel, 1 rank, 8 banks, close-page

Thirteen workloads are evaluated from SPEC CPU2006 benchmarks [26] including both integer and floating point applications. Four copies of each benchmarks were si-

multaneously executed on four cores in the simulations. For each workload, the cache is warmed up with the initial 10M instructions and simulated using the next 500M instructions. There are different read and write access numbers across these workloads, as shown in Figure 3.6. *mcf*, *gromacs* and *xalancbmk* can be categorized as write-intensive applications, while *bwaves*, *libquantum* and *gobmk* are read-intensive benchmarks.

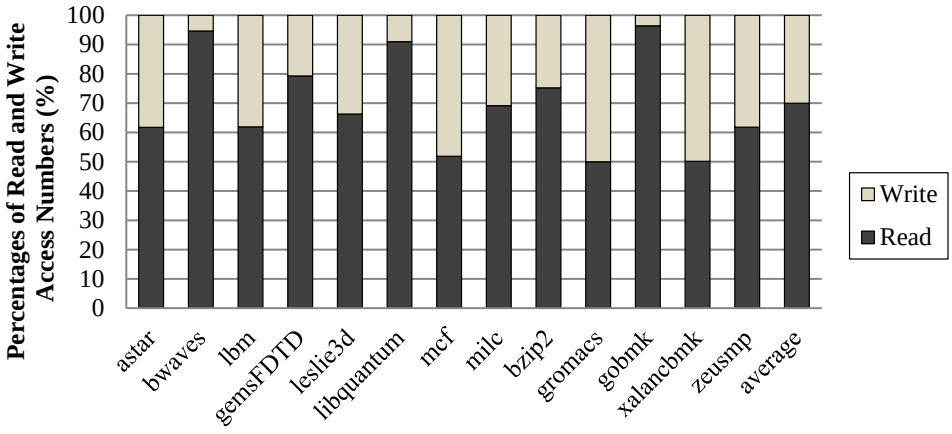


Figure 3.6: The percentages of read and write access numbers.

3.2.6 Experimental Results

By dynamically mapping resistance and logic states, the ratios of energy efficient states (including MEES and SEES) should increase compared with the static encoding scheme [19]. Figure 3.7 presents the result. As expected, the increasing percentages vary a lot across different applications, from 2.5% (*zeusm*) up to 25.2% (*bwaves*). The average (*gmean*) is 11.5% among the thirteen evaluated workloads, with four above 15%, five between 8% and 15% and four below 8%. Such dramatic variance comes from the great dissimilarity of data that are written in those applications. In *lbm*, *libquantum*, *milc*, and *zeusmp*, the ratios of energy-efficient states are already as large as over 65%, and in most write operations, $L0$0 and $L1$1 are the dominant proportion of write data. In such cases, the dynamic encoding algorithm would probably suggest the same mapping

method as the static encoding scheme does. Therefore, there is limited improvement in these applications. In *astar*, *bwaves*, *gemsFDTD* and *bzip*2, write data have varying patterns, and dynamic encoding has the flexibility to handle such cases, and thus shows better results than the static mapping.

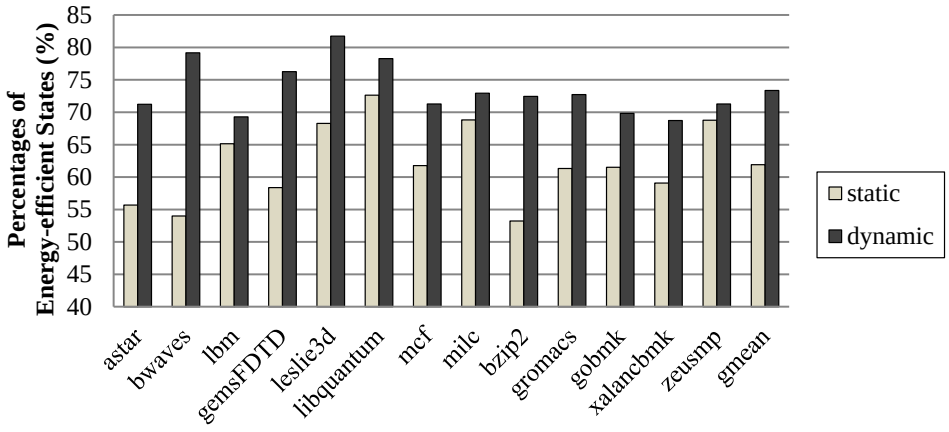


Figure 3.7: Percentages of energy-efficient states (including MEES and SEES) in static data-resistance encoding and dynamic encoding.

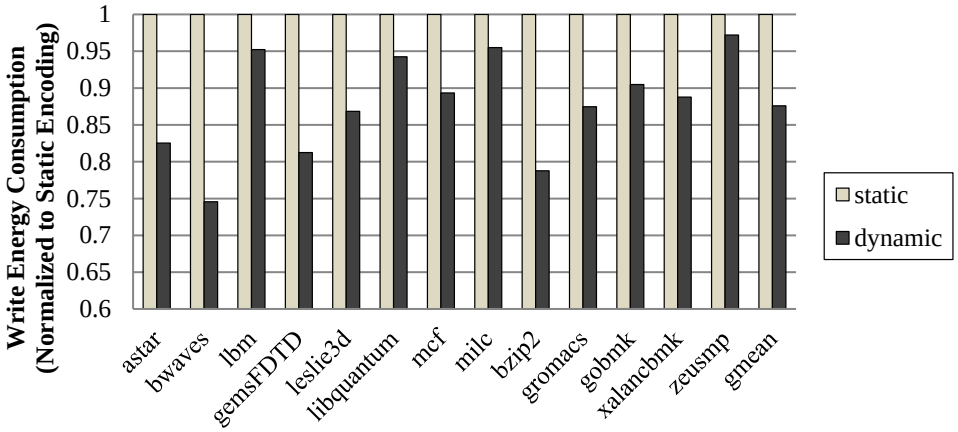


Figure 3.8: Write energy reduction of dynamic data-resistance encoding compared with static encoding.

Figure 3.8 shows write energy reduction by the dynamic data-resistance encoding compared with the static scheme. From Figure 3.8, there is 12.4% on average (*gmean*) and up to 25.4% (*bwaves*) decrease in total write energy consumption, which is in accor-

dance with the increase trend of the percentages of energy-efficient states in Figure 3.7.

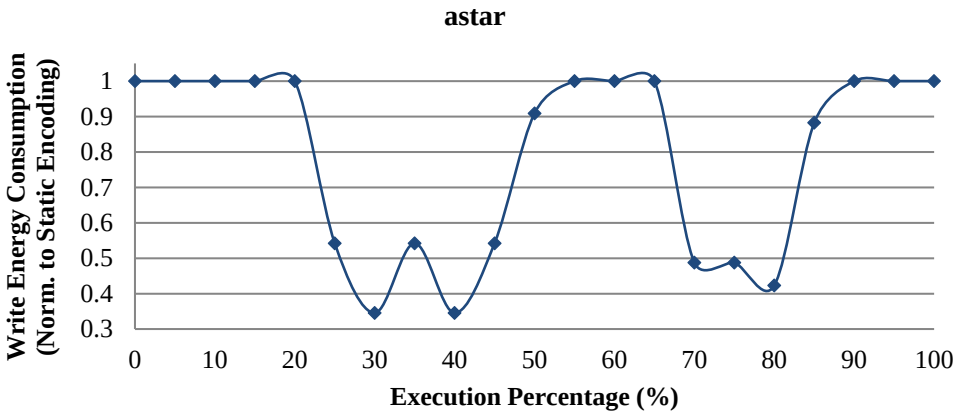


Figure 3.9: Write energy of dynamic encoding over execution time.

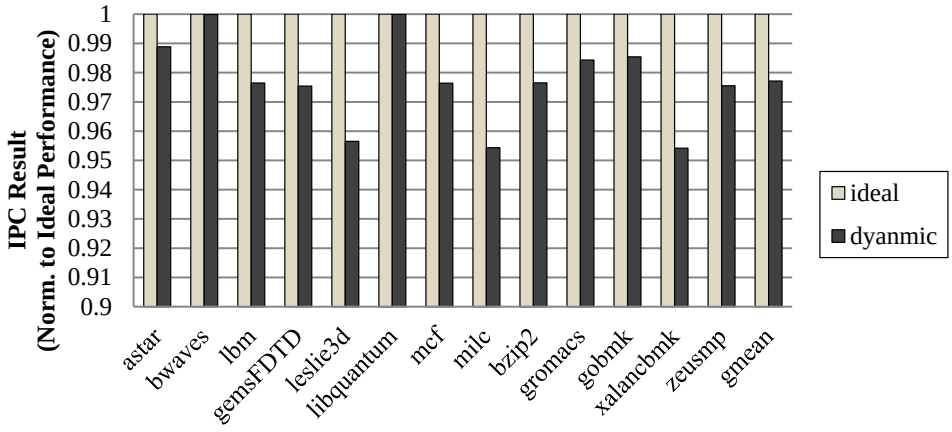


Figure 3.10: Performance degradation of dynamic encoding.

Figure 3.9 describes how dynamic encoding overwhelms the static technique in write energy consumption over execution time. During the execution process of *astar*, there are some phases when the dynamic scheme is comparable to static (0%–20%, 50%–65% and 85%–100%), and there are other phases when the dynamic encoding displays its advantages over the static scheme (20%–50% and 65%–85%). In the latter phases, the write data patterns are more random; hence dynamic encoding can achieve more energy saving by optimizing the data-resistance mapping at runtime.

It is also evaluated how much the dynamic data-resistance encoding technique affects the overall performance, as shown in Figure 3.10. Due to the extra delay induced by the encoder and the decoder, there is 2.3% on average and up to 4.6% degradation in instructions per cycle (IPC) compared with the ideal case in which there is no time overhead. From Figure 3.8 and Figure 3.10, the proposed dynamic data-resistance encoding technique can efficiently save write energy with very limited negative effect on performance.

3.2.7 Summary

Previous work has shown that STT-RAM has great potential to replace SRAM as future cache technology. MLC STT-RAM can provide even higher density than SLC STT-RAM, and is a promising technology to build large on-chip caches in CMP systems. However, MLC STT-RAM suffers from high write energy because its write operations are complex and require very large write currents. This project proposes a dynamic data-resistance encoding scheme to reduce the write energy consumption in MLC STT-RAM cache with limited degradation of overall performance. Compared with the optimal static data-resistance encoding, this dynamic encoding scheme can achieve 12.4% reduction in write energy consumption with 2.3% degradation of IPC on average.

3.3 Optimizing Energy and Performance with Data Compression

This section presents the optimized MLC STT-RAM cache design utilizing data compression. It first discusses the data mapping methods for MLC STT-RAM cache, and then introduces the cache design based on the interleaved mapping method, including the

data compression algorithm used and two proposed techniques using data compression. The overhead estimation of the proposed techniques is presented before the experimental setup and results. Finally, this project is summarized.

This project is focused on the easily fabricated series MLC STT-RAM. For series MLC STT-RAM, reading the soft-bit (MSB) takes only one step, and writing the soft-bit requires only a small switching current which will not flip the hard-bit. Therefore, series MLC STT-RAM can perform like SLC by working only on the soft-bits, which improves the access speed but reduces the capacity.

3.3.1 Data Mapping for MLC Based Cache

To adopt MLC STT-RAM for cache, one key design issue is how to map the data bits of a cache line to MLCs. A straight-forward method is *direct mapping (DM)*, as shown in Figure 3.11 (a). Given a 64B (512-bit) cache line, it uses 256 2-bit MLCs to hold a line and each cell stores two adjacent bits. Thanks to its simplicity, DM has been adopted in some previous work [18, 22]. However, since DM does not differentiate soft-bit and hard-bit regions, the read and write latencies are the worst-case two-step latencies no matter which word of a line is accessed.

To take advantage of the fact that soft-bit region is fast, Bi *et al.* proposed *cell split mapping (CSM)* [20] as shown in Figure 3.11 (b), in which a cache line is stored either in all the soft-bits or in all the hard-bits of the cells. The soft-bit line and the corresponding hard-bit line are a fast way and a slow way respectively of the same set. To make best use of fast ways, a data migration mechanism is designed including an inter-cell swapping scheme and the migration policies. To further improve the access speed, they also propose an application-aware speed enhancement (ASE) mode for MLC STT-RAM based caches, in which MLC STT-RAM can work as fast SLC at a set level

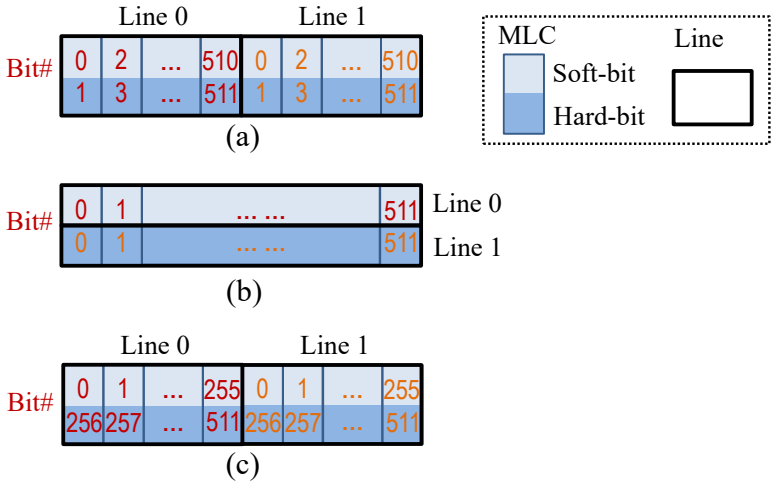


Figure 3.11: MLC data mapping methods for a 64-byte (512-bit) cache line: (a) direct mapping; (b) cell split mapping; (c) interleaved mapping.

of granularity when applications do not benefit much from the large capacity of each set offered by MLC. Each cache set can dynamically halve or double the number of ways by switching between MLC mode and ASE mode according to the result of a mode-predictor.

Interleaved mapping (IM) is another data mapping method [21]. As illustrated in Figure 3.11 (c), the lower half of the data bits of a line are stored in the soft-bit region while the higher half are put in the corresponding hard-bit region. Based on IM, Wang *et al.* proposed a dynamic block size technique (DBS) to optimize MLC STT-RAM based caches. Each cache set can operate in two modes: large block mode (LBM) and small block mode (SBM), as shown in Figure 3.12 (a). LBM is a normal mode, utilizing both the soft-bit region and the hard-bit region to store each line, while SBM uses only the soft-bit region to store the hot data chunks of a line which has half the size of a line. Therefore, SBM is faster and consumes less energy than LBM. The dynamic block size mechanism can reconfigure the block sizes of each cache set at runtime according to their proposed block size reconfiguration policy. Their experiment results show that, the dynamic LBM-SBM switching mechanism based on IM surpasses the dynamic MLC-ASE mode switching mechanism based on CSM [20], achieving about 1% more IPC

improvement and 5% more energy saving over the LBM only baseline.

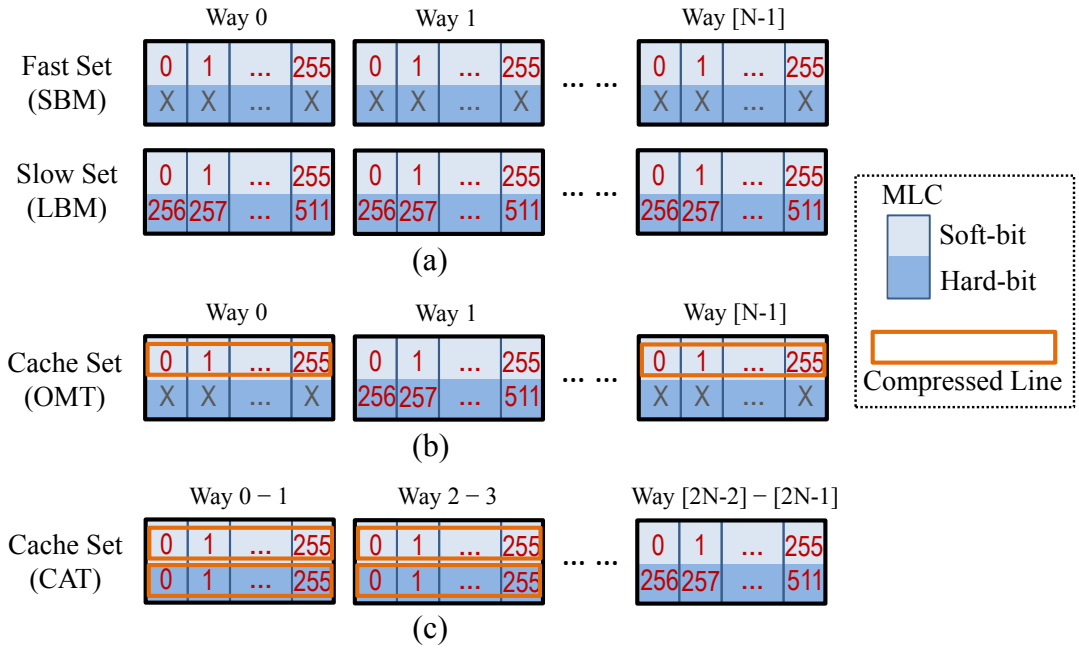


Figure 3.12: (a) The previous dynamic block size technique (DBS); (b) the proposed overhead minimized technique (OMT); (c) the proposed capacity augmented technique (CAT).

3.3.2 Design Overview

In the MLC STT-RAM cache design, the interleaved mapping method (Figure 3.11 (c)) is adopted for data mapping. However, unlike the previous work using dynamic block sizes [21], this work fixes the data block size, and employs data compression to fit a compressible line into only the soft-bit region or the hard-bit region of the cells.

Two techniques based on data compression are proposed. One is the Overhead Minimized Technique (OMT) as shown in Figure 3.12 (b), in which a compressible line is put only into the soft-bit region and the corresponding hard-bit region is not used so that the change of the cache management and the modification of the tag arrays are minimized. The other is the Capacity Augmented Technique (CAT) as shown in Figure 3.12 (c), in

which a compressible line can be put into the soft-bit region as well as the hard-bit region so that the capacity of the cache is enhanced thanks to data compression. Compared with the dynamic block size technique (DBS) as shown in Figure 3.12 (a), OMT supports a finer granularity of fast and slow ways. It does not require one whole cache set to operate in either fast way mode (SBM) or slow way mode (LBM); however, each way in a cache set can be a fast way as long as the data can be compressed into half the size and fit into the soft-bit region. Moreover, OMT is more lightweight than DBS by avoiding the hardware and scheduling overheads of dynamic block size reconfiguration. Compared with DBS, OMT and CAT provide a larger cache capacity, because the soft-bit region can store all the data of a line in a compressed fashion and in CAT the left hard-bit region can be used to store another compressible line.

3.3.3 Data Compression

The effectiveness of the proposed techniques depends on how many cache lines are compressible. In this work, a line is "compressible" specifically means that it can be compressed into half its size so that it can be fit into only the soft-bit region or the hard-bit region. A number of cache compression schemes have been proposed which exploit various data compression methods to expand the effective cache capacity. For example, Zero-Content Augmented caches represent zero-value lines in a very compact way [27]. Also, Frequent Value Compression [28] and Frequent Pattern Compression [29] have been proposed and utilized in cache designs. Additionally, Base-Delta-Immediate (BΔI) Compression [30] has been widely adopted as a practical data compression method for on-chip caches thanks to its high compression ratio, low decompression latency, and modest hardware complexity. Huffman coding based statistical compression has also been explored for cache compression [31].

The proposed techniques for MLC STT-RAM cache design do not rely on any specific compression method. This project takes the state-of-the-art BΔI compression as an example, and integrate it into the cache design. BΔI compression leverages the fact that the values within a cache line have a low dynamic range, and therefore presents a cache line using one or multiple base values and an array of differences ("deltas") whose combined size is smaller than the original cache line [30]. Two bases are used, one of which is default zero, according to the suggested best option. Assuming that the cache line size is 64 bytes, the BΔI encoding is defined in Table 3.4.

Table 3.4: BΔI encoding.

Name	Base	Δ	Size	Encoding
Zeros	1B	0B	1B	00
Base8-Δ1	8B	1B	17B	01
Base8-Δ2	8B	2B	25B	10
No Compression	N/A	N/A	64B	11

The compressor consists of three compression units: one zero compression unit, one 8-byte-base 1-byte-delta (Base8-Δ1) compression unit, and one 8-byte-base 2-byte-delta (Base8-Δ2) compression unit. Therefore, the encoding needs 2 bits, as shown in Table 3.4, and they are attached to the corresponding tag for each cache line. All these compression units operate in parallel, and a selection logic chooses the optimal one if multiple compression options are available for a cache line. In the previous BΔI compressor design [30], it contains other BΔI compression units of different base and delta sizes, *e.g.* 8-byte-base 4-byte-delta, 4-byte-base 1-byte-delta, and so on. The reasons why only Base8-Δ1 and Base8-Δ2 are chosen are: i) they can compress a cache line into half the size; ii) they can achieve almost the best compression ratio of BΔI compression; and iii) the hardware overhead is greatly reduced. In Table 3.4, the compressed cache line size for Base8-Δ1 is 17 bytes, including one 8-byte base, eight 1-byte deltas, and one byte each bit of which indicates the base for each segmentation (or delta) is either the default

zero base or the other base. Similarly, the compressed cache line size for Base8-Δ2 is 25 bytes.

A 3-level cache hierarchy as described in Table 3.5 is simulated, and the data in the last level cache are profiled over the SPEC CPU2006 benchmarks. Figure 3.13 shows the percentages of the compressible lines and the breakdowns of the compression schemes across 15 selected benchmarks. On average, 42% of the target cache line data are compressible. From the breakdown results it can be found that, for some benchmarks such as *zeusmp* and *GemsFDTD*, more lines are compressed by the "Zeros" scheme encoded as "00", and for some other benchmarks such as *milc* and *lbm*, more lines are compressed by the BΔI schemes including Base8-Δ1 and Base8-Δ2 encoded as "01" and "10".

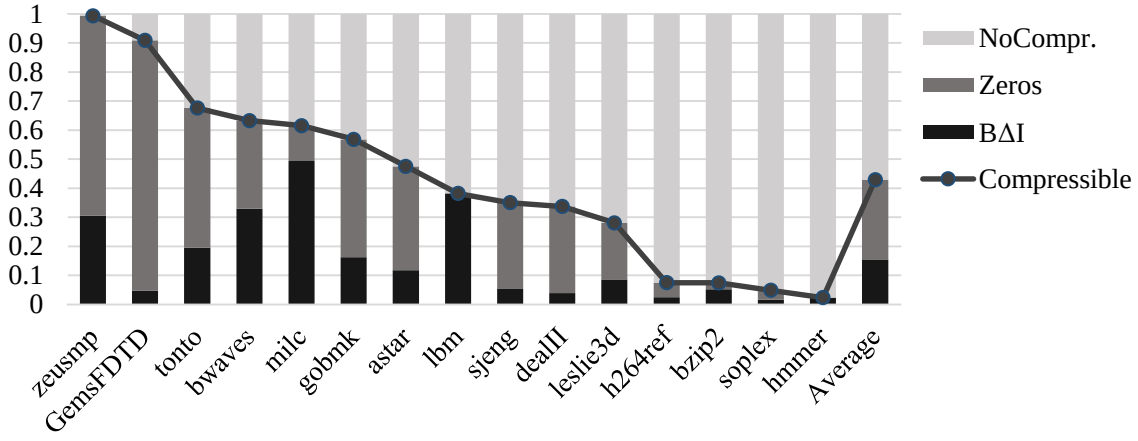


Figure 3.13: The profiling results over SPEC CPU2006 benchmarks: the percentages of compressible lines and the breakdowns of compression schemes.

3.3.4 The Overhead Minimized Technique

As shown in Figure 3.12 (b), with OMT, each way in a cache set can be a fast way or a slow way, depending on whether the data line can be compressed into half the size. As mentioned above, a 2-bit compression scheme code is attached to the tag for each cache line which indicates whether the line is compressed and which compression scheme

is used.

Figure 3.14 describes how OMT processes write and read requests. For a write request, as shown in Figure 3.14 (a), first the input data are processed by the compressor. Then, based on the compression scheme code, it is determined whether the input data have been compressed. If yes, the compressed input data are written into the soft-bit region, and it is a fast and energy-efficient write operation; otherwise, the input data are written into both the soft-bit and the hard-bit regions, and it is a slow and energy-consuming write operation. The corresponding compression scheme code in the tag arrays is also updated. For a read request, as shown in Figure 3.14 (b), first the compression scheme code in the tag arrays is checked to determine whether the line is compressed or not. If yes, the compressed data are read out from the soft-bit region through a fast read operation, and then decompressed by the decompressor according to the compression scheme code before output; otherwise, the data are read out from both the soft-bit and the hard-bit regions through a slow read operation, and then output.

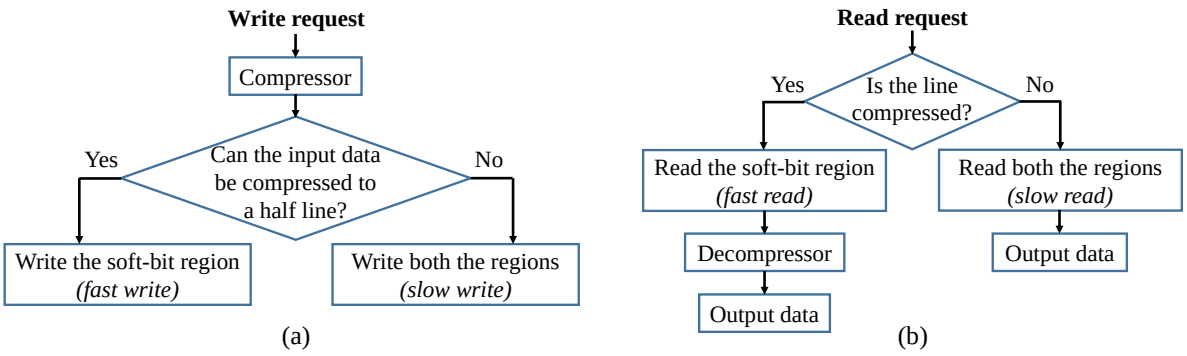


Figure 3.14: The algorithms of OMT to process (a) a write request and (b) a read request.

3.3.5 The Capacity Augmented Technique

As presented above, OMT takes advantage of data compression to reduce the access latency and energy to MLC STT-RAM. However, it does not leverage the capacity

benefits from data compression. Therefore, CAT is proposed, which increases the cache capacity by using the saved hard-bit regions to accommodate more compressible lines of data. As shown in Figure 3.12 (c), two compressible lines can reside in the soft-bit region and the hard-bit region of the cells simultaneously. Therefore, the maximum number of ways that each cache set can hold doubles.

To fully utilize the capacity benefits requires doubling the tags, which incurs a considerable overhead. To reduce the tag overhead, the maximum number of ways of each cache set can be limited. According to the profiling results of the percentage of compressible lines as shown in Figure 3.13, increasing the tags by 50% is a reasonable choice. A simple tag-data mapping based on the conventional one is proposed, in which the original tags are directly mapped to the original data positions while the added tags can be mapped to any hard-bit region of the cache set. Therefore, each added tag needs to store its data position. For example, it requires 4 additional bits per added tag to store 16 possible hard-bit region positions in an originally 16-way cache. Moreover, the cache replacement policy needs to be modified since some ways are compressed while others are not. The very simple least recently used (LRU) policy is applied, which chooses the LRU compressed line or the LRU in-compressed line for replacement according to whether the incoming line is compressible or not. The study of more intelligent cache policies are left for the future work.

3.3.6 Overhead Estimation

The hardware overhead of the proposed techniques using data compression include the compressor and the decompressor. The BΔI compressor and decompressor are implemented using Verilog, and synthesized with Synopsys Design Compiler at $45nm$ technology node. The estimated area cost is $0.0018mm^2$, which is negligible to the entire

chip area. Besides the compressor and decompressor overhead, OMT and CAT also incur some tag overheads. As discussed above, OMT has very small tag overhead, only 2-bit compression scheme code for each cache line; CAT uses 50% more tags to support a larger effective cache capacity.

Data compression also increases the cache access latency and energy. With the system simulation configurations (as shown Table 3.5), compression increases the read latency by 1 cycle, and decompression increases the write latency also by 1 cycle. Moreover, compressing a cache line consumes $0.003nJ$ on average, and decompressing a cache line consumes $0.001nJ$ on average.

3.3.7 Experimental Methodology

In this project, the evaluation is based on the gem5 full system simulator [32] with modification to the cache implementation to simulate the architectural design. The modification to gem5 includes an asymmetric cache read/write latency model and integrated (de)compression unit. The processor and memory configurations are summarized in Table 3.5. Both single-core and four-core systems are simulated with DDR3-1600 as the referenced memory model.15 benchmarks from the SPEC CPU2006 benchmark suite are chosen for single-core simulation. In addition, 10 benchmarks with a wide range of different compression ratio are selected and mixed into 10 groups of 4 for multi-programmed workload simulation as shown in Table 6.1. The accesses per kilo instructions (APKI) is used as a metric to measure the access behavior of LLC, and the APKIs of different workloads are listed in Table 6.1. In the simulation, one billion instructions are fast-forwarded before 5 billion instructions are executed. Note that at least one benchmark with high compressibility is selected for each multi-programmed workload.

Four STT-RAM LLC designs are compared:

Table 3.5: Processor and Memory Configurations.

Processor	1-/4-core, alpha ,4GHz, out-of-order, 8-issue
L1 cache SRAM	private, 32KB I/D, 64B line, 4-way 2-cycle latency
L2 cache SRAM	private, 256KB, 64B line, 8-way 5-cycle latency
L3 cache STT-RAM	shared, 64B line, 16-way 4MB SLC, 8MB MLC
Main memory DRAM	8GB, DDR3-1600, 64bit I/O, 8 banks tCL-tRCD-tRP-tWR: 11-11-11-12

Table 3.6: Evaluated workloads.

No	Benchmark	APKI	No	Benchmark	APKI
1	GemsFDTD	19.38	mix1	4,5,11,15	13.59
2	astar	0.96	mix2	3,6,8,13	9.69
3	bwaves	16.66	mix3	5,11,14,15	10.17
4	bzip2	9.52	mix4	4,6,8,13	10.77
5	dealII	1.73	mix5	3,9,11,14	13.39
6	gobmk	2.92	mix6	4,5,8,15	9.89
7	h264ref	1.92	mix7	3,6,9,13	23.73
8	hmmer	4.52	mix8	3,5,8,11	9.56
9	lbm	55.63	mix9	4,6,13,15	25.69
10	leslie3d	11.97	mix10	3,4,5,14	10.68
11	milc	9.42			
12	sjeng	0.60			
13	soplex	18.52			
14	tonto	0.73			
15	zeusmp	11.50			

- SLC: the SLC STT-RAM cache.
- MLC: the baseline conventional MLC STT-RAM cache.
- OMT: the proposed MLC cache design with overhead-minimized technique.
- CAT: the proposed MLC cache design with capacity-augmented technique.

The STT-RAM LLC configurations are summarized in Table 3.7, where the circuit-level parameters of latency, energy, and leakage power are generated from NVSim [33].

Table 3.7: STT-RAM LLC Parameters.

	SLC	MLC	OMT
Read latency (cycle)	13	19	S: 14, H: 20
Write latency (cycle)	49	90	S: 50, H: 95
Read energy (nJ)	0.415	0.424	S: 0.427, H: 0.579
Write energy (nJ)	0.876	1.859	S: 1.084, H: 2.653
Leakage power (mW)	80.8		

3.3.8 Experimental Results

Energy Results

Figure 3.15 shows the energy consumption normalized to MLC design for single-programmed workloads and multi-programmed workloads respectively. As indicated, SLC design has the lowest energy consumption due to its one-step operation. As shown in Figure 3.15 (a), the OMT scheme reduce system energy by 32.0% on average and up to 70.0% for benchmarks with high compressibility and APKI, e.g. *zeusmp* and *GemsFDTD*. Figure 3.15 (b) shows that the energy consumption for multi-programmed workloads is further reduced by 39.4% on average. This is because applications running on a multi-core system cause more accesses on LLC. Fortunately, OMT avoids compressible cache lines being written into the hard-bit region, which leads to significant energy savings. While CAT design reduce energy by 19.3% on average, since energy saving is traded for potential better performance. Only dynamic energy is considered because the major difference of energy consumption comes from dynamic cache accesses and different cache designs share the similar peripheral circuit.

A major source of energy saving is the write energy reduction from two-step operation to one-step soft-bit only operation. In Figure 3.16, the Fast Write Ratio is the percentage of saved one-step write accesses over total write accesses on LLC and demonstrate that the proposed cache design avoids on average 54.5% of normal two-step writes to one-step

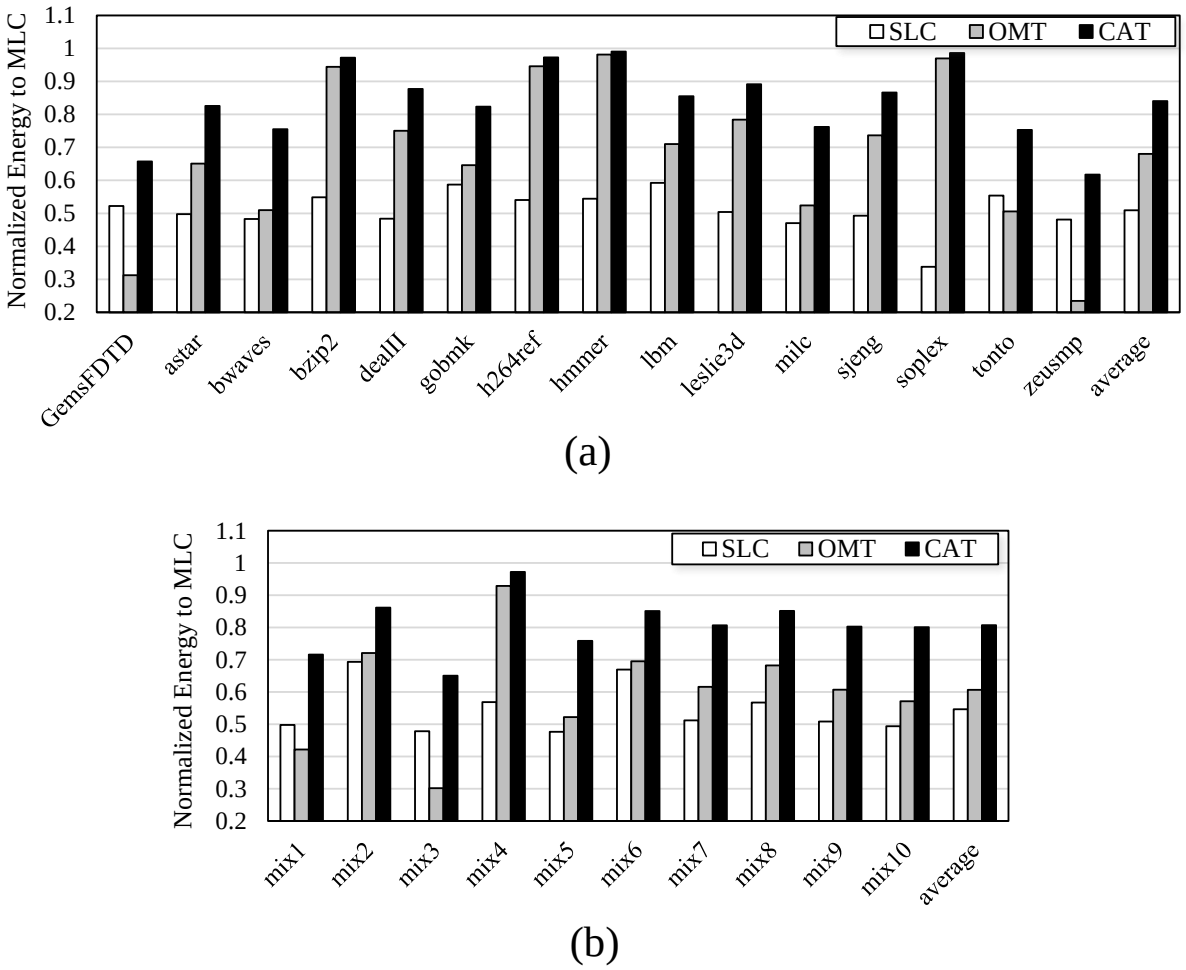


Figure 3.15: Energy results normalized to MLC for: (a) single-programmed and (b) multi-programmed workloads.

programming for shared LLC.

Performance Results

The Instruction Per Cycle (IPC) is used as the metric to evaluate system performance, and the overall throughput ($\sum$IPC) is employed for multi-programmed workloads. First, the effectiveness of MLC design over SLC design is demonstrated. As shown in Figure 3.17 (a), for single-programmed simulation, the SLC design with half of the capacity than MLC design incurs on average 2.8% performance degradation compared with the

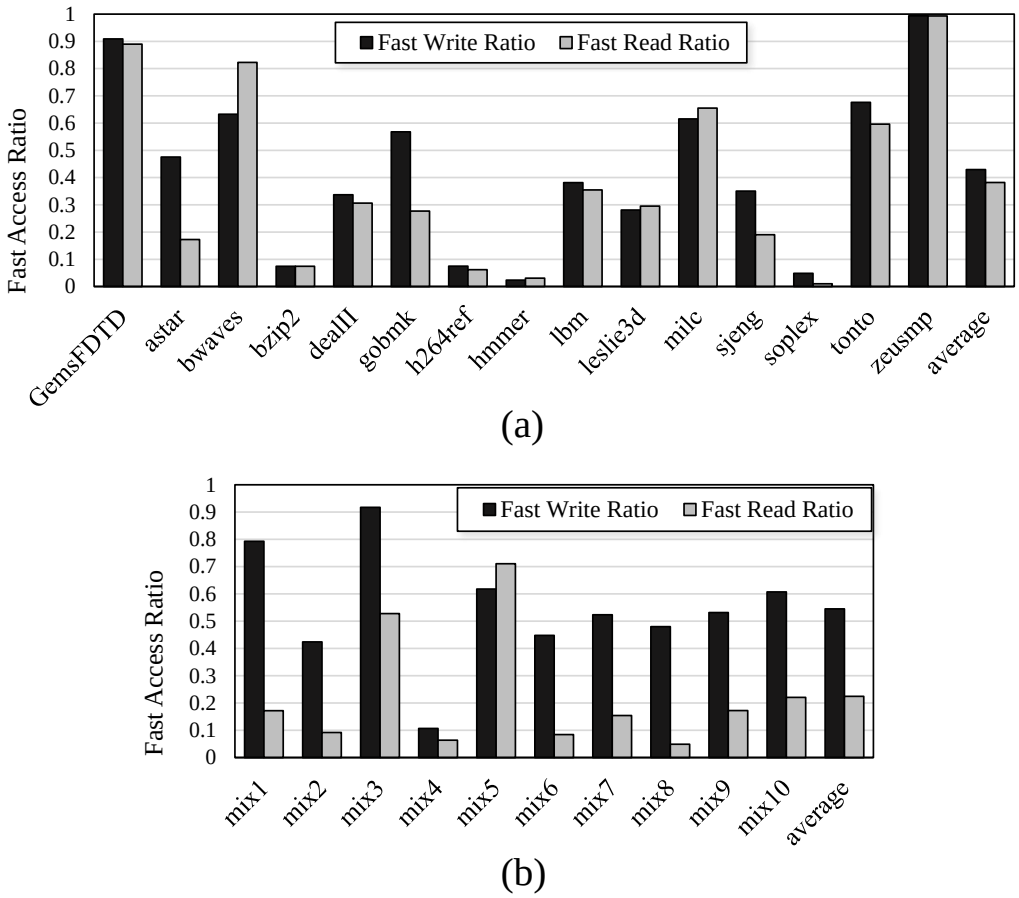


Figure 3.16: Fast access ratio results for: (a) single-programmed and (b) multi-programmed workloads.

baseline MLC design. While the performance benefit of employing MLC design differs among different applications. Compared with SLC design, the system performance of some benchmarks, e.g. *omnetpp* and *hmmer*, actually degrades due to the increased access latency of MLC design, and these benchmarks have a smaller working set on LLC and cannot take much advantage of the increased cache capacity. On the other hand, Figure 3.17 (b) shows that for multi-programmed workloads, the performance of SLC design degrades by 12.1% on average, which comes from the heavier pressure on LLC. Therefore, even though the SLC design consumes the lowest energy, it potentially degrades system performance due to the reduced capacity.

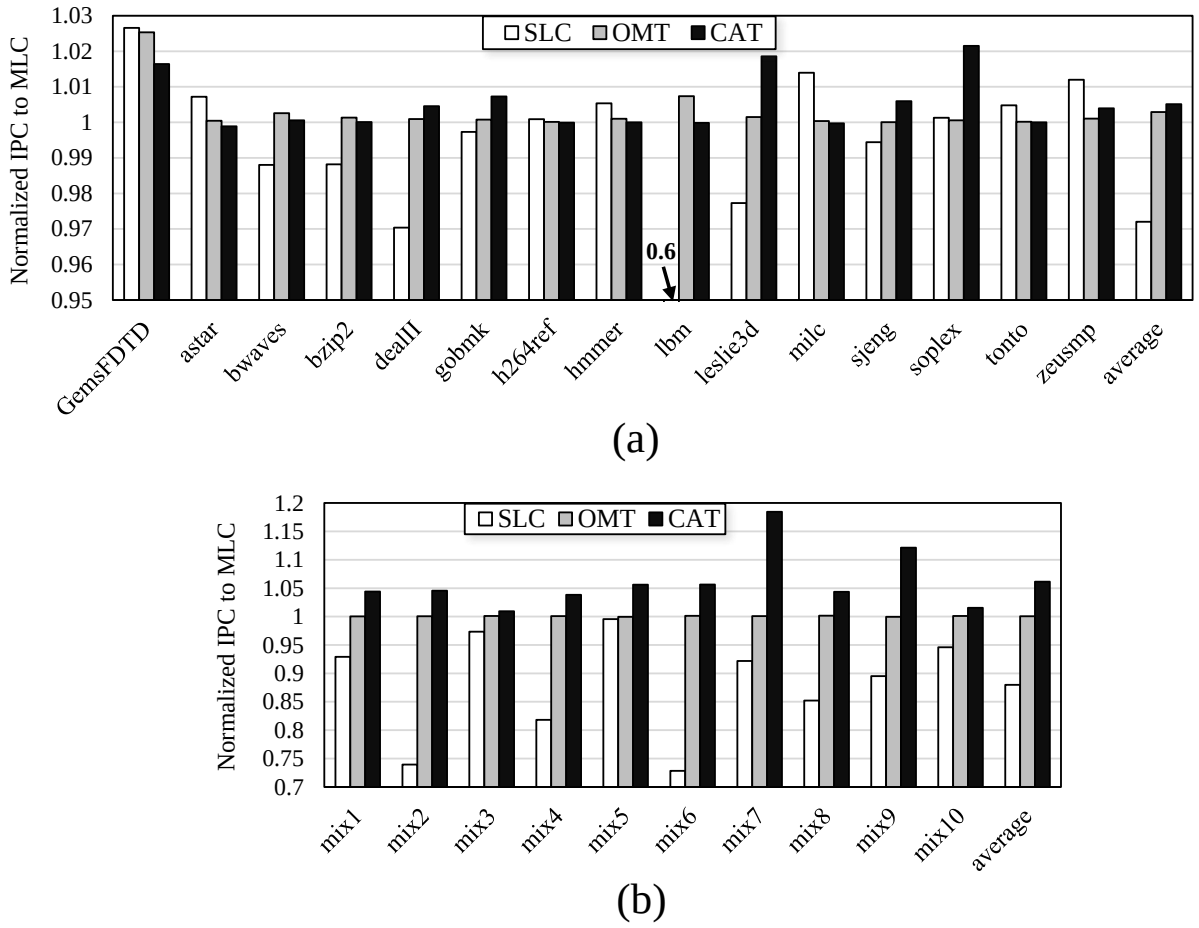


Figure 3.17: IPC results normalized to MLC for: (a) single-programmed and (b) multi-programmed workloads.

The proposed OMT design has the system performance benefit of larger capacity enabled by MLC structure over SLC design. Compared with the baseline MLC design, OMT reduces the energy consumption but incurs no performance degradation. Slightly performance improvement (up to 2.5%) is observed for OMT design since a certain amount of read/write accesses to hard-bit region are eliminated. The performance improvement is insignificant because the target cache is the level-3 cache, which is insensitive to write latency, and the optimization on read latency only has limited impact on system performance. However, it is demonstrated that on average 22.5% (38.2% for single-programmed workloads) of total read accesses are optimized to one-step operation as the Fast Read

Ratio results show in Figure 3.16. Better performance improvement is expected when OMT is applied at higher level of cache hierarchy or when the system is loaded with cache intensive workloads.

Figure 3.17 shows that the CAT design improves system performance by 0.5% for single-programmed benchmarks and by 6.1% for multi-programmed benchmarks, both on average. The effectiveness of the CAT design relies on both compressibility and APKI of different workloads. For workloads with high compressibility and APKI, e.g. *mix7* and *mix9*, it is observed that the performance is improved by over 10.0%. The effective cache capacity is increased when equipped with CAT since the saved hard-bit region can store additional cache lines. Consequently, it reduces memory accesses and improves system performance. While other workloads show little impact on performance, which is due to either low compressibility or insensitive access latency impact on LLC.

3.3.9 Summary

MLC STT-RAM has been widely studied as a replacement of SRAM for constructing large-scale and energy-efficient on-chip cache. However, the two-step read/write accesses of MLC cause energy overhead and system performance degradation. This project proposes two techniques leveraging data compression to optimize series MLC STT-RAM based LLC design. The OMT technique avoids slow and energy-inefficient two-step accesses on both hard-bit and soft-bit regions to one-step accesses on soft-bit region only by applying data compression. The CAT technique increases the effective cache capacity by fitting a second compressible cache line into the saved hard-bit region of STT-RAM cells. The experimental results on the evaluated multi-programmed workloads show that, the cache design with OMT reduces the dynamic energy consumption of LLC by 39.4% on average without performance degradation, while the cache design with CAT improves

system performance by 6.1% and reduces the dynamic energy consumption by 19.3% on average.

Chapter 4

Using MLC STT-RAM for Efficient Local Checkpointing

This chapter presents a project that utilizes MLC STT-RAM for fast and energy-efficient local checkpointing [9]. It is organized as follows. First, the motivation of this project is introduced, and then the related work and the basics of checkpointing. Next, the details of the proposed checkpointing mechanism is presented, where MLC STT-RAM is utilized as main memory for efficient local checkpointing. Next, the experimental methodology and evaluation results are shown before this project is summarized.

4.1 Motivation

In modern large-scale computing systems, high reliability, availability and serviceability (RAS) have been mandatory. However, it becomes more and more challenging to sustain sufficient RAS quality as the node count keeps increasing. Although each component has been designed for a long mean time to failure (MTTF), the entire system encounters a failure every few days or even several hours [34]. As the exascale

era approaches, the MTTF of future large-scale systems is predicted to be only a few minutes [35].

As a well-known error recovery mechanism, checkpointing has been widely used to protect those systems from unexpected failures [36]. In the typical organization of a contemporary supercomputer as shown in Figure 4.1, the process nodes usually deploy DRAM as their main memory, and access the global storage via the I/O nodes. The global storage is permanent storage, usually built with NAND flash based solid state drives (SSDs) or conventional hard disk drives (HDDs), but DRAM is volatile memory and hence vulnerable to errors. One state-of-the-art checkpointing approach is to save the entire process address space to the non-volatile global storage periodically. However, the limited data transfer bandwidth between DRAM and the global storage and the centralized checkpointing topology prevent the performance scaling in future large-scale computing systems. As the application size grows and the system failure rate increases, larger-size and more frequent checkpoints are required, and consequently a significant portion of execution time will be spent writing checkpoints. For instance, checkpointing may incur over 50% performance overhead in a petaFLOPS system [37].

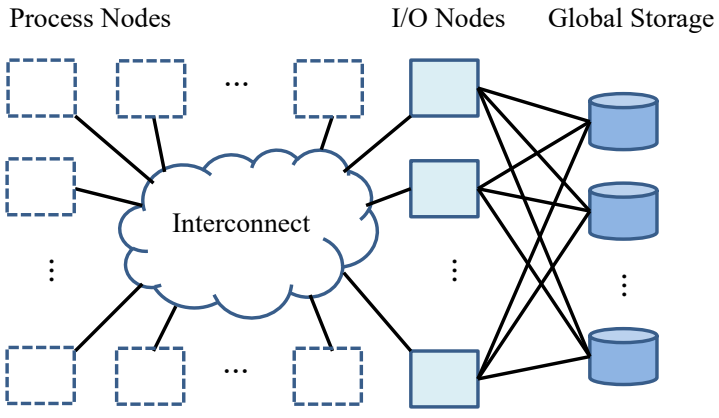


Figure 4.1: The typical organization of a contemporary supercomputer [36].

To address the scalability issue, local/gloabl hybrid checkpointing [38] and multi-level

checkpointing [39] have been proposed as scalable solutions, which combine frequent and fast local checkpoints to local storage with less frequent and slow global/remote checkpoints to global storage. The effectiveness of these approaches is determined by how many failures can be recovered from fast local checkpoints. In fact, most transient errors affect only the failure node, and the system can be simply recovered by rebooting the failure node with its local checkpoint. Dong *et al.* made an estimation for a petaFLOPS system that 83.9% of failures only need local checkpoints [38].

To achieve better performance and higher power efficiency in local checkpointing, some recent work has proposed to utilize the emerging non-volatile memories (NVMs), such as PCM and STT-RAM, as the local storage of each process node thanks to their non-volatility, fast random read access and zero standby leakage power [38, 40].

This project proposes to leverage MLC STT-RAM as the main memory for fast local checkpointing. MLC opens up an inherent multi-versioning opportunity for checkpointing since each cell can store both the working data and the checkpoint data simultaneously [41]. Consequently, the system can be easily recovered from the local checkpoint by moving data within memory cells, which substantially eliminates the data transfer overhead between main memory and backup storage. Previous work has also considered using MLC PCM for in-memory local checkpointing [41]. However, MLC PCM write operations adopt the program-and-verify (P&V) technique, which iteratively apply partial set pulses and verify if a specified precision criterion is met [42], to achieve desirable intermediate resistance levels. Due to the P&V process, MLC PCM has even longer write latency, higher write energy, and lower program throughput than SLC PCM; hence using MLC PCM as main memory would significantly degrade system performance. Different from MLC PCM, writing MLC STT-RAM is relatively simple, taking two steps at most [18]. This proposal takes advantage of the unique characteristics of MLC STT-RAM write operations and ensures that only one-step writes occur during the entire execution

time (including checkpoint and recovery). The evaluation results demonstrate the great potential of using MLC STT-RAM as main memory for fast and energy-efficient local checkpointing.

4.2 Related Work

Prior work has leveraged NVMs, such as PCM, as local storage/memory in each process node to reduce the checkpoint overhead [38, 40]. They are used to save local checkpoints, and also to store their neighbor nodes' global/remote checkpoints. Figure 4.2 (a) depicts the typical structure of a process node in a contemporary supercomputer, which employs DRAM as its main memory; Figure 4.2 (b) illustrates the node structure leveraging SLC PCM as local storage or extended memory.

In **local/global hybrid checkpointing** proposed by Dong *et al.* [38], PCM is used as local storage. The state of each node is backed up in their own persistent storage periodically, and after several local checkpoints, a global checkpoint is created in the background to the global storage or the neighbors' local storage. In **NVM-checkpoint** proposed by Kannan *et al.* [40], NVM (PCM) is used as extend memory, but is not directly exposed to applications. Specialized NVM interface is provided for applications to write checkpoints, and OS-level support is given to manage the data movement across the DRAM/NVM boundary. Their experimental results showed that both the hybrid checkpointing and NVM-checkpoint can efficiently reduce the checkpoint overhead by utilizing PCM-based local storage/memory. Nevertheless, the checkpoint latency is still limited by the bandwidth of writing checkpoints from DRAM to PCM. Although the sophisticated PCM-DIMM design and the novel 3D-PCM scheme (deploying PCM directly atop DRAM) can overcome the limited data transfer bandwidth between DRAM and PCM, the real bottleneck is PCM's low program/write throughput due to its high write power

and long write latency. In this work, MLC STT-RAM is leveraged as the persistent main memory in each process node, which provides higher write throughput than PCM. The proposed node structure is shown in Figure 4.2 (c). During local checkpointing, data are transferred within memory cells, from one bit of each cell to another bit, and no data movement is required between different memories. Therefore, the checkpoint overhead can be significantly reduced.

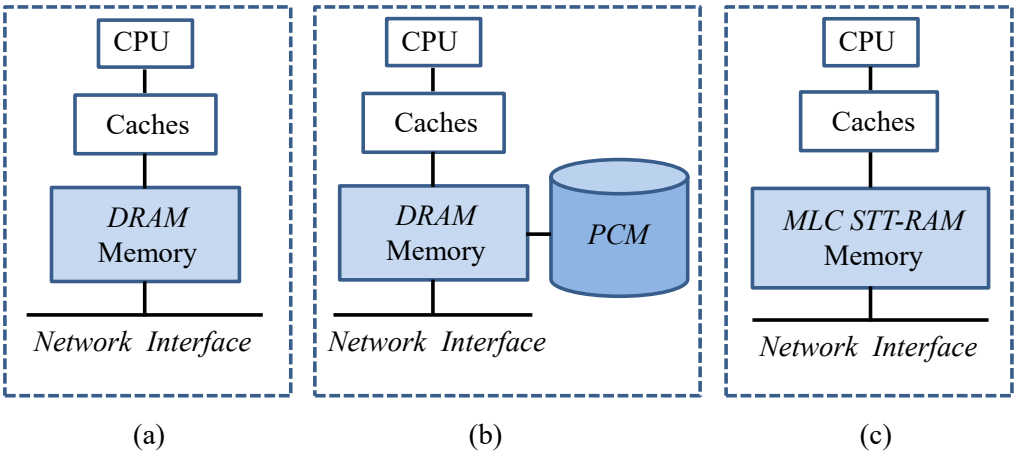


Figure 4.2: (a) The typical structure of a process node; (b) the process node structure leveraging PCM as local storage/memory [38, 40]; (c) the process node structure proposed in this project.

The idea of using a multi-level cell for local checkpointing has been presented by Yoon *et al.* in a patent [41], which gives an example that working data are stored in a first level and checkpoint data are stored in a second level. However, they broadly consider all desired types of MLC NVMs, including flash, PCM and STT-RAM, but did not focus on a specified memory technology. In this work, MLC STT-RAM is leveraged as persistent main memory and take advantage of its unique features that other memory technologies do not have to design an efficient local checkpointing. Furthermore, a more complete solution is provided, and its performance and energy consumption are evaluated.

4.3 The Basics of Checkpointing

Error recovery mechanisms can be classified into forward error recovery (FER) and backward error recovery (BER), and BER is also called *checkpointing* or *rollback recovery* [43]. FER prepares a system for possible future errors, by taking extra tasks with redundant hardware. Triple-modular redundancy (TMR) is the most popular example, in which three systems perform the same process simultaneously and output a single result through majority-voting. Checkpointing attempts to restore a system to an earlier error-free state (*i.e.* a recovery point) after an error occurs by recording recovery information during execution. Checkpointing requires less hardware redundancy than FER. However, it is associated with three overheads: the performance overhead during error-free execution, the storage overhead for recovery information recording, and the recovery overhead to retrieve lost work.

Prvulovic *et al.* designed a taxonomy [43] that classifies checkpointing schemes according to three characteristics: how to achieve checkpoint consistency (globally or locally), how to separate checkpoint from working data (fully or partially), and where to store checkpoint (in external storage or in internal memory). Their categorization can be extended with one more feature: the transparency to applications (transparent or application-initiated) [40]. To avoid the overhead of specialized NVM interface and OS support in NVM-checkpoint, the lightweight local/global hybrid checkpointing is adopted to evaluate the proposed MLC STT-RAM based local checkpointing mechanism in this work. The utilized hybrid checkpointing is located in the design space of checkpointing as below.

Global or Local Consistency. Global checkpointing requires all the process nodes to synchronize to generate a global checkpoint periodically [44]. In local checkpointing, each process node creates their own checkpoints either in a coordinated way [45] or in

an uncoordinated fashion [46]. Coordinated schemes require the nodes that have been interacting with each other to generate their own checkpoints at the same time, while uncoordinated schemes do not. In the hybrid checkpointing, each process node maintains its local checkpoint coordinately and a global checkpoint is periodically generated from an existing local one.

Safe External or Internal Storage. The checkpoint can be stored either in safe external storage (usually disk arrays) [46], or in safe main memory or internal storage [47]. In-memory checkpointing is faster than in-disk checkpointing, but requires more hardware support to assure the data safety and fault-tolerance. In the hybrid checkpointing, both the internal storage (MLC STT-RAM) and external storage (global storage) are employed as the safe backup.

Full or Partial Separation. Full separation means that the checkpointing data is completely separated from working data [45]. For example, it can be achieved by simply copying the entire memory state to another place. With regards to the fact that there may be not much change between two sequential checkpoints, incremental checkpointing has been studied to reduce the checkpointing overhead [48]. Other optimizations to reduce the copying size have been explored, such as memory exclusion [49]. On the other hand, partial separation keeps the checkpointing and working data as the same one, and uses buffering, renaming or logging schemes to record state modifications since the last checkpoint. Even though it can reduce the storage overhead, partial separation increases performance or recovery overhead. The hybrid checkpointing employs full separation, and either the entire copying or incremental checkpointing can be applied.

Transparent or Application-initiated. Transparent checkpointing simply saves the entire process address space, and thus does not require application developers to explicitly handle failures in their algorithm design [50]. Application-initiated checkpointing requires application developers to identify which data to be stored in their code, and

hence can reduce the storage overhead [40]. In the hybrid checkpointing, transparent checkpoints are preferred since the advantages of MLC STT-RAM can be fully utilized and there is no need to restructure the code of applications.

4.4 The Proposed Mechanism

The proposed checkpointing mechanism is presented in this section. This section first introduces the MLC STT-RAM main memory design, and then describes the local checkpointing scheme using MLC STT-RAM. The overall hybrid checkpointing mechanism is explained, before finally the storage overhead is discussed.

4.4.1 MLC STT-RAM Main Memory

Recent work has evaluated SLC STT-RAM as a feasible main memory alternative, and demonstrated that with optimized write operations, STT-RAM can provide comparable performance to equal-capacity conventional DRAM and reduce energy consumption by 60% [7]. This work utilizes MLC STT-RAM as the main memory of each process node. Although MLC STT-RAM offers the opportunity of a higher memory capacity than SLC STT-RAM within the same area constraint, it may degrade performance and increase energy consumption since it has more complex write operations. From Section 3.1, we know that for MLC STT-RAM (either parallel or series), writing soft-bits is fast and energy-efficient, because it requires a small write current and finishes in one step; writing "00" or "11" is also a one-step operation but requires a larger write current; the other cases which have to take two steps are the worst ones in terms of both write latency and energy. When writing a memory row or even a small portion of it, it is most likely that at least one cell requires two steps; therefore, the memory system performance would degrade compared with using SLC STT-RAM. In the MLC STT-RAM main memory

design, the soft-bit of each cell is used as operated memory to keep working data, and the hard-bit is employed to store a checkpoint for the soft-bit. Therefore, during error-free execution, only soft-bits are updated; during checkpoint and recovery, only values "00" and "11" are written. This design ensures that only one-step write operations occur throughout the entire execution time.

The bank organization of the MLC STT-RAM main memory design is shown in Figure 4.3. Different from conventional STT-RAM bank organization, the sense amplifiers and write drivers are controlled by the "Mode" signal. They can work in four modes: 1) normal read, 2) normal write, 3) checkpoint, and 4) recovery. Since the operated memory is soft-bits, in normal read or write mode, only the soft-bits of memory cells are read out or updated. In checkpoint mode, first the soft-bits of memory cells (working data) are sensed, and then values of "00" or "11" are written to the cells according to the values of their soft-bits. In recovery mode, first the hard-bits of memory cells (checkpoint data) are sensed, and then values of "00" or "11" are written to the cells according to the values of their hard-bits. The data flow of the mode control of sense amplifiers and write drivers is shown in Figure 4.4. To clearly illustrate different modes, duplicated sense amplifiers and write drivers are drawn; but in reality, the same sense amplifier and write driver are used to handle both the hard-bit and the soft-bit of a cell.

4.4.2 Local Checkpointing

Figure 4.5 presents an example to illustrate the proposed local checkpointing mechanism using MLC STT-RAM as main memory. As shown in Figure 4.5, each cell contains two bits: the upper one is the soft-bit which stores the working data; and the lower one is the hard-bit which stores the checkpoint data. At Checkpoint i, both the working and checkpoint data values are "0101...0101". The subsequent writes (from i_1 to i_m) only

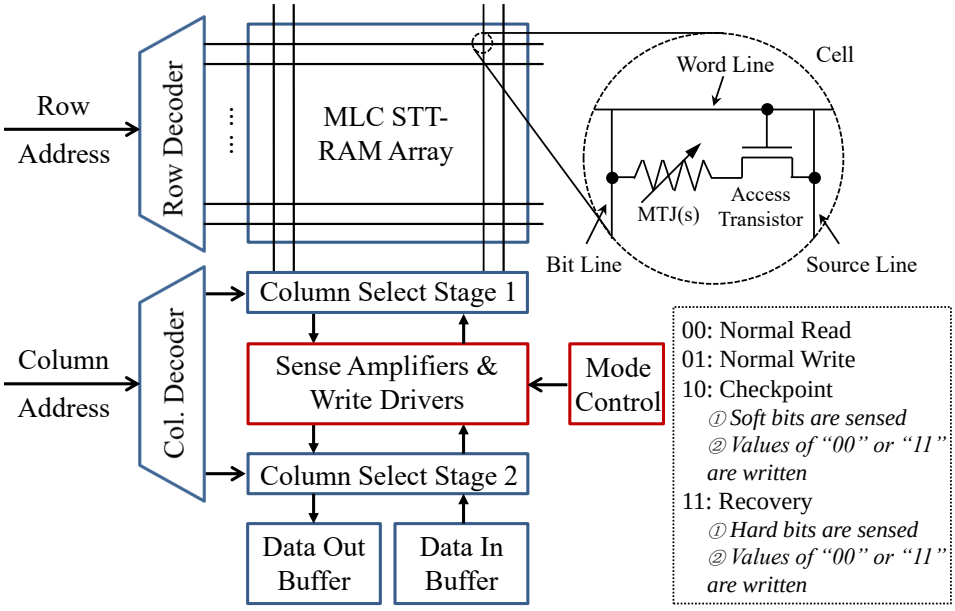


Figure 4.3: The bank organization of MLC STT-RAM main memory. The sense amplifiers and write drives are controlled by the "Mode" signal which has four different states.

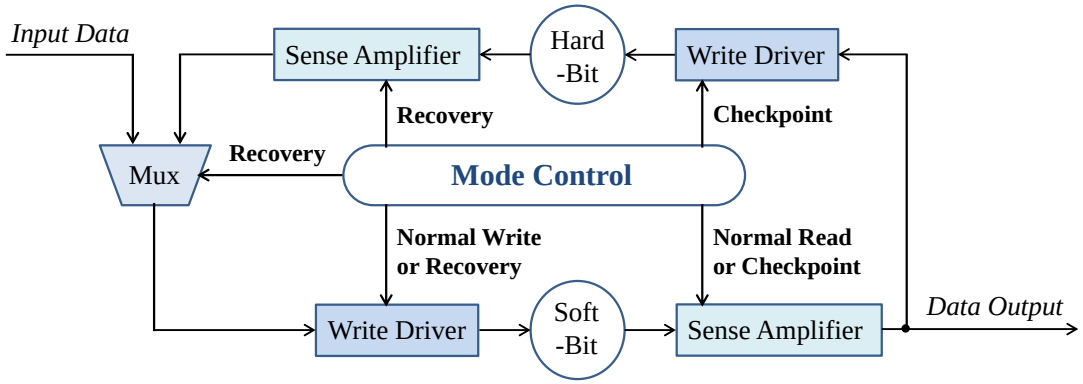


Figure 4.4: The data flow diagram of the mode control of sense amplifiers and write drivers.

work on the soft-bits with small write currents, and the hard-bits will not change under those small write currents. At Checkpoint $i+1$, first the dirty cache data will be dumped into the operated memory (soft-bits), and then the checkpoint data (hard-bits) should be updated by mirroring the current working data. This internal data transfer between the two bits of each cell can be accomplished by the sense amplifiers and write drivers working in checkpoint mode as described in Figure 4.3. With the data comparison write property,

many unnecessary writes can be saved at bit level, and incremental checkpointing can be implemented naturally without other hardware of software overheads. Therefore, it is quite lightweight to create local checkpoints by using MLC STT-RAM as main memory, and the efficiency of the scheme derives from the unique features in MLC STT-RAM write operations.

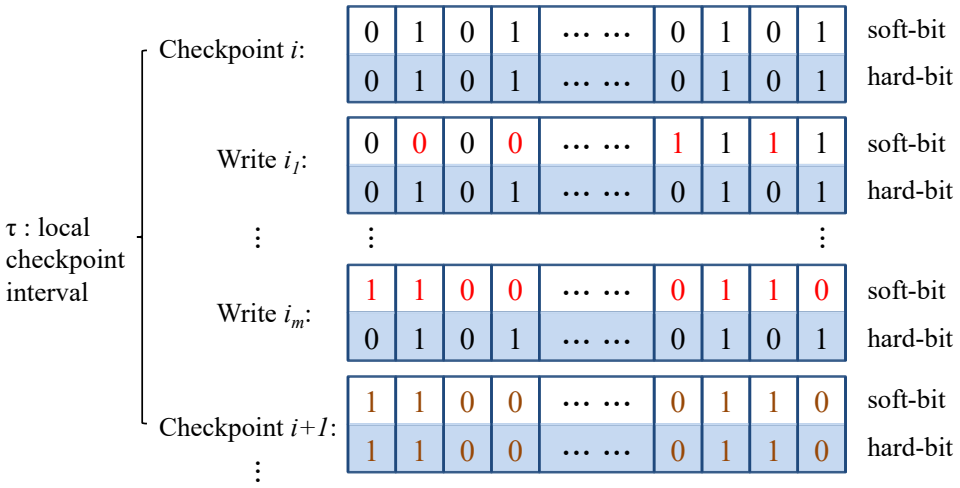


Figure 4.5: An example to illustrate the proposed local checkpointing mechanism using MLC STT-RAM.

4.4.3 The Overall Hybrid Checkpointing

The lightweight local/global hybrid checkpointing [38] is adopted in this work to achieve high RAS for large-scale computer systems. MLC STT-RAM is utilized as main memory and a local checkpoint is recorded in the hard-bits of memory cells. The global checkpoints can be saved in the global storage which is managed by the I/O nodes as shown in Figure 4.1, or in the local storage of the neighbor nodes. If they choose the second way, each process node should possess their own local storage to store the global checkpoints of their neighbors. The local storage can be built with flash, HDD or PCM.

The local/global hybrid checkpointing is demonstrated in Figure 4.6, and the parameters are explained in Table 4.1. During error-free execution, as shown in Figure 4.6

(a), after every local checkpoint interval (τ), each process node makes a local checkpoint which takes a variable time δ_L. At the end of every global checkpoint interval, a global checkpoint is created after a local checkpoint which takes δ_G. All the process nodes do their checkpointing in a coordinated way. Two parameters are critical to balance checkpoint overhead against recovery overhead in the hybrid checkpointing mechanism: the local checkpoint frequency and the local/global checkpoint ratio (*i.e.* how many local checkpoints are made during a global checkpoint interval). The effects of those two parameters have been studied in depth by Dong *et al.* [36]. With this efficient local checkpointing mechanism using MLC STT-RAM, the local checkpoint time overhead (δ_L) and energy consumption can be reduced.

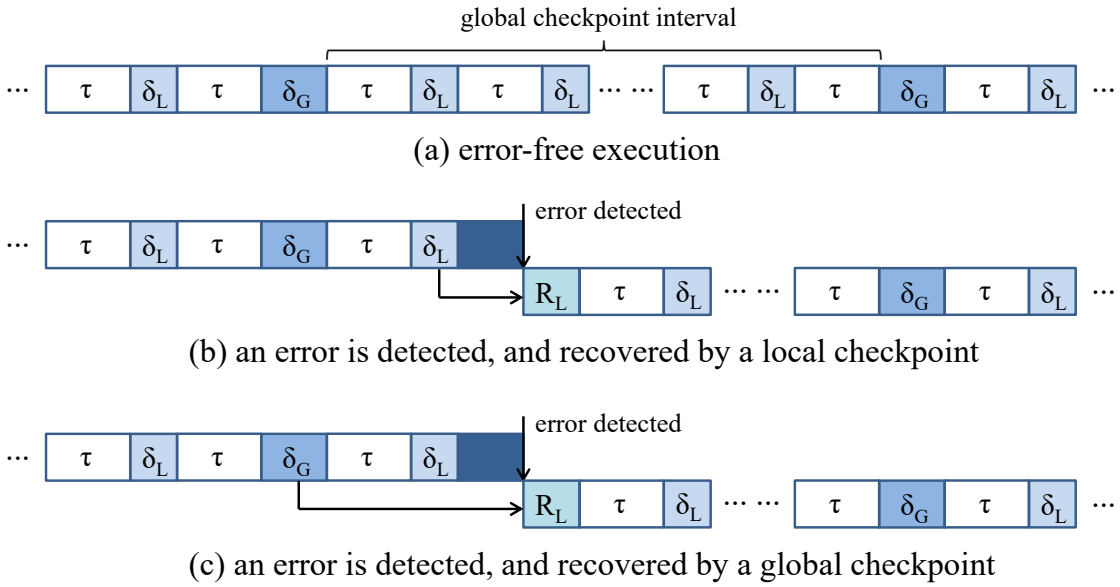


Figure 4.6: The time-lines of error-free execution and recovery from an error by a local or global checkpoint in the hybrid local/global checkpointing [38].

Figure 4.6 (b) and (c) show the time-lines of recovery by a local and a global checkpoint respectively when errors are detected. It is assumed that the error detection latency has an upper bound of a few cycles. Many types of faults can cause a system failure, *e.g.* software bugs, human misoperations, network congestions, hardware damages, *etc.*

Table 4.1: Hybrid Checkpointing Parameters

τ	the local checkpoint interval
δ_G	the global checkpoint time
δ_L	the local checkpoint time
R_G	the global checkpoint recovery time
R_L	the local checkpoint recovery time

In this work, errors can be classified into two categories: those that can be recovered by a local checkpoint, and those that have to be recovered by a global checkpoint. With a transient error, it is probable to restore the system from a local checkpoint. However, with a permanent error, the process node might have to be replaced, and a global checkpoint may be able to recover the system. In the checkpointing mechanism, the latest checkpoint version is stored locally, and multiple checkpoint versions can be maintained in global storage.

When rolling back to a local checkpoint as shown in Figure 4.6 (b), it needs to load all the local checkpoint data to the operated memory. With the proposed checkpointing mechanism using MLC STT-RAM as main memory, it only requires to copy the hard-bits of the memory cells in the process address space to their soft-bits, which can be simply accomplished by the sense amplifiers and write drivers working in recovery mode as described in Figure 4.3. This internal data transfer between the two bits of each cell is quite lightweight, compared to moving data from local PCM storage to DRAM memory. Therefore, the checkpointing mechanism can also reduce the recovery time from a local checkpoint (R_L) and its energy consumption.

Recovering by a global checkpoint suffers from extra rollback overhead compared with recovering by a local checkpoint, as displayed in Figure 4.6 (c). On one hand, loading a global checkpoint is more costly; on the other hand, more lost work needs to be retrieved from a global checkpoint. Therefore, the effectiveness of the local/global hybrid checkpointing depends on how many failures can be recovered by local checkpoints. For

a petaFLOPS system, it has been estimated that 83.9% of failures can be recovered by local checkpoints [38].

4.4.4 Storage Overhead Discussion

The proposed checkpointing mechanism uses the hard-bit of each cell to save a checkpoint for the soft-bit in the MLC STT-RAM main memory. Therefore, the actual usable memory capacity is half the total capacity of MLC STT-RAM. However, it has been considered reasonable to reserve a 1 : 1 capacity ratio for checkpoint storage and operated memory [38]. Since not all the working data necessarily need a checkpoint, to better utilize the memory capacity, future work may try to divide the MLC STT-RAM main memory into two regions, either statically or dynamically, in which one region is reliability-oriented, and the other is capacity-oriented. The reliability-oriented region is used as in this work; whereas in the capacity-oriented region, both bits of a cell can be exploited to store the working data and no local checkpoint is required. Therefore, the density advantage of MLC STT-RAM can be better used, and the actual available memory capacity can be increased.

4.5 Evaluation

In this section, the experimental methodology and the evaluation results are presented.

4.5.1 Experimental Methodology

Gem5 simulator [32] is used as the simulation platform. Two local checkpointing mechanisms are evaluated, one leveraging MLC STT-RAM as main memory (denoted

by MLC STT-RAM), and the other utilizing DRAM as main memory and PCM as local storage (denoted by DRAM-PCM). The processor and memory configurations of a process node are summarized in Table 4.2. Both single-core and 4-core processors are simulated, and DDR3-1600 is used as the reference memory model. Unlike DRAM, STT-RAM does not need refresh operations due to its non-volatility, and because its read operation is non-destructive, there is no need to rewrite data after reading. It is conservatively assumed that MLC STT-RAM has the same read latency with DRAM, and its one-step write takes 10 ns extra latency *vs.* DRAM. Table 4.3 lists the workloads used in the evaluations. For single-core simulation, fifteen benchmarks from the SPEC CPU2006 suite are selected [51], including both integer and float point applications. Additionally, the single benchmarks are mixed and assigned to a 4-core processor for multi-programmed simulation.

Table 4.2: Processor and memory configurations

Processor	1-core/4-core, x86, 2GHz, out-of-order, 8-issue
L1 cache	private, 32KB I/D-L1, 64B line, 4-way, 2-cycle latency
L2 cache	shared, 4MB, SRAM, 64B line, 16-way, 20-cycle latency
Memory	4 GB, DDR3-1600, 800 MHz, 64 bit I/O, 8 banks DRAM: 10 ns read/write latency, 12.8 GB/s peak I/O bandwidth MLC STT-RAM: 10 ns read latency, 20 ns one-step write latency

The performance overhead of local checkpointing is calculated as the sum of two parts: the time to flush all the dirty data in cache to main memory, and the time to copy all the working data in the process address space to the checkpoint storage, as shown in Equation 4.1. The amount of dirty data in cache and the process address space in the operated memory can be tracked by gem5, and the actual data transfer bandwidth is assumed to be 80% of the peak bandwidth (BW) on average. The experimental results indicate that the first term in Equation 4.1 is negligible compared to the second term

Table 4.3: Evaluated workloads

No	Application	Mix	Applications
1	astar	mix1	1,2,3,4
2	bwaves	mix2	5,6,7,8
3	cactusADM	mix3	9,10,11,12
4	gcc	mix4	13,14,15,1
5	GemsFDTD	mix5	3,4,5,6
6	gromacs	mix6	7,8,9,10
7	h264ref	mix7	1,3,5,7
8	lbm	mix8	2,4,6,8
9	leslie3d	mix9	9,11,13,15
10	mcf	mix10	10,12,14,2
11	milc		
12	namd		
13	omnetpp		
14	sjeng		
15	soplex		

even when the local checkpoint interval is as small as $1s$.

$$Time_{checkpoint} = \frac{Data_{dirty-in-cache}}{BW_{operated-memory} \times 80\%} + \frac{Data_{in-operated-memory}}{BW_{checkpoint-storage} \times 80\%}. \quad (4.1)$$

The peak data transfer bandwidth from DRAM to PCM ($BW_{DRAM-PCM}$) and that from soft-bits to hard-bits ($BW_{soft-hard}$) in MLC STT-RAM are estimated as follows. For a fair comparison, it is assumed that PCM and MLC STT-RAM have the same chip configuration and chip power budget with DRAM. Therefore, they have the same write current limitation of ∼168 mA for all the eight banks per chip as modern DDR3 DRAMs [52]. Given that the write (reset) current of PCM is 300 μA per bit and the write (set) latency is 150 ns [13], a PCM chip could only write 560 bits (168 mA / 300 μA per bit) at a time, and the write bandwidth per chip is 0.467 GB/s (560 bits / 150 ns), which is far lower than DRAM's (*e.g.* 8.53 GB/s for DDR3-1066 and 12.8 GB/s for DDR3-1600). Since there are eight chips per rank, the total write bandwidth of PCM

can be 3.73 GB/s. Obviously, $BW_{DRAM-PCM}$ is limited by the write bandwidth of PCM. The one-step write bandwidth of MLC STT-RAM can be calculated in a similar way, given that the write (hard transition) current of MLC STT-RAM is 266 μA per bit and the one-step write latency is 10 ns [18]. The estimated $BW_{soft-hard}$ of MLC STT-RAM is 63.16 GB/s, about 17× of $BW_{DRAM-PCM}$.

4.5.2 Experimental Results

Performance Overhead in Error-Free Execution

First the performance overheads of the two local checkpointing mechanisms are evaluated with a medium local checkpoint interval $\tau = 5s$. The results of a single-core process node running a single application are displayed in Figure 4.7. The checkpoint overhead of each application depends on the total amount of checkpointing data to be written every τ. Some applications require larger address space and consequently have more checkpointing data, like *bwaves*, *cactusADM* and *GemsFDTD*, as shown in Figure 4.7. The average performance overheads of the two mechanisms are both small. The DRAM-PCM mechanism encounters 1.65% performance overhead on average, and the MLC STT-RAM mechanism incurs only 0.097%. However, as the numbers of cores and running applications increase, more memory space is required, and more checkpointing data need to be written accordingly. The performance overheads in a multiprogrammed 4-core process node are evaluated, and the results are shown in Figure 4.8. The average performance overhead of the DRAM-PCM mechanism is 7.79%, whereas it is only 0.46% with the MLC STT-RAM mechanism. Therefore, the efficiency of the proposed local checkpointing mechanism leveraging MLC STT-RAM as main memory is revealed when the core count of a process node is increased.

To reduce the recovery overhead, smaller τ is preferred in local checkpointing. The

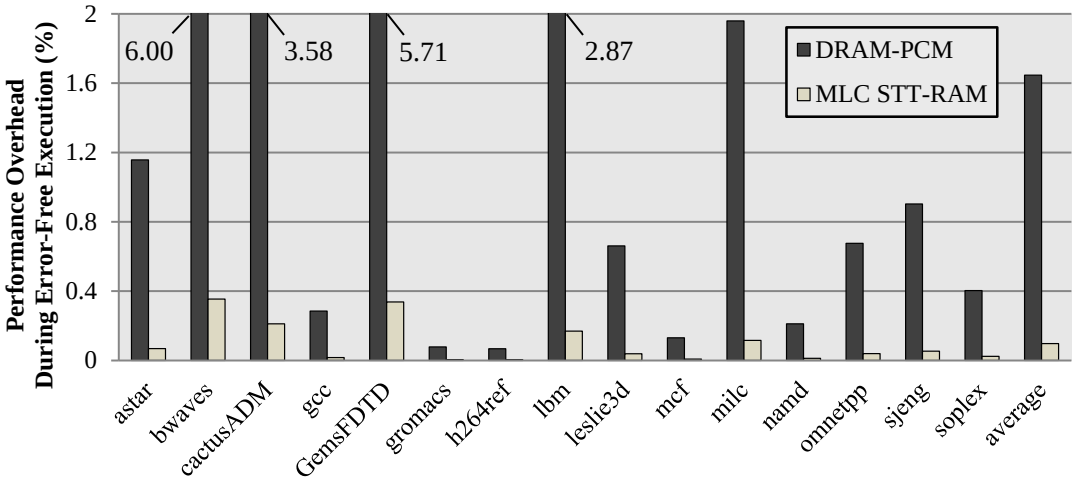


Figure 4.7: Performance overhead of local checkpointing during error-free execution in a single-application and single-core process node with $\tau = 5s$.

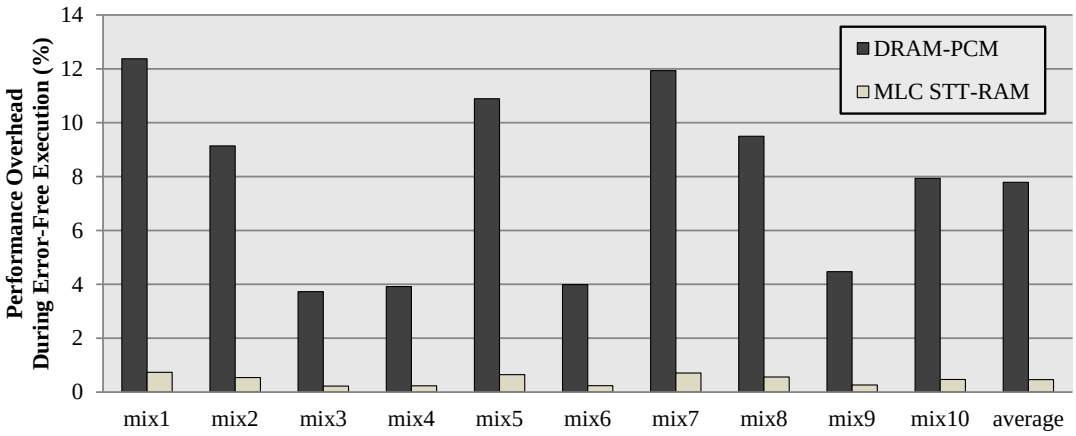


Figure 4.8: Performance overhead of local checkpointing during error-free execution in a multiprogrammed 4-core process node with $\tau = 5s$.

performance overheads of the two mechanisms with $\tau = 1s$ are also evaluated, and the results of single-core and 4-core simulations are presented in Figs. 4.9 and 4.10, respectively. In a single application and single-core process node, the average performance overhead of MLC STT-RAM is 0.26%, and it becomes 0.90% in a multi-programmed 4-core node. Compared to the results of $\tau = 5s$, the checkpoint overheads increase but are still quite small. However, limited by the write bandwidth of PCM, the average performance overheads of the DRAM-PCM mechanism for a single-core node and for

a 4-core node are increased to 4.49% and 15.2%, respectively. Therefore, as the local checkpointing interval τ decreases, the advantage of the MLC STT-RAM mechanism becomes more apparent.

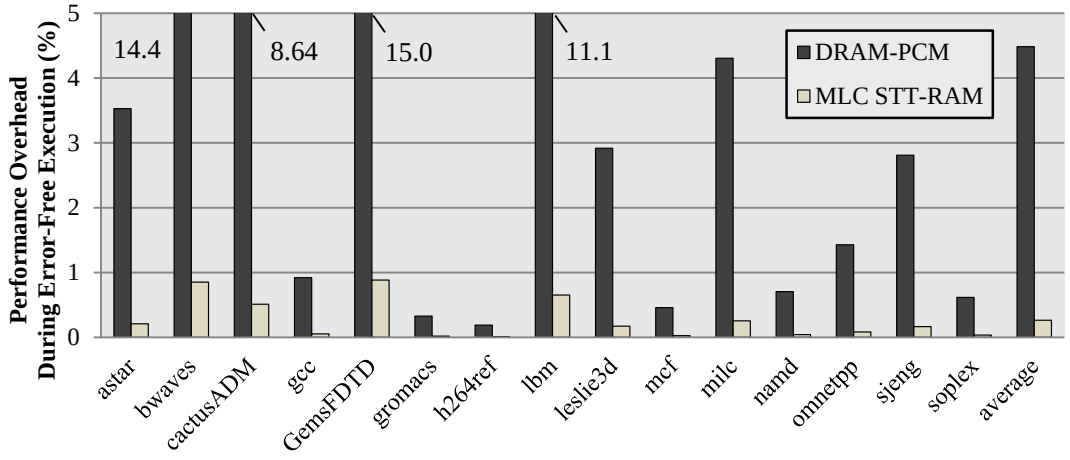


Figure 4.9: Performance overhead of local checkpointing during error-free execution in a single-application and single-core process node with $\tau = 1s$.

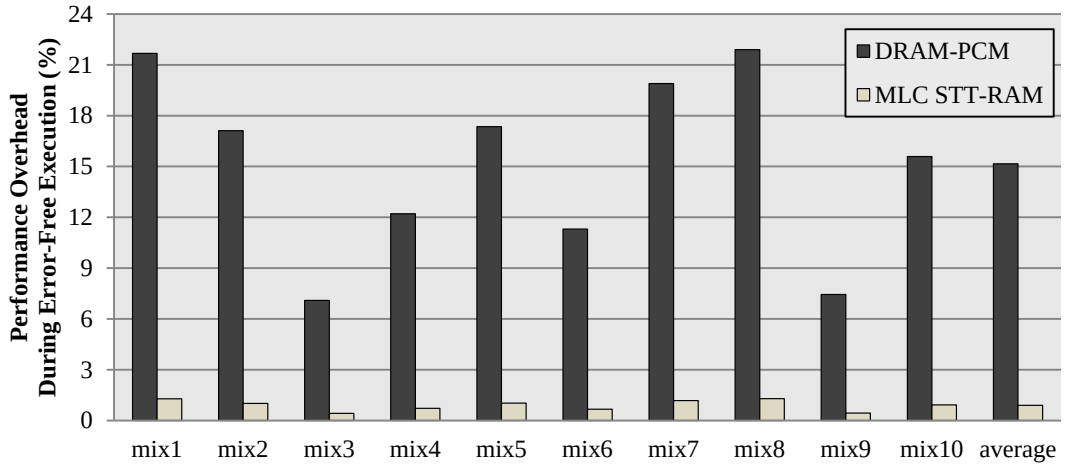


Figure 4.10: Performance overhead of local checkpointing during error-free execution in a multiprogrammed 4-core process node with $\tau = 1s$.

Energy Consumption in Local Checkpointing

The energy efficiency of the MLC STT-RAM local checkpointing mechanism is evaluated by comparing it with the DRAM-PCM mechanism. The parameters used are from the reported data in prior work [13, 18], as listed in Table 4.4. For a fair comparison, the parameters of PCM and MLC STT-RAM are chosen for the same technology node. In Table 4.4, the read and write energy per bit of PCM and MLC STT-RAM do not include the energy consumed by peripheral circuitry. For PCM, given that writing zeros (RESET) and writing ones (SET) have equal possibilities, the average write energy per bit is $(13.5 + 19.2)/2 = 16.35\ pJ$; and the energy consumed by peripheral circuitry per bit for a read or write operation is 0.47 pJ [13]. For MLC STT-RAM, a soft transition (ST) requires a small write current while a hard transition (HT) requires a larger write current; therefore, the write energy of ST per bit is lower than that of HT. It is assumed that MLC STT-RAM has the same peripheral circuitry energy consumption with PCM for each read and write operation.

Table 4.4: Memory energy parameters [13, 18]

	PCM	MLC STT-RAM	DRAM
read (pJ/bit)	2.00	0.38	1.56
write (pJ/bit)	SET 13.5 RESET 19.2	ST 1.92 HT 3.192	0.39
peripheral (pJ/bit)	0.47	0.47	—

The total memory energy consumption of the DRAM-PCM mechanism consists of the energy consumed by writing the dirty data in cache to DRAM, and that to copy data from DRAM to PCM. As to the MLC STT-RAM mechanism, it is composed of the energy to write the dirty data in cache to the soft-bits of MLC STT-RAM, and that consumed by mirroring the soft-bits to the corresponding hard-bits. The results of a single-application and single-core process node and of a multiprogrammed 4-core

process node are presented in Figure 4.11 and Figure 4.12, with $\tau = 5s$. Figure 4.11 displays that, the average memory energy consumption of the DRAM-PCM mechanism for a single-core process node is 0.024 J, whereas that of the MLC STT-RAM mechanism is only 0.005 J. As shown in Figure 4.12, for a multiprogrammed 4-core process node, the average memory energy consumption of the DRAM-PCM mechanism increases to 0.116 J, while that of the MLC STT-RAM mechanism is 0.025 J. Therefore, the MLC STT-RAM mechanism saves more than three quarters of the memory energy consumed by the DRAM-PCM mechanism.

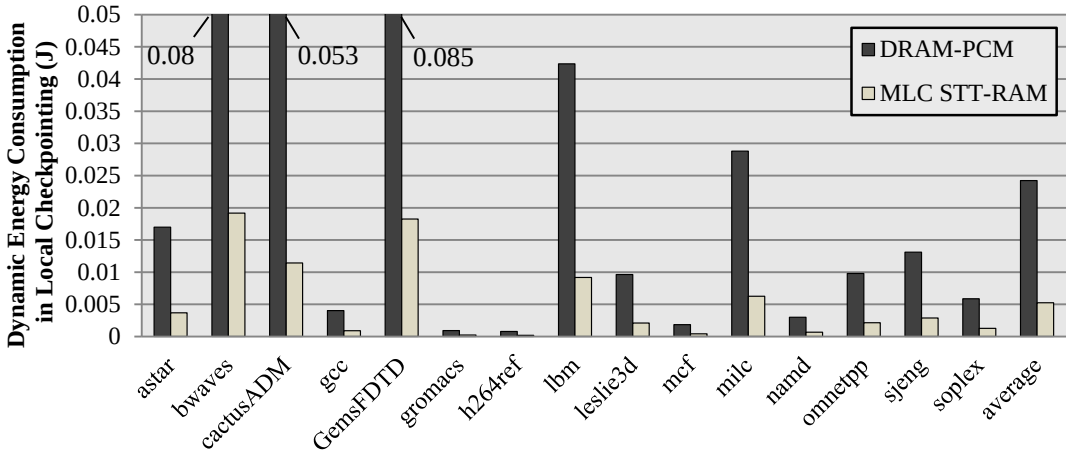


Figure 4.11: Memory energy consumption of creating a local checkpoint in a single-application, single-core process node with $\tau = 5s$.

Recovery Overhead Analysis

The recovery time from a failure includes three phases: the time of diagnosis in which human interactions may be required, the time to restore the system to a checkpoint (rollback), and the time to retrieve lost work. The first phase is out of the scope of this work. The rollback time directly depends on the amount of checkpoint data and the local or global data transfer bandwidth. The third part is determined by the local checkpoint frequency and the local/global checkpoint ratio. With the efficient checkpointing mecha-

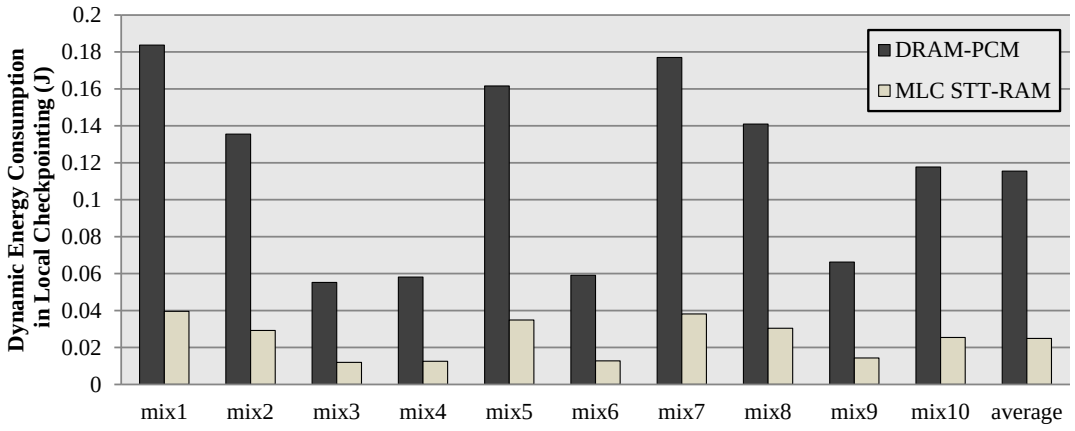


Figure 4.12: Memory energy consumption of creating a local checkpoint in a multi-programmed 4-core process node with $\tau = 5s$.

nism using MLC STT-RAM as main memory, the rollback time to a local checkpoint can be reduced. Furthermore, as indicated by the experiment results, with the MLC STT-RAM mechanism, local checkpointing encounters negligible overhead even when the local checkpoint interval (τ) is as short as $1s$. Consequently, the average overhead of the third phase can be significantly reduced.

4.6 Summary

As the number of nodes increases, modern large-scale computing systems are being challenged by high failure rates. Therefore, smart error recovery mechanisms are urgently needed to help these systems revive from unexpected errors. In this work, emerging STT-RAM is embraced into local/global hybrid checkpointing for efficient in-memory local checkpointing. It leverages MLC STT-RAM as persistent main memory, using the soft-bit of each cell to store the working data while the hard-bit to save the checkpoint data. By taking advantage of the unique features of MLC STT-RAM, the proposed mechanism turns out to be a lightweight and energy-efficient local checkpointing solution. The experimental results show that, the average performance overhead in a multi-programmed

4-core process node is less than 1% even when the checkpoint interval is $1s$, and the memory energy consumption is less than a quarter of the energy required by the existing DRAM-PCM mechanism.

Chapter 5

Making B$^+$tree Efficient for Emerging NVM Based Main Memory

This chapter presents a project that rethinks B$^+$-tree algorithm design for emerging NVMs whose write accesses are much more expensive than read accesses, including PCM, STT-RAM, and ReRAM [53, 54]. First, it introduces the motivation of this project, and then the related work about NVM based system design. Next, it presents a basic cost model for NVM-based memory systems, and detailed CPU cost and memory access models for search, insert and delete operations on a B$^+$-tree. Moreover, based on the models built, the existing NVM-friendly schemes for B$^+$-tree is analyzed. Furthermore, the new schemes are proposed that effectively adapt B$^+$-tree to exploit the full potential of the emerging NVMs without suffering from the issues arising in the existing schemes. Then, the experimental study to compare the proposed B$^+$-tree variants with the state-of-the-art is conducted, before the conclusion of this project.

5.1 Motivation

As we know, PCM, STT-RAM, and ReRAM have all been considered as promising replacements of DRAM for building future main memory systems [3, 7, 10]. Compared with the modern DRAM technology, they consume near-zero idle power and have better scalability. Owing to their non-volatility, they are free of refresh penalty, which has recently been found a big issue of DRAM as its density advances [1]. Additionally, they can support the durability property that traditional main memory databases (MMDBs) built upon volatile memory do not facilitate. Moreover, PCM and ReRAM offer a higher density than DRAM; the multi-level cell (MLC), multi-layer structure, and 3D stacking technologies can further enhance their density [10]. Therefore, under the same area constraint, they can provide a larger memory capacity, and can store most or all of the data in the main memory for many database applications. A previous study has shown that a 4× increase in the memory capacity reduces the number of page faults by 5× on average [3]. Although STT-RAM does not have a density advantage, it provides better read and write performance and energy as well as higher endurance than PCM and ReRAM [7]. With the continuous advance in these technologies, they are anticipated for use in building energy-efficient non-volatile main memory systems of the near future.

Although the nice features of PCM, STT-RAM and ReRAM bring great benefits to MMDBs, new challenges arise due to some of their characteristics. Different from DRAM, they have asymmetric read/write properties. Their read latencies are comparable to that of DRAM, but their write latencies are much slower than their corresponding read latencies [8]. In addition, their write operations consume much more energy than their read operations [8]. They also suffer from endurance issues, and thus it needs to specifically consider their wear-out problems. These unique characteristics change the assumptions that have served as the basis in the designs of conventional database algorithms, making

them suboptimal for the emerging NVM-based main memory systems. Aiming to tackle this issue, the design of database algorithms is reexamined. As writes are much more expensive than reads in these emerging NVMs, the new algorithm design goal is to reduce memory writes, even at the cost of increasing reads.

This project is focused on B$^+$-tree, the widely used index structure in both MMDBs and disk-resident databases (DRDBs), which designs an NVM-friendly variant of B$^+$-tree. Previously, Chen *et al.* proposed unsorted node schemes in their redesign of B$^+$-tree algorithm for PCM [55]. They showed that, using unsorted nodes instead of sorted nodes in the tree can effectively reduce the write accesses required for keeping the nodes in a sorted order. However, according to the analytical results from the proposed cost model for the operations of B$^+$-tree on NVM-based memory systems, it is found that the unsorted node schemes suffer from three problems: 1) in insert operations the node splits are CPU-costly, because sorting the keys in an unsorted node before a split requires intensive computations; 2) the write accesses in insert operations cannot be reduced effectively when branching factors and node sizes are both small, because in this case tree reorganization incurs a lot of writes; 3) the delete operations may waste space significantly because a node is not deleted until it becomes empty. To address the aforementioned problems, three schemes are proposed to adapt B$^+$-tree for the emerging NVMs: 1) the sub-balanced unsorted node scheme which alleviates the computational overhead of sorting before a split is incurred in insert operations, 2) the overflow node scheme which can efficiently reduce the write accesses in insert operations when the existing schemes do not work, and 3) the merging factor scheme which provides more effective trade-offs among execution time, memory energy consumption, memory space usage, and NVM endurance in delete operations.

5.2 Related Work

With their unique characteristics, the read/write costs of emerging NVMs are no longer aligned with the assumptions of the underlying memory systems that have motivated various system designs, including file system design, operating system (OS) design, as well as database system design. Condit *et al.* proposed a new file system based on the properties of persistent, byte-addressable NVMs [56]. Bailey *et al.* examined the implications of fast and cheap NVMs on OS functions and mechanisms [57].

Chen *et al.* proposed to rethink database algorithms for PCM, which inspires this work [55]. They present analytical metrics for PCM endurance, energy and latency, and use them to improve two core databases techniques, B$^+$-tree index and hash joins, for PCM. Their goal for designing PCM-friendly algorithms is to reduce the number of writes while maintaining good cache performance. To improve B$^+$-tree index, they used unsorted nodes instead of sorted nodes in the tree, saving the writes incurred by sorting a node. The unsorted node schemes are simple and effective. However, they suffer from some issues that the proposed schemes avoid.

Hu proposed a predictive B$^+$-tree, called B^p-tree, for PCM-based database systems [58]. She uses a small DRAM buffer to maintain a small B$^+$-tree for current insertions, and predicts future data distribution based on the summary of previously inserted keys in a histogram. Space is pre-allocated in the memory for data to be accessed in the near future so as to reduce the data movements caused by node splits and merges. The approaches proposed in this work are different as the hardware overhead and extra design efforts are minimized by avoiding the usage of additional DRAM buffer.

5.3 Cost Model

This section first introduces the basic cost model for NVM-based main memory systems. Then it analyzes both the CPU costs and the memory behaviors of the original B$^+$-tree algorithm, formulating the memory access numbers incurred by each operation. Based on the cost models built, the existing PCM-friendly unsorted node schemes [55] are analyzed to find out how they reduce the write accesses and what problems they have.

5.3.1 The Basic Cost Model

For each operation of a B$^+$-tree in MMDB, either search, insert or delete, the execution time T is comprised of three components, the CPU execution time T_{CPU} including the access time to the on-chip L1 cache, the access time to all the cache levels except L1 T_{Cache}, and the access time to the main memory T_{Mem}:

$$T = T_{CPU} + T_{Cache} + T_{Mem}. \tag{5.1}$$

T_{CPU} depends on the computational complexity of the algorithm used to implement each operation and the performance of the processor. Let I denote the basic instruction number of the algorithm, CPI denote the average cycle per instruction of the processor to execute basic instructions which does not include memory access latencies in loads and stores, and f denote the frequency of the processor. Then,

$$T_{CPU} = I \times CPI/f. \tag{5.2}$$

For a memory system with l levels of cache, let M_i and L_i denote the miss count and

the access latency of the ith level cache respectively for $i = 1, 2, \ldots, l$. Then,

$$T_{Cache} = \sum_{i=1}^{l-1} M_i \times L_{i+1}. \tag{5.3}$$

The miss count of the ith level cache

$$M_i = A_{total} \times \prod_{k=1}^{i} Mr_k, \tag{5.4}$$

in which A_{total} is the total number of memory accesses, and Mr_k is the average miss rate of the kth level cache. A_{total} simply depends on the algorithm to implement each operation. However, Mr_k is determined by a couple of factors, including both the memory access patterns of the algorithms and the characteristics of the cache hierarchy (*e.g.* capacity, associativity, replacement policies and other cache policies of each level).

Till now, the read memory accesses have not been distinguished from write memory accesses. SRAM and DRAM have similar read and write latency. However, for an emerging NVM, its write latency is much longer than its read latency. T_{Mem} is divided into two parts, the latency of reading cache lines from the main memory to the last level cache and the latency of writing cache lines back to the main memory. The second part can be partially or even completely hidden since writing cache lines back is not in the critical path. Let R_{total} and W_{total} denote the total read and write access numbers, then $A_{total} = R_{total} + W_{total}$. Let Lr_{NVM} and Lw_{NVM} denote the read and write access latencies of an emerging NVM. Then

$$T_{Mem} = R_{total} \times (\prod_{i=1}^{l} Mr_i) \times Lr_{NVM} + \alpha \times W_{total} \times (\prod_{i=1}^{l} Mr_i) \times Lw_{NVM}, \tag{5.5}$$

in which α describes the average impact of writing cache lines back on T_{Mem}, $0 \leq \alpha \leq 1$. Different from the conventional DRAM-based main memory, for an emerging NVM

in which Lw_{NVM} can be several times larger than Lr_{NVM} (Table 2.1), writing may significantly stall the front-end cache line fetches. Therefore, to reduce W_{total} might be beneficial to performance improvement.

In fact, memory access situations are much more complicated than equation (5.3) and (5.5) show. For example, a read request and a write request to the same bank will block each other, that is, the later request has to wait until the previous one gets completed. Several memory access scheduling schemes have been proposed to improve performance for DRAM and NVMs [59, 60].

Because emerging NVMs have high write energy and suffer from the endurance problem, the energy consumption of the NVM-based main memory, E_{Mem}, and the total wear of NVM, $Wear_{total}$, are another two concerns in algorithm design, besides the total execution time T in equation (5.1). To estimate E_{Mem}, there are three parts, the read dynamic energy, the write dynamic energy, and the background energy. Let Er_{NVM} and Ew_{NVM} denote the average energy consumption of a read access and a write access to NVM, and $E_{background}$ denote the background energy. Then,

$$E_{Mem} = R_{total} \times (\prod_{i=1}^{l} Mr_i) \times Er_{NVM} + W_{total} \times (\prod_{i=1}^{l} Mr_i) \times Ew_{NVM} + E_{background}. \quad (5.6)$$

The background energy in NVM is typically much smaller than the read and write dynamic energy. From Table 2.1, Ew_{NVM} can be several times larger than Er_{NVM}. Therefore, reducing W_{total} is helpful to save energy.

Each NVM cell has a limited lifetime. Let γ represent the average number of modified bits per modified cache line, then

$$Wear_{total} = \gamma \times W_{total} \times (\prod_{i=1}^{l} Mr_i). \quad (5.7)$$

Obviously, reducing W_{total} extends the lifetime of NVM.

5.3.2 B$^+$-Tree Parameters

B$^+$-tree is widely used for indexing in file systems and database management systems. There are several important parameters in B$^+$-tree algorithm design. It can have different key types for different applications, *e.g. integer* or *string* of a fixed length. Let *key_size* denote the length of a key in unit of bytes.

Node size (*node_size*) is an important parameter that affects the performance of B$^+$-tree. Previous work has suggested that the node size that results in the best performance should be a few times of the cache line size (*cacheline_size*) [61, 62], *e.g.* 2 cache lines or 4 cache lines. In modern computers, *cacheline_size* is usually 32, 64 or 128 bytes. The best node size also depends on the branching factor of the tree.

The branching factor (or the order) of a B$^+$-tree is the maximum number of children that each internal node can have, denoted by b. Then there are at most $b-1$ keys and b pointers in an internal node. The size of a pointer *pointer_size* is usually 4 bytes in a 32-bit machine and 8 bytes in a 64-bit machine. For leaves, it is assumed that a record is a tuple of <key, pointer>, in which the pointer points to the data value. In this case, the leaves have the same node structure with the internal nodes. Therefore, there are at most $b-1$ records in a leaf.

Each node also maintains some necessary information in their node structure, *e.g.* the number of keys and a flag indicating a leaf or non-leaf node. Let $nodeinfo\_size$ denote the space it takes a node to store the information. Then,

$$b = \lfloor (node\_size - nodeinfo\_size + key\_size)/(key\_size + pointer\_size) \rfloor. \quad (5.8)$$

The height of a B$^+$-tree, denoted by h, indicates how many nodes in a search path,

which means how many nodes that have to be visited from the root to a leaf. Therefore, it is quite important to the search performance of a B$^+$-tree. Given the total number of records that a B$^+$-tree holds, say N, the height of the tree can be estimated with the branching factor of the tree and another parameter — the full factor of the tree.

Definition 1 *The full factor of a B$^+$-tree is the ratio of the average number of children that each internal node has to the branching factor, denoted by f, $0.5 \leq f \leq 1$.*

For a randomly inserted B$^+$-tree, the experimental results show that 0.75 is a good estimate of its full factor. It is reasonable to assume that the leaves are similarly full to the internal nodes. Then the height of a B$^+$-tree can be estimated as:

$$h = \lceil \log_{f \cdot b} N \rceil. \tag{5.9}$$

5.3.3 Search

To search a record (key), a path whose length is the height of the tree is followed, from the very root to the very leaf. Therefore, h nodes are accessed along the path. For searching a key within one node, two algorithms are considered: linear search and binary search.

To search the position of a key in an array of size n with linear search, the maximum number of iterations is n, and the average performance is $n/2$. However, with binary search, the maximum number of iterations is $\lfloor \log_2 n \rfloor + 1$, and the average performance is $\log_2 n$. Let α_L and α_B denote the CPU time of each iteration in linear search and in binary search, then T_{CPU} for the two search algorithms can be estimated as shown in Table 5.1 (in which "L" denotes *linear* and "B" denotes *binary*).

Ideally, there are no write involved in search operations. The read access numbers (in unit of cache lines) of the two algorithms are analyzed as follows. The average usage

Table 5.1: Analytical Cost Model for A Search Operation

	T_{CPU_Search}	R_{total_Search}
L	$\alpha_L \times (f \cdot b/2) \times h$	$(1 + \lceil \frac{f \cdot node\_size}{cacheline\_size} \rceil)/2 \times h$
B	$\alpha_B \times \log_2 (f \cdot b) \times h$	$(1 + \lfloor \log_2 \frac{f \cdot node\_size}{cacheline\_size} \rfloor) \times h$

of each node in unit of cache lines is:

$$k = \lceil f \cdot \frac{node\_size}{cacheline\_size} \rceil. \tag{5.10}$$

For each linear search operation, it is assumed that the probability to access 1 — k cache lines is the same, and then the average read access number is

$$R_{linear} = \sum_{i=1}^{k} \frac{1}{k} \cdot i = (1 + k)/2 = (1 + \lceil \frac{f \cdot node\_size}{cacheline\_size} \rceil)/2. \tag{5.11}$$

For each binary search operation, the average read access number is estimated as

$$R_{binary} = \lfloor \log_2 k \rfloor + 1 = \lfloor \log_2 \frac{f \cdot node\_size}{cacheline\_size} \rfloor + 1. \tag{5.12}$$

The "floor rounding plus 1" is used in equation (5.12), because it needs to consider the case of $k = 1$ and the fact that the first cache line must be accessed as some necessary information (*e.g.* the number of keys in this node) is stored there.

Table 5.1 also summarizes R_{total} for the two search algorithms. Usually, α_B is larger than α_L, *e.g.*, $\alpha_B = 2\alpha_L$. From Table 5.1, it can be found that when b is small or *node_size* is small, linear search might have better performance than binary search.

5.3.4 Insert

To insert a record to a B$^+$-tree, a search is performed first to determine in which leaf and to which position the record should be inserted. Within the target leaf, inserting the record would involve a lot of reads and writes since on average half the number of records in the node need to be shifted; and if the leaf is full, it needs to be split into two nodes, and a key needs to be inserted to the parent node; and a split can be propagated upwards even to the root.

For an insertion to a non-full node, called "a common insertion", either a leaf or non-leaf node, the average write access number involved is

$$W_{comins} = \begin{cases} 1, & \text{if } node\_size = cacheline\_size \\ 1 + \lceil \frac{f \cdot node\_size/2}{cacheline\_size} \rceil, & otherwise. \end{cases} \tag{5.13}$$

When $node\_size > cacheline\_size$, "1" in equation (5.13) comes from updating the node length in the first cache line of the node.

For a split in a full node, either a leaf or non-leaf node, the write access number involved is

$$\begin{aligned} W_{split} &= \frac{1}{2}(\lceil \frac{node\_size/2}{cacheline\_size} \rceil + \lceil \frac{node\_size/4}{cacheline\_size} \rceil) + \frac{1}{2}(\lceil \frac{node\_size/2}{cacheline\_size} \rceil + 1) \\ &= \lceil \frac{node\_size/2}{cacheline\_size} \rceil + \frac{1}{2}(\lceil \frac{node\_size/4}{cacheline\_size} \rceil + 1). \end{aligned} \tag{5.14}$$

Each insert operation (inserting a record to a B$^+$-tree) incurs exactly one common insertion, and zero or a few splits. The average number of splits that an insert operation

incurs can be calculated by

$$P_{split} = \frac{1}{f \cdot b - 1} + \frac{1}{(f \cdot b - 1)^2} + \cdots + \frac{1}{(f \cdot b - 1)^h} \approx \frac{1}{f \cdot b - 2}. \tag{5.15}$$

Then the average W_{total} for an insert operation is

$$W_{total\_Insert} = W_{comins} + P_{split} \times W_{split}. \tag{5.16}$$

It is found that all the data that are written are needed to be read first. Therefore, the estimate of R_{total} for an insert operation is

$$R_{total\_Insert} = R_{total\_Search} + W_{total\_Insert}. \tag{5.17}$$

The dominant component of the CPU time for an insert operation is the search part, so T_{CPU} can by simply estimated as

$$T_{CPU\_Insert} = T_{CPU\_Search}. \tag{5.18}$$

5.3.5 Delete

The cost analysis of a delete operation is a bit more complicated than that of insert. Deleting a record from a B$^+$-tree also executes a search first to determine in which leaf and from which position the record should be deleted. Deleting a key from a node may incur a borrow or a merge, and a merge can be propagated upwards even to the root.

For "a common deletion", which does not incur a borrow or merge, from either a leaf or non-leaf node, half the number of keys in the node on average need to be shifted, so

the average write access number involved is

$$W_{comdel} = \begin{cases} 1, & \text{if } node\_size = cacheline\_size \\ 1 + \lceil \frac{f \cdot node\_size/2}{cacheline\_size} \rceil, & otherwise. \end{cases} \tag{5.19}$$

When deleting a key from a half-full node, if one of its sibling nodes is more than half full, it will need to borrow a key from it; otherwise, it will merge the node with one sibling node. For borrowing a key from a right sibling node, it needs to modify at most two cache lines in the node itself, one for the insertion of the borrowed key to the end, and the other for updating the node length if they are not in the same cache line. In the right sibling node, it needs to shift all the keys to the left by one position since the first key is lent. It also needs to update the parent node by changing one key. It is assumed that the right sibling node is similar to half full, then the write access number incurred by borrowing a key from a right sibling node is

$$W_{borrow\_r} = \begin{cases} 3, & \text{if } node\_size = cacheline\_size \\ 3 + \lceil \frac{node\_size/2}{cacheline\_size} \rceil, & otherwise. \end{cases} \tag{5.20}$$

For borrowing a key from a left sibling node, the write access number incurred is the same with borrowing a key from a right sibling node:

$$W_{borrow\_l} = W_{borrow\_r} = W_{borrow}. \tag{5.21}$$

Therefore, W_{borrow} is used to denote the write access number incurred by borrowing from either a right or a left sibling node.

For a merge of two half-full nodes, either leaves or non-leaf nodes, the write access

number involved is

$$W_{merge} = \begin{cases} 2, & \text{if } node\_size = cacheline\_size \\ 2 + \lceil \frac{node\_size/2}{cacheline\_size} \rceil, & otherwise. \end{cases} \quad (5.22)$$

A merge will result in a common deletion or a borrow in the parent node, or propagate a merge upwards.

Each delete operation (deleting a record from a B$^+$-tree) incurs 0 or a few merges, and one common deletion or one borrow. Let P_{borrow} denote the average number of borrows that a delete operation incurs and let P_{merge} denote the average number of merges that a delete operation incurs. Then the average W_{total} for a delete operation is

$$W_{total\_Delete} = (1 - P_{borrow}) \times W_{comdel} + P_{borrow} \times W_{borrow} + P_{merge} \times W_{merge}. \quad (5.23)$$

All the data that are written are needed to be read first. Therefore, the estimate of R_{total} for a delete operation is

$$R_{total\_Delete} = R_{total\_Search} + W_{total\_Delete}. \quad (5.24)$$

The dominant component of the CPU time for a delete operation is still the search part, so T_{CPU} can be simply estimated as

$$T_{CPU\_Delete} = T_{CPU\_Search}. \quad (5.25)$$

5.3.6 Analysis of Existing Unsorted Node Schemes

In the insert and delete operations, keeping the keys of the involved node in order incurs a lot of write accesses. Let's look at the total write number in an insert operation.

The first term, W_{comins} in equation (5.16), comes from making space for the inserted key (and pointer) by shifting those keys behind the inserted position. For the total write number in a delete operation, the first term, W_{comdel} in equation (5.23), comes from the similar way, filling the space of the deleted key by shifting those keys behind the deleted position. Therefore, the write costs of W_{comins} and W_{comdel} come from keeping all the keys in the node in order. To reduce these terms, one efficient solution is to leave the node unsorted.

Chen et al. proposed three PCM-friendly variants of B$^+$-tree with unsorted node schemes: 1) **unsorted** with all the non-leaf and leaf nodes unsorted, 2) **unsorted leaf** with sorted non-leaf nodes but unsorted leaf nodes, and 3) **unsorted leaf with bitmap** in which each unsorted leaf node uses a bitmap to record valid locations [55]. In the unsorted scheme, since all the nodes are unsorted, a search incurs a lot of computation overhead because it has to use linear search in every node of the search path. In the unsorted leaf scheme, the better one can be chosen from binary search and linear search in the sorted non-leaf nodes and linear search is used in the only one unsorted target leaf. Therefore it can achieve similar search performance to the original B$^+$-tree whose nodes are all sorted. Moreover, since most of the write accesses occur in the leaf nodes, the unsorted leaf scheme also captures the benefits to reduce the write accesses in the unsorted leaf nodes. Let's look at how this unsorted leaf scheme modifies the cost model.

For a search operation, as mentioned above, either linear search or binary search algorithm can be used in all the $h-1$ sorted non-leaf nodes along the search path of length h; at the end of the path, in the unsorted target leaf node, it has to search keys one by one. The terms in Table 5.1 can be easily combined to get the cost model for the search operation of the unsorted leaf scheme.

A "common insertion" to a leaf does not need to shift half the keys on average in the node; it only attaches the inserted key at the end as the last key. Therefore, at most two

cache lines are needed to be written, one for the inserted key, and the other for updating the node length if they are not in the same cache line. Common insertions to non-leaf nodes have the same cost model with equation (5.13), and only splits incur insertions into non-leaf nodes. So the probability of a common insertion to a non-leaf node can be estimated by P_{split} in equation (5.15). The W_{comins} for the unsorted leaf scheme is

$$W_{comins\_usl} = \begin{cases} 1, & \text{if leaf and } node\_size = cacheline\_size \\ 2, & \text{if leaf and } node\_size > cacheline\_size \\ W_{comins} \textit{ in eq.}(5.13), & \text{if nonleaf.} \end{cases} \tag{5.26}$$

Although the unsorted leaf scheme reduces the cost of "common insertions", it makes splits more costly, not only on CPU execution time but also on memory accesses. To split a full unsorted leaf, it needs to sort the keys first, and then split the sorted leaf to two nodes as the original sorted B$^+$-tree does. In-place *Quicksort* is used as the sorting algorithm, whose average time complexity is $O(n \log_2 n)$ (actually when the number of items to be sorted is small, say less than 6, a simple comparison sort is used). Then the CPU overhead due to a split of the unsorted leaf is

$$T_{split\_leaf\_usl} = \alpha_s \times (b-1)\log_2(b-1), \tag{5.27}$$

where α_s is the CPU time unit of the sorting algorithm.

The write number incurred by splitting an unsorted leaf can be estimated by

$$W_{split\_leaf\_usl} = \frac{node\_size}{cacheline\_size} + \lceil \frac{node\_size/2}{cacheline\_size} \rceil \tag{5.28}$$

in which the first term denotes the leaf node itself and the second term denotes the new allocated node. Additional space cost by the sorting algorithm is ignored.

P_{split_leaf} describes the weight how $T_{split\_leaf\_usl}$ and $W_{split\_leaf\_usl}$ affect the total CPU time and the total write access number. From equation (5.15),

$$P_{split\_leaf} = \frac{1}{f \cdot b - 1}. \tag{5.29}$$

It can be found that when the branching factor b is small, P_{split} and P_{split_leaf} might be large enough so that the split costs have considerable impacts on the overall CPU time and write access number.

It is noticed that $P_{split\_leaf} \times T_{split\_leaf\_usl}$ is $\sim \alpha_s \times \log_2(b-1)$, which indicates that for each insert operation, the CPU overhead is not small to sort a full unsorted node before its split. The sub-balanced unsorted node scheme is proposed in the next section to reduce such costs in existing unsorted node schemes.

For small bs and small *node_size*s, the unsorted leaf scheme might be inefficient to reduce the write number because the reduction on W_{comins} is not significant and splits might incur more write accesses than the original B$^+$-tree algorithm. Figure 5.1 shows the relative write access number of the unsorted leaf scheme compared to the original B$^+$-tree algorithm, according to different branching factors and different node sizes, in which $write\ number > 1$ means the total write access number of the unsorted leaf scheme is even larger than that of the original B$^+$-tree algorithm. Therefore, from Figure 5.1, it can be found that when the branching factor b and the node size *node_size* are small, *e.g.* $b \leq 10$ and $node\_size \leq 4$, the unsorted leaf scheme cannot efficiently reduce the total write number. The overflow node scheme is proposed in the following section to cope with such cases with small bs and *node_size*s.

For the delete operation in the unsorted leaf scheme, a "common deletion" from a leaf does not need to shift half the keys on average in the node either; it only puts the last key at the deleted position and decreases the node length. Therefore, at most two

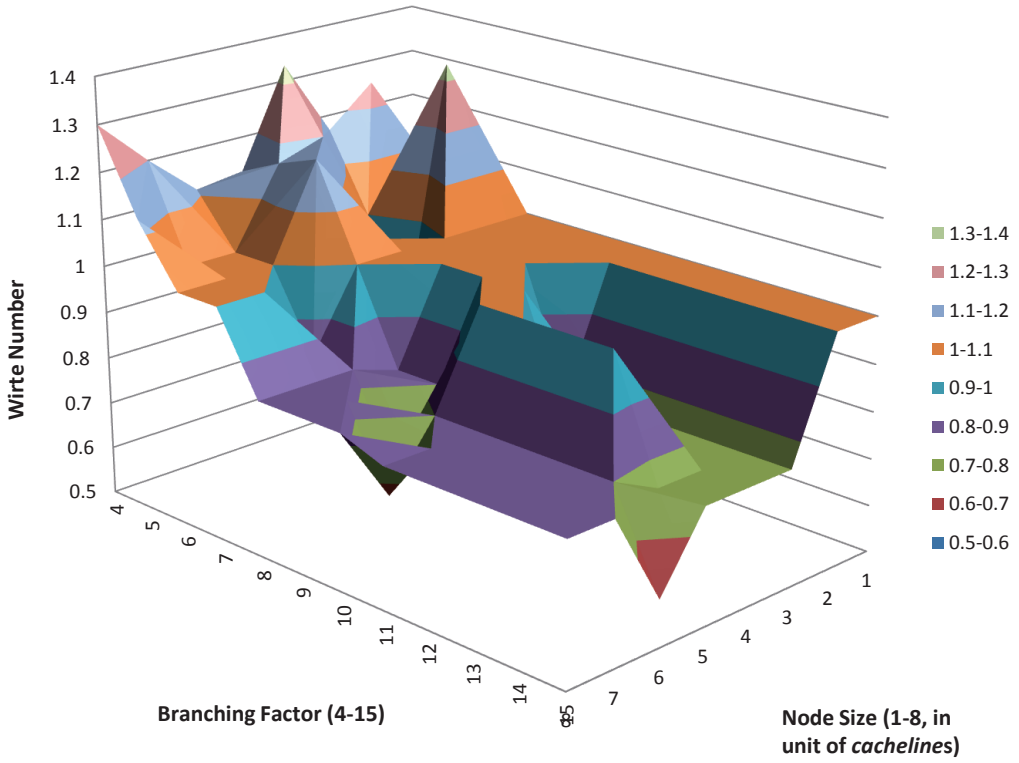


Figure 5.1: The relative write access number of the unsorted leaf scheme compared to the original B$^+$-tree algorithm according to different branching factors and different node sizes .

cache lines are modified in a "common deletion" from a leaf. Therefore, it can reduce the write accesses incurred by keeping the node in order. To reduce the write accesses incurred by the other source, tree reorganization, in unsorted node schemes, a node, either a leaf or non-leaf, will not be deleted unless there is no key in it. However, such delete algorithm may waste a lot of space when there are a lot of delete operations, and even degrade the search performance greatly. In the original B$^+$-tree algorithm, to delete a key from a half-full node, will either borrow a key from or merge with one of its sibling nodes, which incurs a lot of write accesses. The merging factor scheme is proposed in the following section which makes a compromise with neither too much space nor too many write accesses.

5.4 Algorithms

The goal of this project is to make B$^+$-tree NVM-friendly. Due to NVM's asymmetric characteristics between read and write and its limited lifetime, it aims to reduce the number of write accesses that are involved in insert and delete operations as well as to keep the new indexing method efficient for search operations. According to the issues found by the analysis in Section 5.3.6 that the existing unsorted node schemes suffer from, three schemes are introduced in this section, *i.e.*, the sub-balanced unsorted node scheme, the overflow node scheme, and the merging factor scheme. They can improve the performance and reduce the write accesses in the cases where the unsorted node schemes are inefficient.

5.4.1 Sub-balanced Unsorted Node Scheme

The existing unsorted node schemes sort the keys in a full unsorted node before split it, and then the two consequent nodes have to be at least half full to keep the tree balanced. As shown above, the sorting overhead is considerable. In the *sub-balanced unsorted node scheme*, the nodes can be less than half full, *i.e.*, unbalanced, after the split from a full unsorted node. With this scheme, during a split of a full unsorted node, it only needs to choose a pivot to assign the keys into two nodes, of which one holds the keys greater than the pivot, and the other holds the rest keys. Since it does not need to sort all the keys in the full unsorted node before the split, the intensive computation of sorting is saved.

To choose a good pivot is very important to keep the tree balanced and hence efficient. However, to choose the perfect pivot, which results in two half-full nodes, is as complex as to sort the keys, *i.e.*, $O(n \log_2 n)$. The middle key value, $key_{middle} = (key_{max} + key_{min})/2$, can be used as the pivot. It requires to find the maximum key (key_{max}) and the minimum

key (key_{min}) in the node first. Experiment results show that key_{middle} is usually a very good choice. The time complexity of this scheme for a split is $O(n)$.

The sub-balanced unsorted node scheme also saves the additional space required by the sorting algorithm and reduces some memory accesses. In sub-balanced unsorted leaf scheme, the unbalanced leaf nodes have little impact on the search performance.

5.4.2 Overflow Node Scheme

When the branching factor b and the node size $node\_size$ are both small ($b \leq 10$ and $node\_size \leq 4$ as shown in Figure 5.1), the unsorted leaf scheme cannot reduce the total write access number at all. The reason is that, when b and $node\_size$ are small, an insertion to or a deletion from a node will probably incur a write access to the whole node, no matter if the node is sorted or unsorted. Moreover, when b and $node\_size$ are small, a great portion of write accesses come from tree reorganizations, and the unsorted node schemes are not able to reduce such write accesses. The *overflow node scheme* is designed to reduce the write accesses incurred by tree reorganization operations, *i.e.*, splits and merges.

In the original B$^+$-tree, a split of a leaf node would involve write accesses to at least three nodes: *i*) the original leaf node which splits, *ii*) the new leaf node to split to, and *iii*) the parent node. The overflow node scheme postpones the update to the parent node by splitting the leaf node to an overflow node, and later execute several updates in batches in the parent node after many more splits occur in its child nodes. Because each update involves a certain number of cache line writes, to execute several updates together will save a lot of write accesses. Similarly, for delete operations, since a leaf node can merge with an overflow node which does not require an update to the parent node, the write accesses are also reduced. The overflow node scheme is different from L.

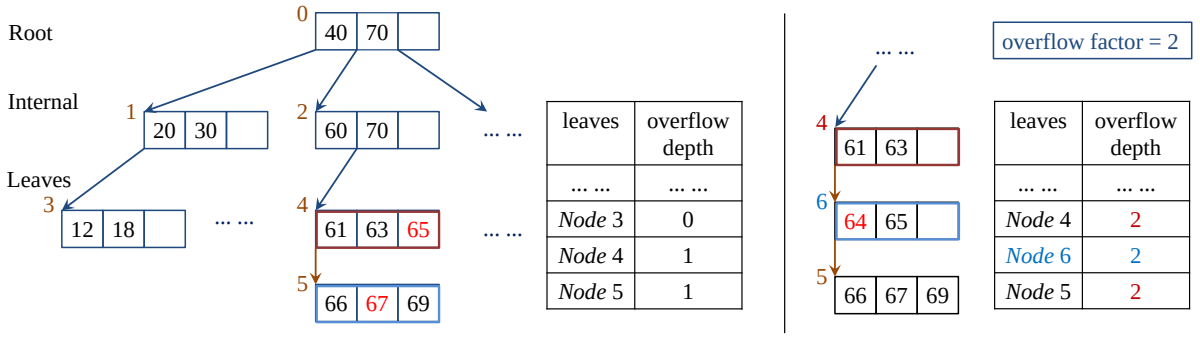
Arge's buffer tree technique which allocates buffers to internal nodes in order to support batched insertions to the leaves for I/O efficiency [63, 64].

In the overflow node scheme, a leaf node can have one or more overflow nodes, and all the nodes are sorted. The overflow nodes have the same structure with ordinary leaf nodes. Here are two important definitions in this scheme.

Definition 2 *The overflow depth of a leaf node is the number of overflow nodes it attached; the overflow depth of the overflow nodes are the same with the first leaf node.*

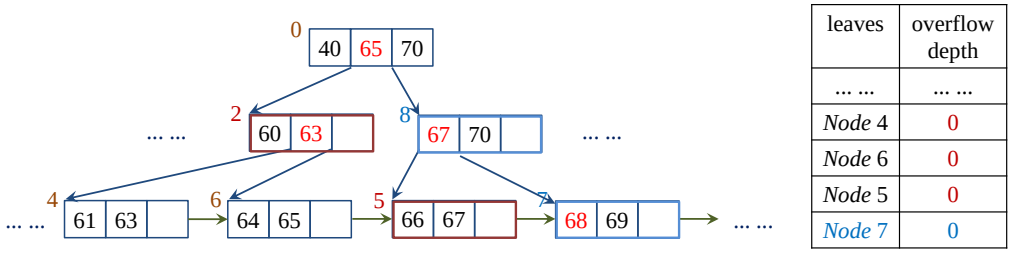
For example, if a leaf node has no overflow node, its overflow depth is 0; if a leaf node has one overflow node, the overflow depths of both the leaf node and the overflow node are 1. Only when the overflow depth of a leaf node reaches the overflow factor of the tree, the next split will cause a reorganization of the tree, in which all the overflow nodes of the leaf node become independent leaf nodes with an overflow depth of 0 and a set of keys are inserted into the parent node all at once.

In the implementation, the overflow factor of the tree is maintained as a global variable, and the structure of leaf nodes are modified to keep the overflow information for each leaf node. It adds the overflow depth and an overflow pointer that points to the following overflow node to the original leaf node structure. The overflow depth costs 1 byte since 255 is large enough for a possible overflow factor. Because all the overflow nodes of a leaf node has the same overflow depth, only the overflow depth in the 0^{th} overflow node is updated each time it changes in order to reduce the number of writes. The overflow pointer is a normal pointer which costs 4 bytes for each leaf node in a 32-bit machine. For $nodesize = 2$ cache lines, the space overhead to keep the overflow information in a leaf node is $(1+4)/(64 \times 2) \approx 4\%$, assuming the cache line size is 64 bytes. For $nodesize = 4$ cache lines, the overhead is about 2%. This overhead has little impact on the performance of the algorithm.



(a) a B+-tree with overflow factor =2; search *key* 67

(b) insert *key* 64



(c) insert *key* 68

Figure 5.2: An example of the overflow node scheme (a) a B$^+$-tree with *overflow factor* = *2*; search a record with *key* 67 (b) insert a record with *key* 64 to the tree; then *Node* 4 splits to a new overflow node *Node* 6, and its overflow depth becomes 2 reaching the overflow factor (c) insert a record with *key* 68 to the tree; then *Node* 5 splits to a new node *Node* 7, and *Nodes* $4-7$ become independent leaves, and a set of keys are inserted to their parent node *Node* 2; then *Node* 2 splits to a new node *Node* 8, and a key is inserted to its parent node *Node* 0.

Insert

Figure 5.2 shows an example of the insertion to a B$^+$-tree whose overflow factor is 2. Figure 5.2(a) is the tree before insertion. It has 6 nodes labeled from 0 to 5, and each node is able to store up to three keys. *Node* 4 is a leaf node, and it has an overflow node, *Node* 5. Figure 5.2(b) depicts inserting a record with *key* 64 to the tree. *Node* 4 splits to a new overflow node, *Node* 6, and the overflow depth of *Node* 4 becomes 2, reaching the overflow factor of the tree. Figure 5.2(c) presents the result of the next insertion, inserting *key* 68 to *Node* 5. First, *Node* 5 has to split to a new node *Node* 7. Second,

because the overflow depth of *Node* 5 has already reached the overflow factor, all the four leaf nodes, *i.e.*, *Node* $4-7$, become independent leaf nodes, and their overflow depths are reset to 0. Third, it needs to send all the index information for each independent leaf node to the parent node, so *Node* 2 and a set of three $< key, pointer >$ tuples are inserted at once. Then *Node* 2 is full and has to split to a new node, *Node* 8, and at last a key needs to be inserted to its parent node, *Node* 0. From this example, it can be found that the overflow node scheme has the ability to postpone the insertions to the parent node incurred by splits and deal with several updates in batches.

Search

It is easy to search a key with the overflow node scheme. Like in the original B$^+$-tree, the first leaf node or the 0^{th} overflow node can be located that may contain the key. Then its overflow depth is checked to see if it has overflow nodes. If it has no overflow node, this leaf node is the only target leaf node that may contain the key, so we can execute either binary search or linear search within the node to find out the key. Otherwise, the key is first compared with the last key, also the greatest key, of the leaf node. If the key is no greater than the last key, this leaf node is the target leaf node; otherwise, the overflow pointer is followed to locate its overflow node, and then to check if the overflow node is the target leaf node; and so on. For example, we search *key* 67 in the B$^+$-tree in Figure 5.2(a). We first locate the first leaf node that may contain *key* 67, *i.e.*, *Node* 4. As it is found that *Node* 4 has an overflow node, we compare *key* 67 with the last key in *Node* 4, *i.e.*, *key* 65. *key* 67 is greater, so we directly enter the overflow node, *Node* 5. Since *Node* 5 does not have an overflow node, we execute a binary or linear search in *Node* 5 to find *key* 67. With the overflow node scheme, the path to search a key might be longer, so the overflow node scheme may affect the search performance.

Delete

The delete operation of the overflow node scheme is similar to the original B$^+$-tree algorithm. To delete a key, first it needs to find the key in the target leaf node, and then check if the node is half-full. If not, or the node is the root node, simply delete the key in the node; otherwise borrow a key from or merge with a neighbor node. In the original B$^+$-tree, the neighbor node must be a right or left sibling node. In the overflow node scheme, however, the neighbor node can be either a left or right overflow node, or a left or right sibling node. In the algorithm, an overflow node is preferred to a sibling node, because either to borrow a key from or to merge with a sibling node needs to write the parent node, which incurs an extra write access. Therefore, the overflow node scheme has the ability to postpone the updates in the parent node involved by a borrow or merge, thus reducing the write accesses incurred by a deletion from a half-full leaf node compared with the original B$^+$-tree.

5.4.3 Merging Factor Scheme

In the unsorted node schemes, a node will not be deleted unless there is no key in it. Such a delete algorithm removes most of the writes that may be involved by tree reorganizations due to deletions. However, for applications with a large amount of delete operations, this delete algorithm may waste too much space and even degrade the performance of the tree.

In the original B$^+$-tree, when a key is deleted from a node, it needs to merge with a sibling node if they are both half full. However, it seems too early to merge two nodes when they just become less than half full. On one hand, after a merge, the resulting node is 100% full, not a stable state since an insertion will cause it to split. On the other hand, after a split, the resulting two nodes are both half-full, also not a stable state

since a deletion may cause them to borrow keys or to merge. Therefore, to reduce the write accesses that may be incurred by such unstable states, the merging factor scheme is proposed which looses the merging conditions of the original B$^+$-tree. There are two important definitions in this scheme.

Definition 3 *The filling degree of a node is the ratio of the number of keys in the node to the maximum number of keys that the node can contain.*

Definition 4 *The merging factor of a B^+-tree algorithm is the filling degree when two neighbor nodes need to merge with each other if a key is to be deleted from one of them.*

The filling degree is in the range of $[0-1]$, describing a node's full state, in which "0" means empty and "1" means full. The merging factor should be in range of $[0-0.5]$, and the merging factor of the original B$^+$-tree algorithm is 0.5 and the merging factor of the unsorted node schemes is 0.

In the merging factor scheme, the merging factor can be defined less than 0.5, in order to reduce the write accesses caused by early merges. As the merging factor decreases, the merging condition is loosen, and hence merges occur less frequently so that the involved write accesses are saved. However, if the merging factor is set too small, the leaves become sparse or even near empty and thus a lot of space is wasted.

5.5 Evaluation

In this section, first the experimental methodology is introduced, and then the experiment results are presented.

5.5.1 Experimental Methodology

A pin-based simulator is built to model a 64-bit out-of-order processor and different NVM-based memory systems. Pin is a dynamic binary instrumentation framework from Intel, supporting computer architecture analysis for IA-32 and x86-64 instruction-set architectures [65]. For cache architecture, two levels of cache are modeled, including separate L1I and L1D cache each of size 32KB, and a large L2 cache of size 2MB. The cache line size is 64 bytes. For main memory, the simulator is tailored to support PCM, STT-RAM and ReRAM models. The processor and memory system configurations are summarized in Table 5.2.

Table 5.2: Simulation Setup

Processor	1-core, x86-64, 2GHz, out of order
Cache	L1I, 32KB, 64B line, 4-way, 1 cycle latency L1D, 32KB, 64B line 4-way, 1 cycle latency L2, 2MB, 64B line, 16-way, 10 cycle latency
Memory	4GB NVM, 2 ranks, 8 banks PCM: 50 ns read latency, 500 ns write latency STT-RAM: 15 ns read latency, 20 ns write latency ReRAM: 15 ns read latency, 100 ns write latency

5.5.2 Experimental Results

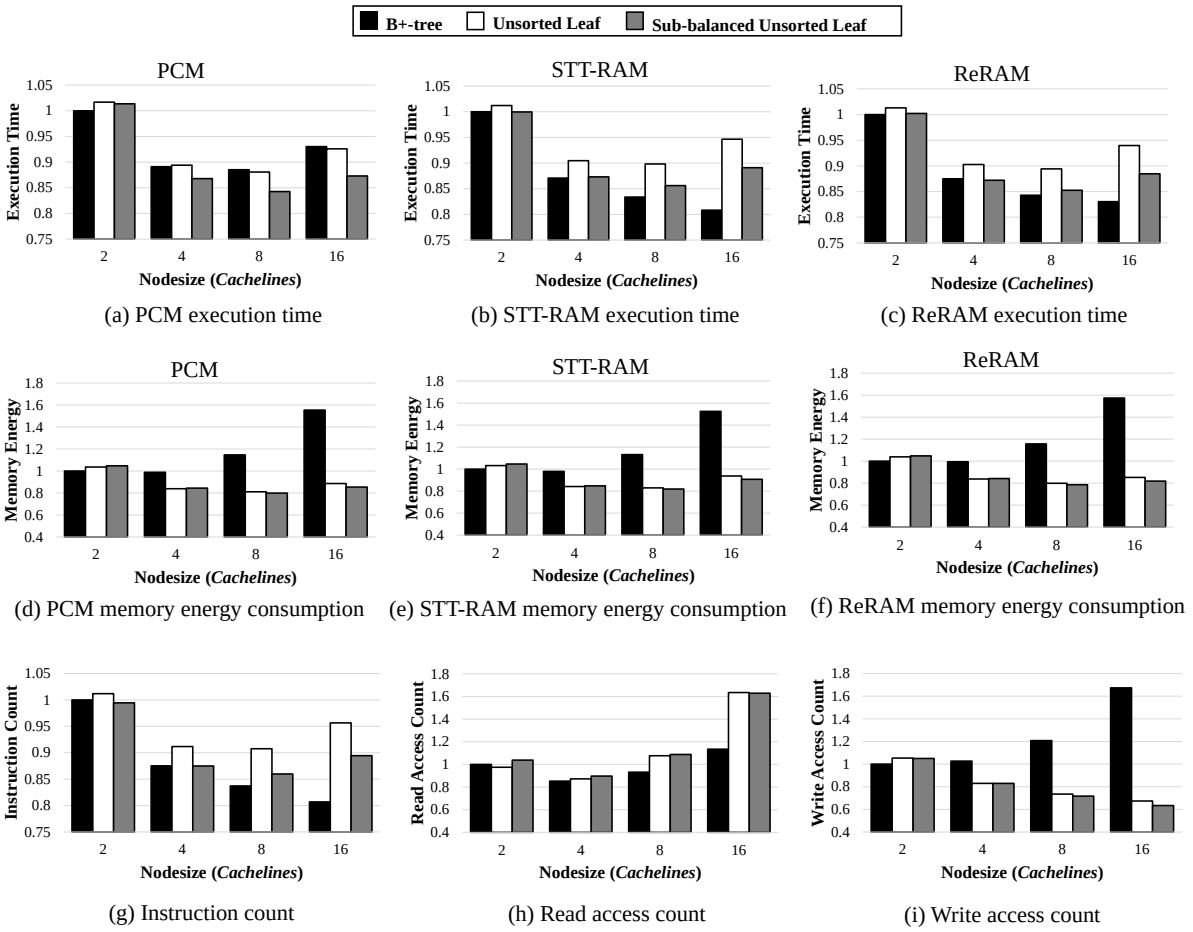
Results for The Unsorted Node Schemes

The unsorted leaf scheme and the sub-balanced unsorted leaf scheme are compared with the original B$^+$-tree for PCM, STT-RAM and ReRAM based main memory systems under the insert-, search- and delete-only workloads. The insert-only workload inserts one million records with random keys to an empty tree; the search-only workload searches every record of a tree with one million records in a random order; and the delete-only workload randomly deletes half the records from a tree holding one million records.

Binary search is used for searching in all the levels of sorted non-leaf nodes, and linear search is used for searching in the unsorted leaves. The key type used is 16-byte *string*. For the insert- and search-only workloads, the node sizes from 2 to 16 cache lines are evaluated. For the delete-only workload, the node size of 4 cache lines is tested. A tree with one million records is large enough to cause a high last level (L2) cache miss rate. The L2 miss rate for the insert- or delete-only workload is $4\% - 10\%$ with the increase of the node sizes; and for the search-only workload, it ranges from 20% to 60%.

Figure 5.3 presents the comparison results for the insert-only workload, and all the results are normalized to the result of the original B$^+$-tree with $node\_size = 2$. Figure 5.3(a), (b), and (c) show the execution time in PCM, STT-RAM, and ReRAM based systems, respectively. Compared with the original B$^+$-tree, the execution time of the unsorted leaf scheme is similar in PCM based system; however, it is increased by $1.2\% - 13.8\%$ in STT-RAM based system, and by $1.3\% - 10.9\%$ in ReRAM based system, among different node sizes. The reason is that, although the unsorted leaf scheme reduces the write access count significantly as the node size increases as shown in Figure 5.3(i), the instruction count as well as the read access count rise greatly as shown in Figure 5.3(g) and (h). Since the unsorted leaf scheme reduces the write access count by $19.8\% - 100\%$ ($19.3\% - 62.1\%$ if not normalized to the result of the original B$^+$-tree with $node\_size = 2$) when $node\_size$ ranges from 4 to 16 $cacheline\_size$, it reduces the memory energy consumption a lot, $15.0\% - 66.8\%$ for the PCM-based main memory, $13.8\% - 58.8\%$ for the STT-RAM-based main memory, and $15.7\% - 72.2\%$ for the ReRAM-based main memory.

Compared with the unsorted leaf scheme, the proposed sub-balanced unsorted leaf scheme reduces the execution time by removing the CPU-intensive sorting before splits in all PCM, STT-RAM, and ReRAM systems. As shown in Figure 5.3(a), the execution time is decreased to the extent even better than that of the original B$^+$-tree, in



(a) PCM execution time (b) STT-RAM execution time (c) ReRAM execution time

(d) PCM memory energy consumption (e) STT-RAM memory energy consumption (f) ReRAM memory energy consumption

(g) Instruction count (h) Read access count (i) Write access count

Figure 5.3: Comparison among B$^+$-tree, the unsorted leaf scheme, and the proposed sub-balanced unsorted leaf scheme in PCM, STT-RAM, and ReRAM based systems for an insert-only workload (insert one million records with random keys to an empty tree). The results are normalized to those of B$^+$-tree with $node\_size = 2$.

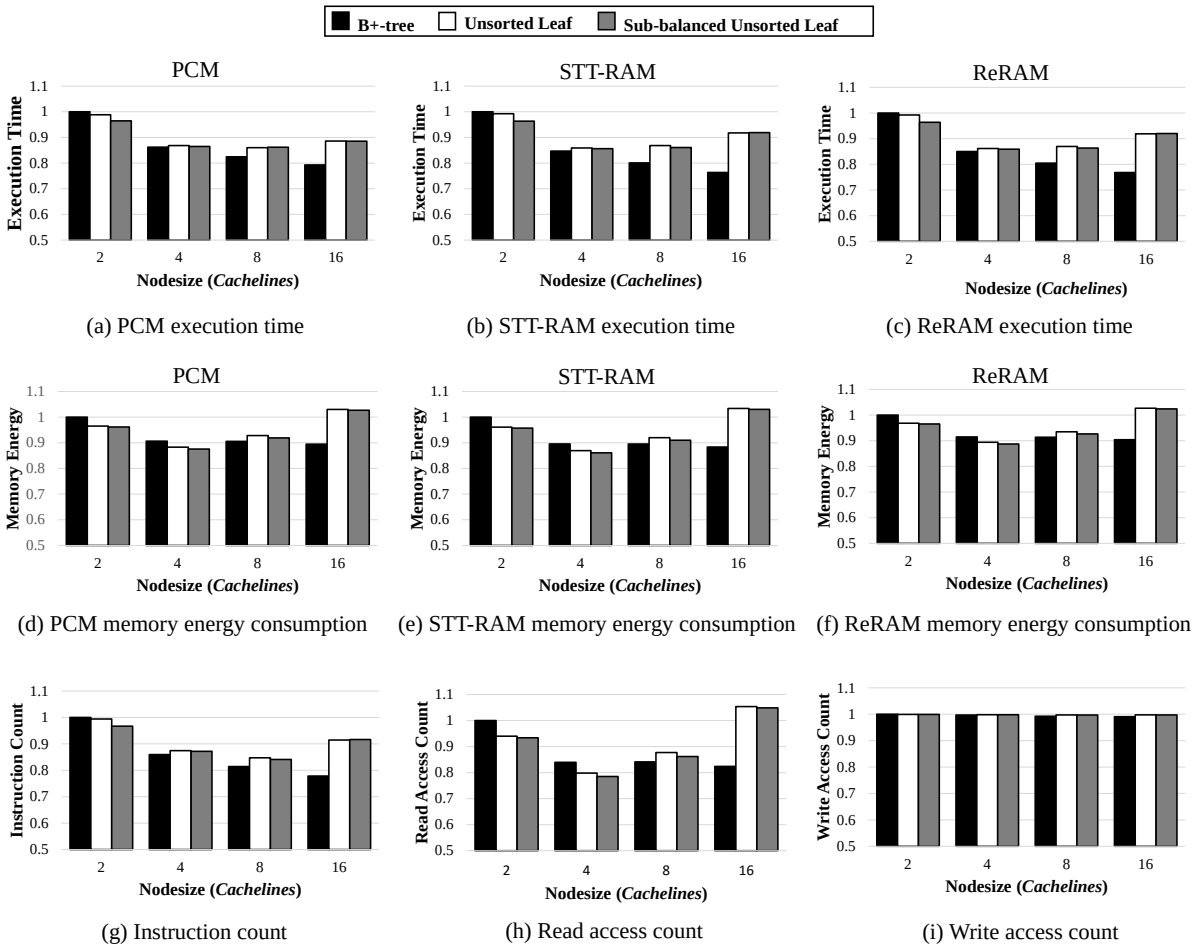
PCM based system. In STT-RAM and ReRAM based systems, the execution time is reduced by $1.1\% - 5.5\%$ among different node sizes, as shown in Figure 5.3(b) and (c). Figure 5.3(g) shows that the instruction count is decreases in sub-balanced unsorted leaf scheme compared with the unsorted leaf scheme. It has similar read and write access counts and memory energy consumption in PCM, STT-RAM, and ReRAM systems to the unsorted leaf scheme, as shown in Figure 5.3(h), (i), (d), (e), and (f).

When $node\_size = 2$ ($b = 6$), from Figure 5.3(i), it is found that neither of the two

unsorted leaf schemes can reduce the write access count. From Figure 5.3(a), (b), (c), (d), (e), and (f), they cannot reduce the execution time and memory energy consumption either. That is why the overflow node scheme is proposed for small *node_size*s and small *b*s. As the node size increases, both the two unsorted leaf schemes demonstrate their efficiency in reducing the write access count and the total memory energy consumption compared with the original B$^+$-tree; however, in the meanwhile, the read access count increases.

Figure 5.4 demonstrates how the two unsorted leaf schemes affect the search performance compared with original B$^+$-tree. In this experiment, every record of a tree with one million records is searched in a random order. The results show that when $node\_size = 2, 4, 8$ ($b = 6, 13, 25$), the two unsorted leaf schemes incur little or affordable performance overhead or memory energy consumption overhead. However, when $node\_size = 16$ ($b = 51$), compared with B$^+$-tree, they have 9.3% increase in execution time and 13.3% increase in total memory energy consumption for PCM based system, 15.4% increase in execution time and 15.0% increase in total memory energy consumption for STT-RAM based system, and 15.1% increase in execution time and 12.2% increase in total memory energy consumption for ReRAM based system. It is because as the branching factor increases, the cost (instruction count and read access count) of linear search in the unsorted leaves becomes higher; when the branching factor is large enough, the cost of linear search becomes dominant, as shown in Figure 5.4(g) and (h). The write access count stays stable for the search-only workload, among different node sizes and different B$^+$-tree schemes as shown in Figure 5.4(i). These results indicate that, although the unsorted leaf scheme and the sub-balanced unsorted leaf scheme can reduce the write count more efficiently for larger branching factors in insert operations, they may degrade the search performance significantly.

For delete operations, the unsorted leaf scheme and the sub-balanced unsorted leaf



(a) PCM execution time (b) STT-RAM execution time (c) ReRAM execution time

(d) PCM memory energy consumption (e) STT-RAM memory energy consumption (f) ReRAM memory energy consumption

(g) Instruction count (h) Read access count (i) Write access count

Figure 5.4: Comparison among B$^+$-tree, the unsorted leaf scheme, and the proposed sub-balanced unsorted leaf scheme in PCM, STT-RAM, and ReRAM based systems for a search-only workload (search every record of a tree with one million records in random order). The results are normalized to those of B$^+$-tree with $node\_size = 2$.

scheme have shorter execution time than the original B$^+$-tree in PCM, STT-RAM, and ReRAM based systems when $node\_size = 4$, as shown in Figure 5.6. The reason is that, they remove the CPU-intensive sorting in each delete operation. For the delete-only workload, when the merging factor $mgf = 0.5$, the unsorted leaf scheme reduces the instruction count by 4.8%, and saves the execution time by 3.8%, 4.1%, and 4.1% in PCM, STT-RAM and ReRAM based systems respectively, compared with the original B$^+$-tree; the sub-balanced unsorted leaf node scheme reduces the instruction count by

5.3%, and saves the execution time by 4.5%, 4.6%, and 4.6% in PCM, STT-RAM and ReRAM based systems respectively. Their read and write access counts, and memory energy consumptions in PCM, STT-RAM and ReRAM based systems, and memory space usages are similar to those of the original B$^+$-tree for the delete-only workload when $mgf = 0.5$. It also can be found that, when $node\_size = 4$, the unsorted leaf node scheme and the sub-balanced leaf node scheme cannot reduce the write accesses efficiently in delete operations.

Results for The Overflow Node Scheme

The overflow node scheme is evaluated for the cases of small node sizes and branching factors in PCM, STT-RAM, and ReRAM based systems. Figure 5.5 presents the results of the overflow node scheme with overflow factors (ovf) from 1 to 4 when $node\_size = 2$ and $b = 6$. For the insert-only workload, as Figure 5.5(a) shows, the overflow node scheme reduces the write count by $6.2\% - 12.9\%$, and saves memory energy by $4.0\% - 8.3\%$ in PCM based system, by $3.5\% - 7.1\%$ in STT-RAM based system, and by $4.3\% - 9.0\%$ in ReRAM based system, as the overflow factor increases from 1 to 4, without hurting the performance. This demonstrates the effectiveness of the overflow node scheme to reduce the write accesses in insert operations. However, for the search-only workload, the overflow node scheme increases the read access number by $0.4\% - 12.2\%$, as shown in Figure 5.5(b), and thus incurs about $1.6\% - 5.2\%$ performance overhead and increases the total memory energy consumption by $2.0\% - 8.6\%$ in PCM, STT-RAM, and ReRAM based systems, with the overflow factor from 1 to 4. Therefore, when considering the performance and memory energy consumption of both the insert and search operations, $ovf = 1$ and $ovf = 2$ are two good choices.

For delete operations, Figure 5.6 shows the results of the overflow node scheme with $node\_size = 4$ and $ovf = 2$. Compared with the original B$^+$-tree, it decreases the

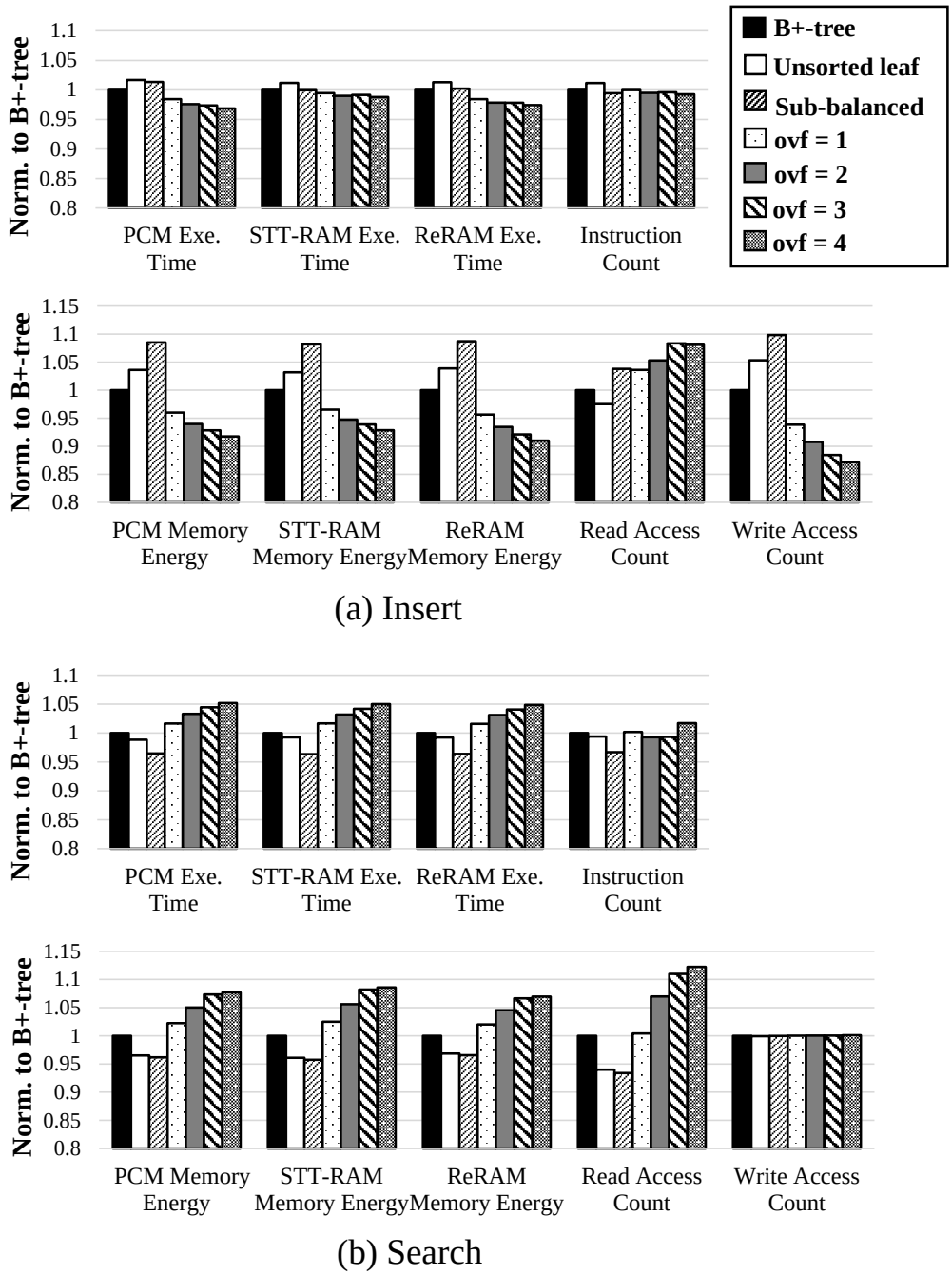


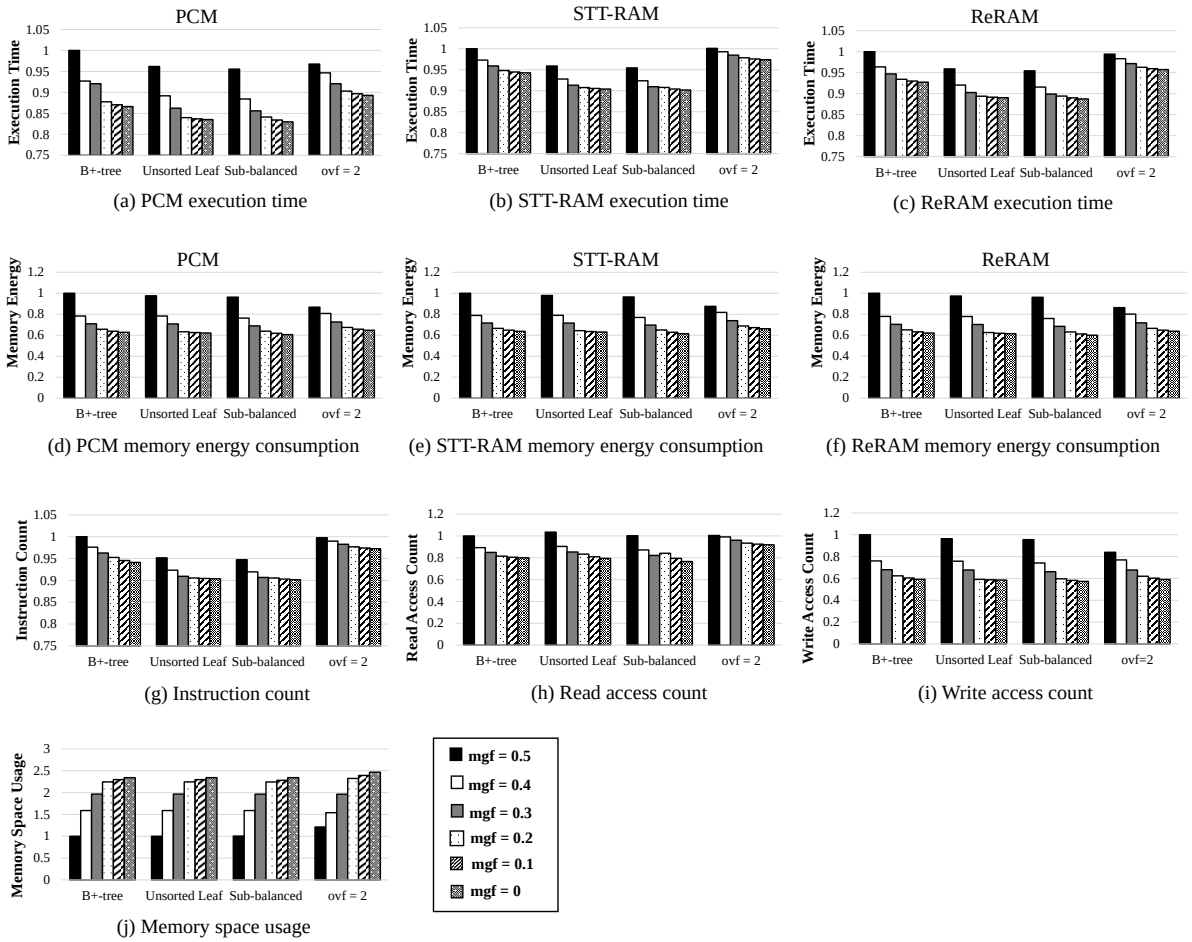
Figure 5.5: Comparison results of the overflow node schemes for (a) insert-only and (b) search-only workloads in PCM, STT-RAM, and ReRAM based systems when $node\_size = 2$ and $b = 6$ (normalized to the B$^+$-tree results).

write accesses by 16.0% when $mgf = 0.5$, as shown in Figure 5.6(i), because the write accesses due to merge and borrow are reduced since a leaf node can borrow keys from and

merge with an overflow node without updating the parent node. In PCM based system, it reduces the execution time by 3.2% when $mgf = 0.5$, as shown in Figure 5.6(a). However, in STT-RAM and ReRAM based systems, the execution time is similar to that of the original B$^+$-tree, as shown in Figure 5.6(b) and (c). This is because the write latencies of STT-RAM and ReRAM are much shorter than that of PCM, and the reductions of write accesses in STT-RAM and ReRAM based systems have less impact on the execution time than in PCM based system. Moreover, Figure 5.6(d), (e) and (f) show that, the overflow node scheme saves memory energy consumption by 13.2%, 12.5%, and 13.7% in PCM, STT-RAM and ReRAM based systems, respectively. These benefits come at the cost of 21.2% more memory space, as shown in Figure 5.6(j). The reason is that a leaf can be sparser before incurring a merge since it can borrow keys from either an overflow node or a neighbor node.

Results for The Merging Factor Scheme

The merging factor scheme is implemented with the merging factor from 0 to 0.5. The original B$^+$-tree, the unsorted leaf node scheme, the sub-balanced unsorted leaf node scheme, and the overflow node scheme with $ovf = 2$, are evaluated in PCM, STT-RAM, and ReRAM based systems, for a delete-only workload. The results are shown in Figure 5.6. It can be found that with the decrease of the merging factor from 0.5 to 0, the write accesses of all the four schemes are decreased (up to $\sim 40\%$) and so are their execution time, instruction count, read access count, and memory energy consumption (decreased by up to $\sim 35\%$). However, the space usage is increased dramatically (up to $\sim 2.5\times$). For MMDB applications, space efficiency is also an important goal for algorithm design. With the merging factor scheme, a proper merging factor can be chosen to make better trade-offs among execution time, total wear, memory energy and space usage.



(a) PCM execution time (b) STT-RAM execution time (c) ReRAM execution time

(d) PCM memory energy consumption (e) STT-RAM memory energy consumption (f) ReRAM memory energy consumption

(g) Instruction count (h) Read access count (i) Write access count

(j) Memory space usage

Figure 5.6: The results of the merging factor scheme in PCM, STT-RAM, and ReRAM based systems for a delete-only workload (randomly delete half the records from a tree holding one million records with random keys) with $node\_size = 4$ (normalized to the B$^+$-tree results).

5.6 Summary

Among the emerging NVM technologies, PCM, STT-RAM, and ReRAM, are becoming promising to build future energy-efficient main memory systems. They will benefit MMDB systems with their nice features. This paper focuses on making B$^+$-tree NVM-friendly. A new algorithm design goal is to reduce the write accesses that have long latency, high energy consumption, and endurance issues. In this project, a basic cost model for NVM-based memory systems is presented which distinguishes writes from reads

according to NVM's asymmetric read/write characteristics, and also the CPU costs and memory accesses for search, insert and delete operations on a B$^+$-tree are formulated. The model is then used to analyze the existing NVM-friendly B$^+$-tree schemes, *i.e.*, the unsorted node schemes, and find that they suffer from three problems. Consequently, three schemes are proposed to address these challenges: 1) the sub-balanced unsorted node scheme, 2) the overflow node scheme, and 3) the merging factor scheme. Experimental results show that they can provide more algorithm options for making trade-offs among performance improvement, NVM lifetime extension, memory energy saving, and space usage reduction under different workloads.

Chapter 6

Accelerating Neural Network Computation in ReRAM Based Main Memory

This chapter presents a project that accelerates neural network (NN) computation with a novel processing-in-memory (PIM) architecture, called PRIME (processing in ReRAM based main memory) [66]. It first introduces the motivation of this project. Then, it gives the basics of using ReRAM for NN computation, and introduces the related work about PIM and accelerating NN in hardware. Next, it presents the PRIME architecture design as well as hardware-software interface design. Furthermore, it shows the evaluation methodology and results before summarizes this project.

6.1 Motivation

The gap between microprocessor and memory speeds has kept increasing for several decades. To mitigate this issue, processing-in-memory (PIM) has been proposed since the

1990s, which integrates computation logic with main memory [67, 68, 69], storage [70], and last level cache [71]. In recent years, as the amount of data to process grows exponentially, data movement between the processing units (PUs) and the memory is becoming one of the most critical performance and energy bottlenecks in various computer systems. PIM, as a promising solution, again attracts a lot of interest from academia as well as industry, by leveraging emerging 3D memory technologies to integrate logic with memory [72, 73, 74, 75].

All the PIM work mentioned above integrates additional logic with memory, which increases the total cost of memory chips significantly. Recent work demonstrated that some emerging NVMs, including ReRAM, STT-RAM, and PCM, have the capability of performing logic and arithmetic operations beyond data storage. This allows the memory to serve both computation and memory functions, promising a radical renovation of the relationship between computation and memory. Among them, ReRAM can perform matrix-vector multiplications efficiently in a crossbar structure, and has been widely studied to represent synapses in neural computation [76, 77, 78, 79, 80, 81, 82].

Neural network (NN) and deep learning (DL) have the potential to provide optimal solutions in various applications including image/speech recognition and natural language processing, and are gaining a lot of attention recently. The state-of-the-art NN and DL algorithms, such as multi-layer perceptron (MLP) and convolutional neural network (CNN), require a large memory capacity as the size of NN increases dramatically (e.g., 1.32GB synaptic weights for Youtube video object recognition [83]). High-performance acceleration of NN requires high memory bandwidth since the PUs are hungry for fetching the synaptic weights [84]. To address this challenge, recent special-purpose chip designs have adopted large on-chip memory to store the synaptic weights. For example, DaDianNao [85] employed a large on-chip eDRAM for both high bandwidth and data locality; TrueNorth utilized an SRAM crossbar memory for synapses in each core [86]. Although

those solutions effectively reduce the transfer of synaptic weights between the PUs and the off-chip memory, the data movement including input and output data besides synaptic weights is still a hinderance to performance improvement and energy saving. Instead of integrating more on-chip memory, PIM is a promising solution to tackle this issue by putting the computation logic into the memory chip, so that NN computation can enjoy the large capacity of main memory and sustain high memory bandwidth via in-memory data communication at the same time.

This project proposes a novel PIM architecture using ReRAM as main memory and also for efficient NN computation, called PRIME (processing in ReRAM-based main memory). ReRAM has been previsioned to build the next-generation main memory [10], and is also a good candidate for PIM thanks to its large capacity, fast read speed, and computation capability. In the ReRAM main memory design, a small portion of memory arrays in each bank are enabled to serve as NN accelerators besides normal memory by specific peripheral circuit design. The circuit, architecture, and software interface designs allow these ReRAM arrays to dynamically reconfigure between memory and accelerators, and also to represent various NNs. The current PRIME design supports large-scale MLPs and CNNs, which can produce the state-of-the-art performance on varieties of NN applications, e.g. top classification accuracy for image recognition tasks. Distinguished from all prior work on NN acceleration, PRIME can benefit from both the efficiency of using ReRAM for NN computation and the efficiency of the PIM architecture to reduce the data movement overhead, and therefore can achieve significant performance gain and energy saving. As no dedicated processor is required, PRIME incurs very small area overhead. It is also manufacture friendly with low cost, since it remains as the memory design without requirement for complex logic integration or 3D stacking.

6.2 The Basics of Using ReRAM for NN Computation

Artificial neural networks (ANNs) are a family of machine learning algorithms inspired by the human brain structure. Generally, they are presented as network of interconnected neurons, containing an input layer, an output layer, and probably one or more hidden layers. Figure 6.1 (a) shows a simple neural network with an input layer of three neurons, an output layer of three neurons, and no hidden layers. The output b_j is calculated as,

$$b_j = \sigma(\sum_{\forall i} a_i \cdot w_{i,j}), \tag{6.1}$$

where $w_{i,j}$ are synaptic weights, σ is a non-linear function, $i = 1, 2, 3$, and $j = 1, 2, 3$.

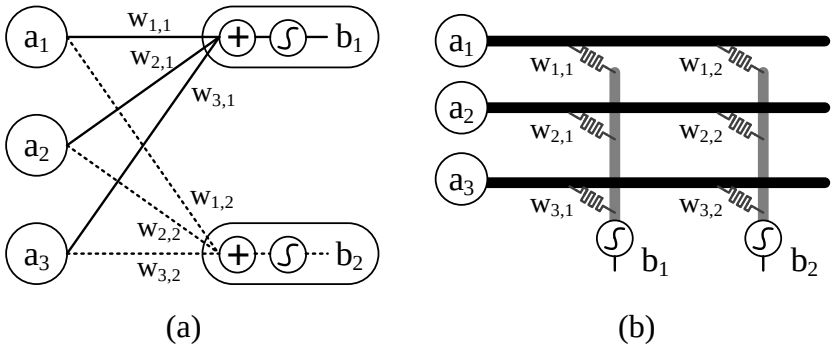


Figure 6.1: (a) An ANN with one input/output layer; (b) using a ReRAM crossbar array for neural computation.

Figure 6.1 (b) shows an example of using a 3×3 ReRAM crossbar array to execute the neural networks in Figure 6.1 (a). The input data a_i is represented by analog input voltages on the wordlines. The synaptic weights $w_{i,j}$ are programmed into the cell conductances in the crossbar array. Then the current flowing to the end of each bitline is viewed as the result of the matrix-vector multiplication, $\sum_i a_i \cdot w_{i,j}$. After sensing the current on each bitline, the neural networks adopt a non-linear function unit to complete

the execution.

Implementing NNs with ReRAM crossbar arrays requires specialized peripheral circuit design. For example, digital-to-analog converters (DACs) and analog-to-digital converters (ADCs) are needed for analog computing. Also, a sigmoid unit as well as a substraction unit is required, since matrices with positive and negative weights are implemented as two separated crossbar arrays.

6.3 Related Work

6.3.1 Processing-in-memory (PIM)

PIM is not a new concept, and there has been a lot of work on it since 1990s, e.g., IRAM [67, 87, 88, 89] and DIVA [69]. Early efforts explored to integrate simple ALU [90], vectorization [87], SIMD [68], general processor [91], and FPGA [92] with DRAM. Unfortunately, the idea of integrating performance-optimized logic with density-optimized memory aroused a lot of criticism from the cost-sensitive memory industry [93]. Recently, driven by the data intensive applications and the 3D-stacking technology, PIM or near data computing (NDC) is resurgent, with lots of industry effort (e.g., IBM [94], AMD [74], and Samsung [95]). Recent efforts decouple logic and memory designs in different dies, adopting 3D stacked memories with a logic layer that encapsulates processing units to perform computation [72, 73, 74, 75, 96]. This architecture design is compatible with the hybrid memory cube (HMC) [97] and high bandwidth memory (HBM) [98].

In this project, the proposed PRIME is a distinct solution from either early or recent PIM work. Instead of adding logic to memory, PRIME leverages the memory cells themselves for computing, and hence the area overhead is very small. The add-on hardware in PRIME to enable the computing functionality is simple modifications of the existing

memory peripheral circuits, which is much more manufacture-friendly than integrating complex logic into the memory die. Moreover, PRIME does not rely on 3D-stacking technology, exempt from its high cost and thermal problems. Also, different from previous work, which is usually focused on database and graph processing applications [75, 73], PRIME aims at accelerating NN applications.

Recent work also employs NVM technologies (ReRAM, phase change memory, and spin-transfer torque RAM) to build ternary content addressable memories (TCAMs), which exploits memory cells to perform associative search operations [99, 100, 101]. However, to support such search operations, these studies require redesign of NVM cell structures to enable much larger cell sizes, which inevitably increases the cost of memory. Compared to previous TCAM designs, PRIME obviates the cost of redesigning memory cells. Furthermore, PRIME supports much more sophisticated computation than TCAMs.

6.3.2 Accelerating NNs in Hardware

In the era of big data, machine learning is widely used to learn from and make predictions on huge amount of data. With the advent of deep learning, some neural network algorithms such as convolutional neural networks (CNNs) and deep neural networks (DNNs) start to show their power and effectiveness across a wide range of applications [84, 85]. Prior studies [102, 86, 103] strive to build neuromorphic systems with CMOS-based neurons and synapses, which introduce substantial design challenges due to the huge area occupied by ten thousands of transistors used to implement numerous neurons and synapses. Alternatively, ReRAM is becoming a promising candidate to build area-efficient synaptic arrays for NN computation [76, 77, 78, 80], as it emerges with crossbar architecture. Recently, Presioso *et al.* fabricated a 12×12 ReRAM cross-

bar prototype with a fully operational neural network, successfully classifying 3×3-pixel black/white images into 3 categories [78].

Most prior work exploit ReRAM either as DRAM/flash replacement [104, 105, 10] or as synapses for NN computation [76, 77, 78, 79, 80, 106]. In this project, PRIME is a morphable architecture, where ReRAM can work as main memory as well as NN computation units, by enabling a small portion of the subarrays in ReRAM based main memory with the NN computation functionality, referred as full function subarrays. When NN applications are running, PRIME can execute them with the full function subarrays to improve performance or energy efficiency; while the NN applications are not executed, the full function subarrays can be freed to provide extra memory capacity.

Distinguished from the existing work on accelerating NNs in hardware, PRIME proposes a PIM solution. There is a lot of previous studies on using ReRAM for NN acceleration, from stand-alone accelerator [76], co-processor [77], to many-core or NoC [106] architecture. There are also many other studies on accelerating NNs on the platforms of GPU [107, 108, 109], FPGA [110, 111, 112] and ASIC [86, 103, 102, 84, 85]. The efforts of prior work focus on the co-processor architecture that accesses data from main memory in a conventional way. However, some NN applications require a high memory bandwidth to fetch large-size input data or synaptic weights, and the data movement from memory to processors is energy-consuming. As reported, DRAM accesses consume 95% of the total energy in DianNao design [84]. This project proposes to accelerate NNs in a PIM architecture, moving the computing resources to the memory side by adapting a small portion of the subarrays in ReRAM based main memory for NN computation. It takes advantage of the large internal bandwidth of main memory, and also provides a more energy efficient solution with minimal data movement.

6.4 The Proposed Design

Processing in ReRAM-based main memory, PRIME, is proposed which efficiently accelerates NN computation by leveraging ReRAM's computation capability and the PIM architecture. In this section, both the architecture-level and the system-level designs are presented.

6.4.1 Architecture Design

Figure 6.2(c) depicts an overview of the PRIME architecture design. While most previous NN acceleration approaches require additional processing units (PU) (Figure 6.2 (a) and (b)), PRIME directly leverages ReRAM cells to perform computation without the need for extra PUs. To achieve this, as shown in Figure 6.2(c), PRIME partitions a ReRAM bank into three regions: memory (Mem) subarrays, full function (FF) subarrays, and Buffer subarrays.

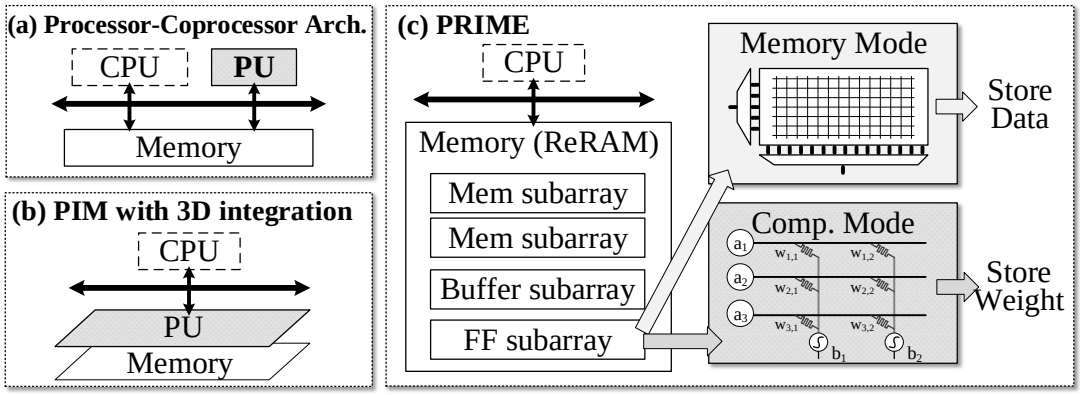


Figure 6.2: (a) Traditional shared memory based processor-coprocessor architecture, (b) PIM approach using 3D integration technologies, (c) PRIME design.

The Mem subarrays only have data storage capability (the same as conventional memory subarrays). Their microarchitecture and circuit designs are similar to a recent design of performance-optimized ReRAM main memory [10]. The FF subarrays have

both computation and data storage capabilities, and they can operate in two modes. In memory mode, the FF subarrays serve as conventional memory; in computation mode, they can execute NN computation. There is a PRIME controller to control the operation and the reconfiguration of the FF subarrays. The Buffer subarrays serve as data buffers for the FF subarrays, and they are the memory subarrays that are closest to each FF subarray. They are connected to the FF subarrays that they serve through private data ports, so that buffer accesses do not consume the bandwidth of the Mem subarrays. While not being used as data buffers, the Buffer subarrays can also be used as normal memory. From Figure 6.2(c), it can be found that for NN computation, the FF subarrays enjoy the high bandwidth of in-memory data movement, and can work in parallel with CPU, with the help of the Buffer subarrays.

To enable the NN computation function in FF subarrays, it needs to modify decoders and drivers, column multiplexers (MUX), and sense amplifiers (SA) in circuit design, and also needs to add connection units to connect the FF subarrays and the Buffer subarrays. To minimize the area overhead, the reuse of peripheral circuits is maximized for both storage and computation in FF subarrays. The goal of the Buffer subarray is two-fold. First, they are used to cache the input and output data for the FF subarrays. For example, to fetch data for the FF subarrays, the data are first loaded from a Mem subarray to the global row buffer, and then they are written from the row buffer to the Buffer subarray. Second, the FF subarrays can communicate with the Buffer subarrays directly without the involvement of the CPU, so that the CPU and the FF subarrays can work in parallel. The adjacent memory subarray to the FF subarrays is chosen as the Buffer subarray, which is also closest to the global row buffer so as to minimize the delay.

The current circuit design of the FF subarrays supports the acceleration of MLP and most layers of CNN. Matrix-vector multiplication is one of the most important

operations in MLP and the fully-connected layer and convolution layer of CNN, which can be implemented by ReRAM crossbar arrays: the weight matrix is pre-programmed in ReRAM cells; the input vector is the voltages on the wordlines driven by the drivers, and the output currents are accumulated at the bitlines. Actually, the synaptic weight matrix is separated into two matrices, one storing the positive weights and the other storing the negative weights, and they are programmed into two crossbar arrays. A subtraction unit is then used to subtract the result of the negative part from that of the positive part. For activation functions, the circuit design support sigmoid and ReLU functions by specific sigmoid and ReLU units. For the pooling layer, both max pooling and mean pooling are supported. To implement max pooling, a 4:1 max pooling hardware is adopted which is able to support n:1 max pooling with multiple steps for $n > 4$. Mean pooling is easier to implement than max pooling, because it can be done with ReRAM and does not require extra hardware. To perform n:1 mean pooling, it simply pre-programs the weights $[1/n, \cdots, 1/n]$ in ReRAM cells, and executes the dot product of the inputs and the weights to obtain the mean value of n inputs. Currently, PRIME does not support local response normalization layer (LRN) acceleration. The hardware for LRN is not added, because state-of-the-art CNNs do not contain LRN layers [113]. When LRN layers are applied, PRIME requires the help of CPU for LRN computation.

6.4.2 Software-Hardware Interface

Figure 6.3 shows the stack of PRIME to support NN programming, which allows developers to easily configure the FF subarrays for NN applications. From software programming to hardware execution, there are three stages: programming (coding), compiling (code optimization), and code execution. In the programming stage, PRIME provides application programming interfaces (APIs) so that they allow developers to: 1)

map the topology of the NN to the FF subarrays, *Map_Topology*, 2) program the synaptic weights into mats, *Program_Weight*, 3) configure the data paths of the FF subarrays, *Config_Datapath*, 4) run computation, *Run*, and 5) post-process the result, *Post_Proc*. In this work, the training of NN is done off-line so that the inputs of each API are already known (*NN param.file*). Prior work explored to implement training with ReRAM crossbar arrays [114, 115, 116, 117, 118, 119], and it can be future work to further enhance PRIME with the training capability.

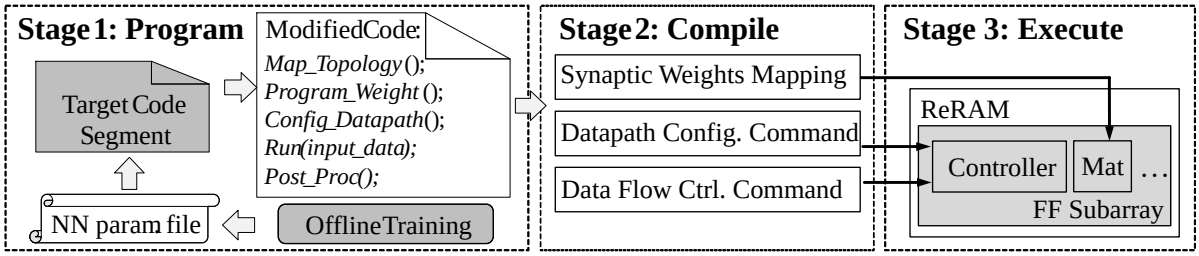


Figure 6.3: The software perspective of PRIME: from source code to execution.

In the compiling stage, the NN mapping to the FF subarrays and the input data allocation are optimized. The output of compiling is the metadata for synaptic weights mapping, data path configuration, and execution commands with data dependency and flow control. The metadata is also the input for the execution stage. In the execution stage, PRIME controller writes the synaptic weights to the mapped addresses in the FF subarrays; then it (re-)configures the peripheral circuits according to the datapath configuration commands to set up the data paths for computation; and finally, it executes data flow control commands to manage data movement into or out of the FF subarrays at runtime.

Since FF subarrays reside in banks, PRIME intrinsically inherits bank-level parallelism to speed up execution. Note that the FF subarrays in different banks have the same computing configurations. For instance, if a bank is considered as an NPU, PRIME contains 64 NPUs per bank (8 banks×8 chips) so that 64 images can be processed in

parallel. To take advantage of the bank-level parallelism, the OS is required to place one image in one bank, and evenly distribute images to all banks. As current page placement strategies expose memory latency or bandwidth information to the OS [120, 121], PRIME further exposes the bank ID information to the OS, so that each image can be mapped to a single bank.

6.5 Evaluation

In this section, the PRIME design is evaluated. First, the experiment setup is described, and then the performance and energy results are presented and the area overhead is estimated.

6.5.1 Experimental Methodology

Benchmark

The benchmarks used (*MlBench*) comprise six NN designs for machine learning applications, as listed in Table 6.1. *CNN-1* and *CNN-2* are two CNNs, and *MLP-S/M/L* are three MLPs with different network scales: small, medium, and large. Those five NNs are evaluated on the widely used *MNIST* database of handwritten digits [122]. The sixth NN, *VGG-D*, is well known for ImageNet ILSVRC[113]. It is an extremely large CNN, containing 16 weight layers and 1.4×10^8 synapses, and requiring $\sim 1.6 \times 10^{10}$ operations.

PRIME Configurations

There are 2 FF subarrays and 1 Buffer subarray per bank (totally 64 subarrays). In FF subarrays, for each mat, there are 256×256 ReRAM cells and eight 6-bit reconfigurable SAs; for each ReRAM cell, it assumes 4-bit MLC for computation while SLC for memory;

Table 6.1: The Benchmarks and Topologies.

MlBench		*MLP-S*	784-500-250-10
CNN-1	conv5x5-pool-720-70-10	*MLP-M*	784-1000-500-250-10
CNN-2	conv7x10-pool-1210-120-10	*MLP-L*	784-1500-1000-500-10
VGG-D	conv3x64-conv3x64-pool-conv3x128-conv3x128-pool conv3x256-conv3x256-conv3x256-pool-conv3x512 conv3x512-conv3x512-pool-conv3x512-conv3x512 conv3x512-pool-25088-4096-4096-1000		

the input voltage has 8 levels (3-bit) for computation while 2 levels (1-bit) for memory. With the input and synapse composing scheme, for computation, the input and output data are 6-bit dynamic fixed point, and the weights are 8-bit.

Methodology

PRIME is compared with several counterparts. The baseline is a CPU-only solution. The configurations of CPU and ReRAM main memory are shown in Table 6.2, including key memory timing parameters for simulation. Two different NPU solutions are also evaluated: using a complex parallel NPU [84] as a co-processor (pNPU-co), and using the NPU as a PIM-processor through 3D stacking (pNPU-pim). The configurations of these comparatives are described in Table 6.3.

Table 6.2: Configurations of CPU and Memory.

Processor	4 cores; 3GHz; Out-of-order
L1 I&D cache	Private; 32KB; 4-way; 2 cycles access;
L2 cache	Private; 2MB; 8-way; 10 cycles access;
ReRAM-based Main Memory	16GB ReRAM; 533MHz IO bus; 8 chips/rank; 8 banks/chip; tRCD-tCL-tRP-tWR 22.5-9.8-0.5-41.4 (ns)

The above NPU designs is modeled using Synopsys Design Compiler and PrimeTime with $65nm$ TSMC CMOS library. The ReRAM main memory and the PRIME system are modeled with modified NVSim [33], CACTI-3DD [123] and CACTI-IO [124]. Pt/TiO2-

Table 6.3: The Configurations of Comparatives.

Description		Data path	Buffer
pNPU-co	Parallel NPU as co-processor, similar to DianNao [84]	16x16 multiplier, 256-1 adder tree	2KB in/out 32KB weight
pNPU-pim	PIM version of the parallel NPU, 3D stacked to each bank		

x/Pt devices [125] with $R_{\text{on}}/R_{\text{off}} = 1k\Omega/20k\Omega$ and 2V SET/RESET voltage are adopted. The FF subarray is modeled by heavily modified NVSim, according to the peripheral circuit modifications, i.e., write driver [126], sigmoid [127], and sense amplifier [128] circuits. A trace-based in-house simulator is built to evaluate different systems, including CPU-only, PRIME, NPU co-processor, and NPU PIM-processor.

6.5.2 Experimental Results

Performance Results

The performance results for *MlBench* are presented in Figure 6.4. *MlBench* benchmarks use large NNs and require high memory bandwidth, and therefore they can benefit from PIM. To demonstrate the PIM advantages, two pNPU-pim solutions are evaluated: pNPU-pim-x1 is a PIM-processor with a single parallel NPU stacked on top of memory; and pNPU-pim-x64 with 64 NPUs, for comparison with PRIME which takes advantages of bank-level parallelism (64 banks). By comparing the speedups of pNPU-co and pNPU-pim-x1, it is found that the PIM solution has a 9.1× speedup on average over a co-processor solution. Among all the solutions, PRIME achieves the highest speedup over the CPU-only solution, about 4.1× of pNPU-pim-x64's. PRIME achieves a smaller speedup in *VGG-D* than other benchmarks, because it has to map the extremely large *VGG-D* across 8 chips where the data communication between banks/chips is costly. The performance advantage of PRIME over the 3D-stacking PIM solution (pNPU-pim-x64) for NN applications comes from the efficiency of using ReRAM for NN computation,

because the synaptic weights have already been pre-programmed in ReRAM cells and do not require data fetches from the main memory during computation. In the performance and energy evaluations of PRIME, it does not include the latency and energy consumption of configuring ReRAM for computation, because it assumes that once the configuration is done, the NNs will be executed for tens of thousands times to process different input data.

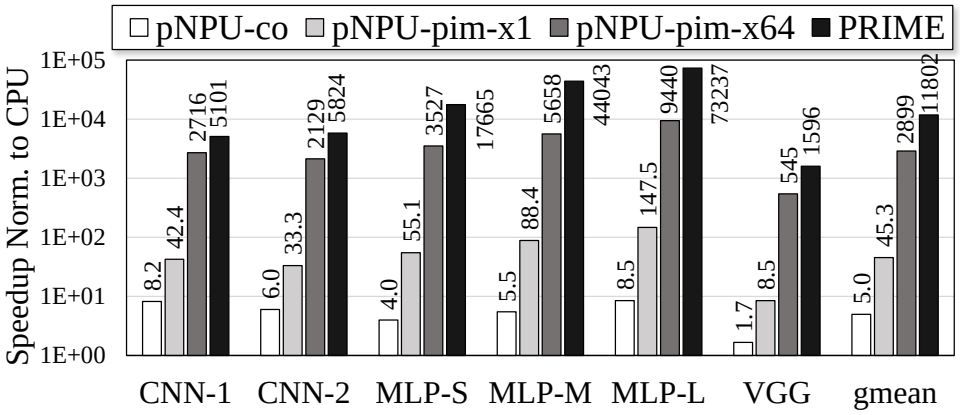


Figure 6.4: The performance speedups normalized to CPU-only.

Figure 6.5 presents the breakdown of the execution time normalized to pNPU-co. To clearly show the breakdown, the results of pNPU-pim with one NPU, and PRIME without leveraging bank parallelism for computation are evaluated. The execution time is divided into two parts, computation and memory access. The computation part also includes the time spent on the buffers of NPUs or the Buffer subarrays of PRIME in managing data movement. It is found that pNPU-pim reduces the memory access time a lot, and PRIME further reduces it to zero. Zero memory access time does not imply that there is no memory access, but it means that the memory access time can be hidden by the Buffer subarrays.

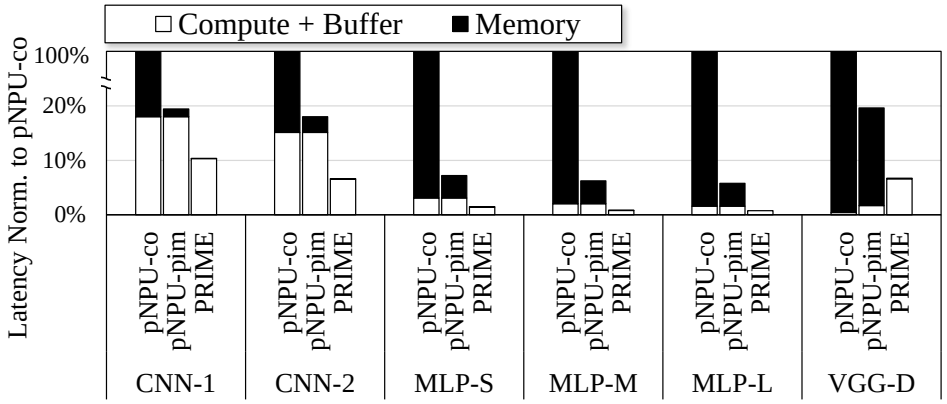


Figure 6.5: The execution time breakdown normalized to pNPU-co).

Energy Results

The energy saving results for *MlBench* are presented in Figure 6.6. Figure 6.6 does not show the results of pNPU-pim-x1, because they are the same with those of pNPU-pim-x64. From Figure 6.6, PRIME shows its superior energy-efficiency to other solutions. pNPU-pim-x64 is several times more energy efficient than pNPU-co, because the PIM architecture reduces memory accesses and saves energy. The energy advantage of PRIME over the 3D-stacking PIM solution (pNPU-pim-x64) for NN applications comes from the energy efficiency of using ReRAM for NN computation.

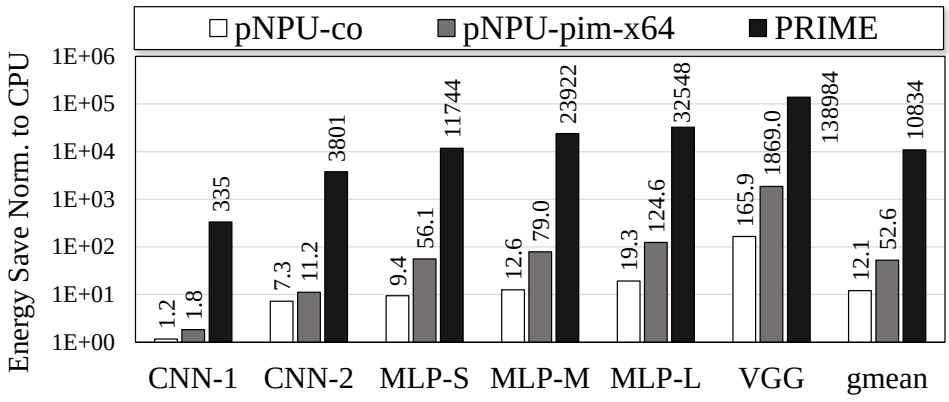


Figure 6.6: The energy saving results normalized to CPU-only.

Figure 6.7 provides the breakdown of the energy consumption normalized to pNPU-co. The total energy consumptions are divided into three parts, computation energy, buffer energy, and memory energy. From Figure 6.7, pNPU-pim-x64 consumes almost the same energy in computation and buffer with pNUP-co, but saves the memory energy by 93.9% on average by decreasing the memory accesses and reducing memory bus and I/O energy. PRIME reduces all the three parts of energy consumption significantly. For computation, ReRAM based analog computing is very energy-efficient. Moreover, since each ReRAM mat can store 256×256 synaptic weights, the cache and memory accesses to fetch the synaptic weights are eliminated. Furthermore, since each ReRAM mat can execute as large as a $256 - 256$ NN at one time, PRIME also saves a lot of buffer and memory accesses to the temporary data. From Figure 6.7, CNN benchmarks consume more energy in buffer and less energy in memory than MLP benchmarks. The reason is that the convolution layers and pooling layers of CNN usually have a small number of input data, synaptic weights, and output data, and buffers are effective to reduce memory accesses.

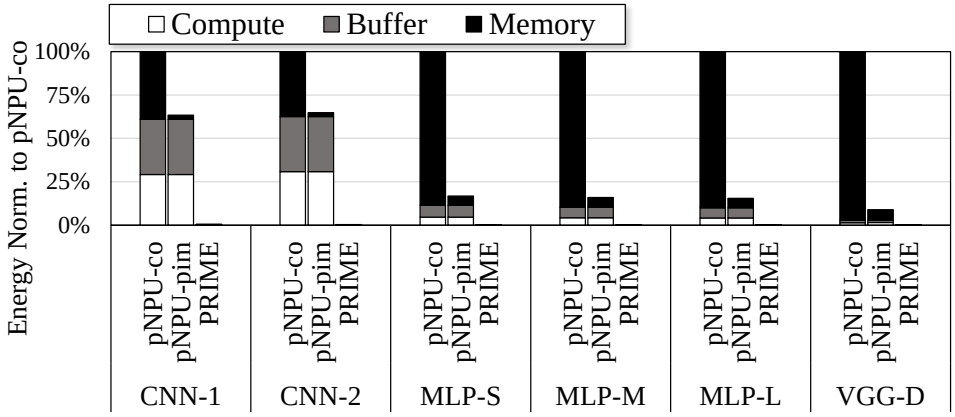


Figure 6.7: The energy consumption breakdown normalized to pNPU-co.

Area Overhead

Given two FF subarrays and one Buffer subarray per bank (64 subarrays in total), PRIME only incurs 5.76% area overhead. The choice of the number of FF subarrays is a tradeoff between peak GOPS and area overhead. The experimental results on *Mlbench* (except VGG-D) show that the utilities of FF subarrays are 39.8% and 75.9% on average before and after replication, respectively. For *VGG-D*, the utilities of FF subarrays are 53.9% and 73.6% before and after replication, respectively. Figure 6.8 shows the breakdown of the area overhead in a mat of an FF subarray. There is 60% area increase to support computation: the added driver takes 23%, the subtraction and sigmoid circuits take 29%, and the control, the multiplexer, and etc. cost 8%.

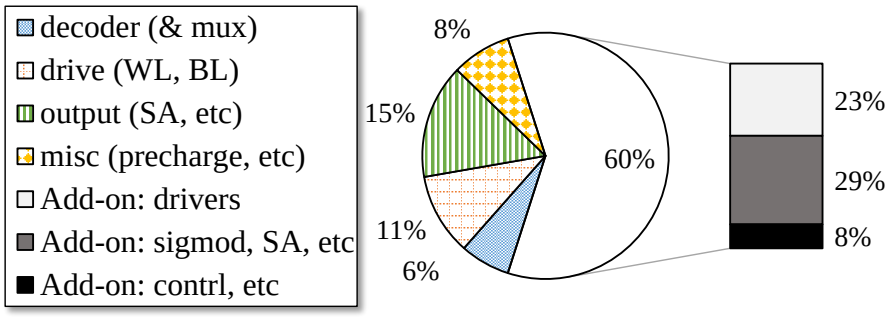


Figure 6.8: The area overhead of an FF subarray.

6.6 Summary

In this project, a novel processing in ReRAM-based main memory design, PRIME, is proposed which substantially improves the performance and energy efficiency for neural network (NN) applications, benefiting from both the PIM architecture and the efficiency of ReRAM based NN computation. In PRIME, part of the ReRAM memory arrays are enabled with NN computation capability. They can either perform computation

to accelerate NN applications or serve as memory to provide a larger working memory space. The architecture-level and system-level designs are presented in this chapter. The experimental results show that, PRIME can achieves a high speedup and significant energy saving for various NN applications using MLP and CNN.

Chapter 7

Conclusion

As computation is increasingly limited by data movement and energy consumption, memory system design becomes more and more important. In conventional computer systems, SRAM and DRAM are the common embodiments of different levels of the memory hierarchy. However, they are suffering from scalability issues, such as increasing leakage power and degraded reliability. In recent years, multiple NVM technologies are emerging, including STT-RAM, PCM, and ReRAM. With their attractive properties, such as good scalability, low leakage power, fast access speed, and non-volatility, they have been considered as promising candidates to replace SRAM and DRAM in future memory system design.

This dissertation is an effort to facilitate STT-RAM, PCM, and ReRAM to build next-generation memory systems. First, it is necessary to address the challenges due to their disadvantages, such as high write energy, long write latency, and limited lifetime. Second, it is beneficial to take advantage of their unique features, such as non-volatility, multi-level cell (MLC), and computation capability with a crossbar structure, to improve the system performance, energy consumption, and reliability.

MLC STT-RAM based cache design is studied first which suffers from the high write

energy problem. From the write energy model of MLC STT-RAM built, it is found that the write energy consumption is value-dependent. Based on the observation, a dynamic data-resistance encoding mechanism is proposed which maps more frequent data values to more energy-efficient resistance states at runtime. This solution can reduce the write energy consumption of MLC STT-RAM last level cache (LLC) efficiently, with a slight performance degradation due to the encoding overhead. To improve the performance when decreasing the energy consumption, the properties of MLC STT-RAM read and write operations are exploited. In series MLC STT-RAM, the soft-bit region is fast-read and fast-write, which involves only one-step read and write operations, while the hard-bit region has slow and energy-consuming two-step read and write operations. It is proposed to use data compression to compress a cache line into half its size or less and fit it into only the soft-bit region. The improved read and write latency and energy overwhelm the compression overhead, and hence improve the LLC performance and save the energy consumption. The saved space from data compression is further utilized to store more data at the cost of more complex cache management, which increases the effective cache capacity and further improves the LLC performance.

Utilizing the features of MLC STT-RAM read and write operations again, an MLC STT-RAM main memory design for efficient local checkpointing is proposed. In traditional large-scale computing systems, the data transfer between DRAM and the backup storage is the performance and energy bottleneck for checkpointing. MLC opens up an inherent multi-version opportunity for local checkpointing since each cell can store multiple bits/versions of data. In the MLC STT-RAM main memory design, the soft-bit of each cell is used to store the working data and the hard-bit to store the corresponding checkpoint data. Therefore, only one-step write operations are required during the error-free execution time as well as the checkpoint and recovery periods. The ultra high inter-cell data transfer bandwidth significantly reduces the performance overhead of local

checkpointing. Also, the proposed design achieves high energy-efficiency of creating local checkpoints.

Next, the impact of the asymmetric read and write properties of PCM, STT-RAM, and ReRAM, are explored on the indexing algorithm design for main memory databases built on these NVMs. Since write operations are much more expensive than read for these NVMs, the new algorithm design goal is to reduce write accesses, even at the cost of increasing read accesses. A cost model is built, and the CPU costs and the memory behaviors of the B^+-tree algorithm are analyzed. To address the issues of the previous NVM-friendly redesign, three new schemes are proposed, providing more algorithm options for making trade-offs among system performance, memory energy consumption, NVM lifetime, and space usage, under different workloads.

Finally, the analog computation capability of ReRAM crossbar is explored and a novel processing-in-memory (PIM) architecture built on ReRAM based main memory is proposed for accelerating neural network (NN) applications. ReRAM crossbar can perform matrix-vector multiplications very efficiently, and it has been widely studied to accelerate NN computation since matrix-vector multiplications are commonly used in NN algorithms, such as multi-layer perceptron (MLP) and convolutional neural network (CNN). This proposal is based on an existing ReRAM main memory design. In this design, a small portion of ReRAM crossbar arrays in each bank are enabled to perform NN computation by additional peripheral circuit support. Those ReRAM arrays can work in two modes: as NN accelerators or as normal memory. Since memory itself can compute, this proposal significantly reduces the data movement between the processing units and the off-chip memory which has been identified as the performance and energy bottleneck for NN applications. Along with the efficiency of the analog computation of ReRAM, the proposed PIM architecture can achieve high speedups and energy saving.

In this dissertation, the solutions are from architecture-level and application-level per-

spectives. many detailed circuit and software designs are left for future work. Moreover, checkpointing, database, and NN applications have been investigated in this dissertation. The unique characteristics of these emerging NVMs will be beneficial to other applications. Furthermore, these NVMs have only been explored as potential alternatives to SRAM and DRAM. They are also promising to replace flash as secondary storage. Since emerging NVMs can implement every level in the memory hierarchy, it is interesting to build a large flattened memory/storage system with a blurred boundary between main memory and storage using NVM, which can provide a high density and the flexibility to dynamically partition main memory and storage.

It is hoped that this dissertation would be useful and inspirational for the research on emerging NVMs in future computer system design.

Bibliography

[1] T. Zhang, M. Poremba, C. Xu, G. Sun, and Y. Xie, *CREAM: a concurrent-refresh-aware DRAM memory architecture*, in *Proc. of IEEE 20th international symposium on high performance computer architecture (HPCA'14)*, pp. 368–379, Feb., 2014.

[2] S. Ikeda, K. Miura, H. Yamamoto, K. Mizunuma, H. D. Gan, M. Endo, S. Kanai, J. Hayakawa, F. Matsukura, and H. Ohno, *A perpendicular-anisotropy CoFeB-MgO magnetic tunnel junction*, *Nature Materials* **9** (2010) 721–724.

[3] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, *Scalable high performance main memory system using phase-change memory technology*, in *Proc. of the 36th international symposium on computer architecture (ISCA'09)*, pp. 24–33, June, 2009.

[4] H.-S. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, *Metal–oxide RRAM*, *Proceedings of the IEEE* **100** (2012), no. 6 1951–1970.

[5] S. Yu, Y. Wu, and H. Wong, *Investigating the switching dynamics and multilevel capability of bipolar metal oxide resistive switching memory*, *Applied Physics Letters* **98** (2011) 103514.

[6] J. Meza, J. Li, , and O. Mutlu, *Evaluating row buffer locality in future non-volatile main memories*, in *SAFARI Technical Report No. 2012-002*, Dec, 2012.

[7] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, *Evaluating stt-ram as an energy-efficient main memory alternative*, in *Proc. of the 2013 IEEE international symposium on performance analysis of systems and software (ISPASS'13)*, pp. 256–267, April, 2013.

[8] E. Doller, *Phase change memory and its impacts on memory hierarchy*, *http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf* (2009).

[9] P. Chi, C. Xu, T. Zhang, X. Dong, and Y. Xie, *Using multi-level cell stt-ram for fast and energy-efficient local checkpointing*, in *Proceedings of the 2014*

IEEE/ACM International Conference on Computer-Aided Design, ICCAD '14, pp. 301–308, 2014.

[10] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, *Overcoming the challenges of crossbar resistive memory architectures*, in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 476–488, Feb, 2015.

[11] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, *A durable and energy efficient main memory using phase change memory technology*, in *Proc. of the 36th international symposium on computer architecture (ISCA'09)*, pp. 14–23, June, 2009.

[12] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, *Enhancing lifetime and security of PCM-based main memory with Start-Gap wear leveling*, in *Proc. of the 42nd annual IEEE/ACM international symposium on microarchitecture (MICRO'09)*, pp. 14–23, 2009.

[13] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, *Architecting phase change memory as a scalable DRAM alternative*, in *Proc. of the 36th international symposium on computer architecture (ISCA'09)*, pp. 2–13, June, 2009.

[14] S. Cho and H. Lee, *A simple deterministic technique to improve PRAM write performance, energy and endurance*, in *Proc. of the 42nd annual IEEE/ACM international symposium on microarchitecture (MICRO'09)*, pp. 347–357, 2009.

[15] X. Lou, Z. Gao, D. V. Dimitrov, and M. X. Tang, *Demonstration of multilevel cell spin transfer switching in MgO magnetic tunnel junctions*, *Applied Physics Letters* **93** (2008) 242502.

[16] T. Ishigaki, T. Kawahara, R. Takemura, K. Ono, K. Ito, H. Matsuoka, and H. Ohno, *A multi-level-cell spin-transfer torque memory with series-stacked magnetotunnel junctions*, in *2010 Symposium on VLSI Technology*, pp. 47–48, June, 2010.

[17] Y. Zhang, L. Zhang, W. Wen, G. Sun, and Y. Chen, *Multi-level cell STT-RAM: Is it realistic or just a dream?*, in *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 526–532, Nov, 2012.

[18] Y. Chen, X. Wang, W. Zhu, H. Li, Z. Sun, G. Sun, and Y. Xie, *Access scheme of multi-level cell spin-transfer torque random access memory and its optimization*, in *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, pp. 1109–1112, Aug, 2010.

[19] Y. Chen, W. F. Wong, H. Li, and C. K. Koh, *Processor caches built using multi-level spin-transfer torque RAM cells*, in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, pp. 73–78, Aug, 2011.

[20] X. Bi, M. Mao, D. Wang, and H. Li, *Unleashing the potential of MLC STT-RAM caches*, in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 429–436, Nov, 2013.

[21] J. Wang, P. Roy, W. F. Wong, X. Bi, and H. Li, *Optimizing mlc-based stt-ram caches by dynamic block size reconfiguration*, in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pp. 133–138, Oct, 2014.

[22] P. Chi, C. Xu, X. Zhu, and Y. Xie, *Building energy-efficient multi-level cell STT-MRAM based cache through dynamic data-resistance encoding*, in *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, pp. 639–644, March, 2014.

[23] M. Mishra and S. Akashe, *High performance, low power 200 Gb/s 4:1 MUX with TGL in 45 nm technology*, *Applied Nanoscience* **4** (2014) 271–277.

[24] C. Xu, D. Niu, X. Zhu, S. H. Kang, M. Nowak, and Y. Xie, *Device-architecture co-optimization of stt-ram based memory for low power embedded systems*, in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 463–470, Nov, 2011.

[25] T. E. Carlson, W. Heirman, and L. Eeckhout, *Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation*, in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12, 2011.

[26] J. L. Henning, *Spec cpu2006 benchmark descriptions*, *ACM SIGARCH Computer Architecture News* **34** (2006), no. 4 1–17.

[27] J. Dusser, T. Piquet, and A. Seznec, *Zero-content augmented caches*, in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pp. 46–55, 2009.

[28] J. Yang, Y. Zhang, and R. Gupta, *Frequent value compression in data caches*, in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pp. 258–265, 2000.

[29] A. R. Alameldeen and D. A. Wood, *Adaptive cache compression for high-performance processors*, in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pp. 212–223, 2004.

[30] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, *Base-delta-immediate compression: Practical data compression for on-chip caches*, in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 377–388, 2012.

[31] A. Arelakis *et. al.*, *Sc2: A statistical compression cache scheme*, in *Proceedings of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pp. 145–156, 2014.

[32] N. Binkert, 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, *The gem5 simulator*, *SIGARCH Comput. Archit. News* **39** (Aug., 2011) 1–7.

[33] X. Dong, C. Xu, Y. Xie, and N. Jouppi, *NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory*, *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* **31** (July, 2012) 994–1007.

[34] B. Schroeder and G. A. Gibson, *A large-scale study of failures in high-performance computing systems*, in *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pp. 249–258, 2006.

[35] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, *et. al.*, *Exascale software study: software challenges in extreme scale systems*, .

[36] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, *Hybrid checkpointing using emerging nonvolatile memories for future exascale systems*, *ACM Trans. on Architecture and Code Optimization (TACO)* **8** (2011), no. 2.

[37] R. A. Oldfield, S. Arunagiri, P. J. Teller, S. Seelam, M. R. Varela, R. Riesen, and P. C. Roth, *Modeling the impact of checkpoints on next-generation systems*, in *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pp. 30–46, Sept, 2007.

[38] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, *Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems*, in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 57:1–57:12, 2009.

[39] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, *Design, modeling, and evaluation of a scalable multi-level checkpointing system*, in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, 2010.

[40] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, *Optimizing checkpoints using nvm as virtual memory*, in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 29–40, May, 2013.

[41] D. H. Yoon, R. Schreiber, P. Faraboschi, J. Chang, N. Muralimanohar, and P. Ranganathan, *Local checkpointing using a multi-level cell*, 2013. WO Patent App. PCT/US2012/035485.

[42] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. M. A. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande, *A bipolar-selected phase change memory featuring multi-level cell storage*, *IEEE Journal of Solid-State Circuits* **44** (Jan, 2009) 217–227.

[43] M. Prvulovic, Z. Zhang, and J. Torrellas, *Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors*, in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pp. 111–122, 2002.

[44] J. S. Plank, K. Li, and M. A. Puening, *Diskless checkpointing*, *IEEE Trans. on Parallel and Distributed Systems* **9** (1998), no. 10.

[45] M. Banâtre, A. Gefflaut, P. Joubert, C. Morin, *et. al.*, *An architecture for tolerating processor failures in shared-memory multiprocessors*, *IEEE Trans. on Computers* **45** (1996), no. 10.

[46] E. N. Elnozahy and W. Zwaenepoel, *Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit*, *IEEE Trans. on Computers* **41** (1992), no. 5.

[47] T.-C. Chiueh and P. Deng, *Evaluation of checkpoint mechanisms for massively parallel machines*, in *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pp. 370–379, Jun, 1996.

[48] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, *The performance of consistent checkpointing*, in *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pp. 39–47, Oct, 1992.

[49] J. S. Plank, Y. Chen, K. Li, M. Beck, *et. al.*, *Memory exclusion: Optimizing the performance of checkpointing systems*, *Software Practice and Experience* **29** (1999), no. 2.

[50] J. Hursey, T. I. Mattox, and A. Lumsdaine, *Interconnect agnostic checkpoint/restart in open MPI*, in *HPDC*, 2009.

[51] J. L. Henning, *SPEC CPU2006 benchmark descriptions*, *SIGARCH Comput. Archit. News* **34** (Sept., 2006) 1–17.

[52] A. Hay, K. Strauss, T. Sherwook, G. H. Loh, *et. al.*, *Preventing pcm banks from seizing too much power*, in *Micro*, 2011.

[53] P. Chi, W. C. Lee, and Y. Xie, *Adapting b+-tree for emerging nov-volatile memory based main memory*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **PP** (2016), no. 99.

[54] P. Chi, W.-C. Lee, and Y. Xie, *Making b+-tree efficient in pcm-based main memory*, in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pp. 69–74, 2014.

[55] S. Chen, P. B. Gibbons, and S. Nath, *Rethinking database algorithms for phase change memory*, in *Proc. of the 5th biennial conference on innovative data systems research (CIDR'11)*, January, 2011.

[56] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, *Better I/O through byte-addressable, persistent memory*, in *Proc. of the ACM SIGOPS 22nd symposium on operating systems principles (SOSP'09)*, pp. 133–146, 2009.

[57] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, *Operating system implications of fast, cheap, non-volatile memory*, in *Proc. of the 13th USENIX conference on hot topics in operating systems (HotOS'11)*, pp. 2–2, 2011.

[58] W. Hu, *Redesign of database algorithms for next generation non-volatile memory technology*, Master's thesis, National University of Singapore, Singapore, 2013.

[59] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, *Memory access scheduling*, in *Proc. of the 27th international symposium on computer architecture (ISCA'00)*, pp. 128–138, 2000.

[60] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, *Improving read performance of phase change memories via write cancellation and write pausing*, in *Proc. of IEEE 16th international symposium on high performance computer architecture (HPCA'10)*, pp. 1–11, Jan., 2010.

[61] S. Chen, P. B. Gibbons, and T. C. Mowry, *Improving index performance through prefetching*, in *Proc. of the 2001 ACM SIGMOD interational conference on management of data (SIGMOD'01)*, pp. 235–246, 2001.

[62] R. A. Hankins and J. M. Patel, *Effect of node size on the performance of cache-conscious B^+-tree*, in *Proc. of the 2003 ACM SIGMETRICS interational conference on measurement and modeling of computer systems (SIGMETRICS'03)*, pp. 283–294, 2003.

[63] L. Arge, *The buffer tree: a new technique for optimal I/O-algorithms*, in *Proc. of Workshop on Algorithms and Data Structures*, (Berlin), pp. 334–345, Springer-Verlag, 1995.

[64] L. Arge, *The buffer tree: a technique for designing batched external data structures*, *Algorithmica* **37** (2003), no. 1 1–24.

[65] S. Berkowits, *Pin - a dynamic binary instrumentation tool*, *https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool*
(2012).

[66] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, *PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory*, in *Proc. of the 43th international symposium on computer architecture (ISCA'16)*, June, 2016.

[67] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, *A case for intelligent ram*, *Micro, IEEE* **17** (Mar, 1997) 34–44.

[68] D. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. McKenzie, *Computational ram: Implementing processors in memory*, *IEEE Des. Test* **16** (Jan., 1999) 32–41.

[69] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, *The architecture of the DIVA processing-in-memory chip*, in *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pp. 14–25, 2002.

[70] A. De, M. Gokhale, R. Gupta, and S. Swanson, *Minerva: Accelerating data analysis in next-generation ssds*, in *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pp. 9–16, 2013.

[71] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, *Sqrl: Hardware accelerator for collecting software data structures*, in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pp. 475–476, 2014.

[72] B. Akin, F. Franchetti, and J. C. Hoe, *Data reorganization in memory using 3D-stacked DRAM*, in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 131–143, 2015.

[73] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, *A scalable processing-in-memory accelerator for parallel graph processing*, in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pp. 105–117, 2015.

[74] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, *Top-pim: Throughput-oriented programmable processing in memory*, in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pp. 85–98, 2014.

[75] S. H. Pugsley, J. Jestes, H. Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li, *Ndc: Analyzing the impact of 3d-stacked memory+ logic devices on mapreduce workloads*, in *International Symposium on Performance Analysis of Systems and Software*, 2014.

[76] M. Hu, H. Li, Q. Wu, and G. Rose, *Hardware realization of bsb recall function using memristor crossbar arrays*, in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 498–503, June, 2012.

[77] B. Li, Y. Shan, M. Hu, Y. Wang, Y. Chen, and H. Yang, *Memristor-based approximated computation*, in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pp. 242–247, Sept, 2013.

[78] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, *Training and operation of an integrated neuromorphic network based on metal-oxide memristors*, *CoRR* **abs/1412.0611** (2014).

[79] P. Gu, B. Li, T. Tang, S. Yu, Y. Cao, Y. Wang, and H. Yang, *Technological exploration of rram crossbar array for matrix-vector multiplication*, in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pp. 106–111, Jan, 2015.

[80] Y. Kim, Y. Zhang, and P. Li, *A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing*, *J. Emerg. Technol. Comput. Syst.* **11** (Apr., 2015) 38:1–38:25.

[81] Z. Chen, B. Gao, Z. Zhou, P. Huang, H. Li, W. Ma, D. Zhu, L. Liu, H.-Y. Chen, X. Liu, and J. Kang, *Optimized learning scheme for grayscale image recognition in a rram based analog neuromorphic system*, in *Electron Devices Meeting, 2015. IEDM'15 Technical Digest. IEEE International*, Dec, 2015.

[82] G. Burr, P. Narayanan, R. Shelby, S. Sidler, I. Boybat, C. di Nolfo, and Y. Leblebici, *Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power)*, in *Electron Devices Meeting, 2015. IEDM'15 Technical Digest. IEEE International*, Dec, 2015.

[83] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, *Deep learning with cots hpc systems*, in *Proceedings of the 30th international conference on machine learning*, pp. 1337–1345, 2013.

[84] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, *Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning*, in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 269–284, 2014.

[85] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, *Dadiannao: A machine-learning supercomputer*, in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622, Dec, 2014.

[86] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, *A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm*, in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pp. 1–4, 2011.

[87] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, *Scalable processors in the billion-transistor era: Iram*, *Computer* **30** (1997), no. 9 75–78.

[88] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick, *Intelligent ram (iram): The industrial setting, applications, and architectures*, in *Computer Design, IEEE International Conference on*, pp. 2–7, 1997.

[89] D. Burger, *System-level implications of processor-memory integration*, in *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[90] M. Gokhale, B. Holmes, and K. Iobst, *Processing in memory: The terasys massively parallel pim array*, *Computer* **28** (1995), no. 4 23–31.

[91] T. Yamauchi, L. Hammond, and K. Olukotun, *A single chip multiprocessor integrated with DRAM*, in *Workshop on Mixing Logic and DRAM, held at the 24th International Symposium on Computer Architecture*, 1997.

[92] M. Oskin, F. T. Chong, and T. Sherwood, *Active pages: a computation model for intelligent memory*, in *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, pp. 192–203, 1998.

[93] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, *Near-data processing: Insights from a micro-46 workshop*, *Micro, IEEE* **34** (July, 2014) 36–42.

[94] R. Nair, S. Antao, C. Bertolli, P. Bose, J. Brunheroto, T. Chen, C. Cher, C. Costa, J. Doi, C. Evangelinos, B. Fleischer, T. Fox, D. Gallo, L. Grinberg, J. Gunnels, A. Jacob, P. Jacob, H. Jacobson, T. Karkhanis, C. Kim, J. Moreno, J. O'Brien, M. Ohmacht, Y. Park, D. Prener, B. Rosenburg, K. Ryu, O. Sallenave, M. Serrano, P. Siegl, K. Sugavanam, and Z. Sura, *Active memory cube: A processing-in-memory architecture for exascale systems*, *IBM Journal of Research and Development* **59** (March, 2015) 17:1–17:14.

[95] Z. Guz, M. Awasthi, V. Balakrishnan, M. Ghosh, A. Shayesteh, and T. Suri, *Real-time analytics as the killer application for processing-in-memory*, in *WoNDP: 2nd Workshop on Near-Data Processing, International Symposium on Microarchitecture*, 2014.

[96] N. S. Mirzadeh, O. Kocberber, B. Falsafi, and B. Grot, *Sort vs. hash join revisited for near-memory execution*, in *Fifth Workshop on Architectures and Systems for Big Data (ASBD), International Symposium on Computer Architecture*, 2015.

[97] J. Jeddeloh and B. Keeth, *Hybrid memory cube new DRAM architecture increases density and performance*, in *VLSI Technology (VLSIT), 2012 Symposium on*, pp. 87–88, June, 2012.

[98] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong, *A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (hbm) stacked DRAM with effective microbump i/o test methods using 29nm process and tsv*, in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pp. 432–433, Feb, 2014.

[99] F. Alibart, T. Sherwood, and D. Strukov, *Hybrid cmos/nanodevice circuits for high throughput pattern matching applications*, in *Adaptive Hardware and Systems (AHS), 2011 NASA/ESA Conference on*, pp. 279–286, June, 2011.

[100] Q. Guo, X. Guo, Y. Bai, and E. İpek, *A resistive tcam accelerator for data-intensive computing*, in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 339–350, 2011.

[101] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, *Ac-dimm: Associative computing with stt-mram*, in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 189–200, 2013.

[102] S. K. Esser, A. Andreopoulos, R. Appuswamy, P. Datta, D. Barch, A. Amir, J. Arthur, A. Cassidy, M. Flickner, P. Merolla, S. Chandra, N. Basilico, S. Carpin, T. Zimmerman, F. Zee, R. Alvarez-Icaza, J. Kusnitz, T. Wong, W. Risk, E. McQuinn, T. Nayak, R. Singh, and D. Modha, *Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores*, in *The*

2013 International Joint Conference on Neural Networks (IJCNN), pp. 1–10, Aug, 2013.

[103] J. Seo, B. Brezzo, Y. Liu, B. Parker, S. Esser, R. Montoye, B. Rajendran, J. Tierno, L. Chang, D. Modha, and D. Friedman, *A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons*, in *Custom Integrated Circuits Conference (CICC), 2011 IEEE*, pp. 1–4, 2011.

[104] M. Jung, J. Shalf, and M. Kandemir, *Design of a large-scale storage-class RRAM system*, in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pp. 103–114, 2013.

[105] C. Xu, P.-Y. Chen, D. Niu, Y. Zheng, S. Yu, and Y. Xie, *Architecting 3D vertical resistive memory for next-generation storage systems*, in *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pp. 55–62, 2014.

[106] T. M. Taha, R. Hasan, C. Yakopcic, and M. R. McLean, *Exploring the design space of specialized multicore neural processors*, in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, Aug, 2013.

[107] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, *Flexible, high performance convolutional neural networks for image classification*, in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pp. 1237–1242, 2011.

[108] J. Schmidhuber, *Multi-column deep neural networks for image classification*, in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR '12, pp. 3642–3649, 2012.

[109] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng, *Deep learning with cots hpc systems.*, in *ICML (3)*, vol. 28 of *JMLR Proceedings*, pp. 1337–1345, 2013.

[110] S. Sahin, Y. Becerikli, and S. Yazici, *Neural network implementation in hardware using fpgas*, in *Neural Information Processing*, vol. 4234 of *Lecture Notes in Computer Science*, pp. 1105–1112. Springer Berlin Heidelberg, 2006.

[111] C. Farabet, C. Poulet, J. Han, and Y. LeCun, *Cnp: An fpga-based processor for convolutional networks*, in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 32–37, Aug, 2009.

[112] J.-Y. Kim, M. Kim, S. Lee, J. Oh, K. Kim, and H.-J. Yoo, *A 201.4 gops 496 mw real-time multi-object recognition processor with bio-inspired neural perception engine*, *Solid-State Circuits, IEEE Journal of* **45** (Jan, 2010) 32–45.

[113] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in *Proceedings of the International Conference on Learning Representations (ICLR)*, pp. 1–14, May, 2015.

[114] F. Alibart, E. Zamanidoost, and D. B. Strukov, *Pattern classification by memristive crossbar circuits using ex situ and in situ training*, *Nature communications* **4** (2013).

[115] M. Hu, H. Li, Y. Chen, Q. Wu, and G. S. Rose, *BSB training scheme implementation on memristor-based circuit*, in *Computational Intelligence for Security and Defense Applications (CISDA), 2013 IEEE Symposium on*, pp. 80–87, 2013.

[116] B. Li, Y. Wang, Y. Wang, Y. Chen, and H. Yang, *Training itself: Mixed-signal training acceleration for memristor-based neural network*, in *ASP-DAC*, pp. 361–366, 2014.

[117] B. Liu, M. Hu, H. Li, Z.-H. Mao, Y. Chen, T. Huang, and W. Zhang, *Digital-assisted noise-eliminating training for memristor crossbar-based analog neuromorphic computing engine*, in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–6, 2013.

[118] B. Liu, H. Li, Y. Chen, X. Li, T. Huang, Q. Wu, and M. Barnell, *Reduction and IR-drop compensations techniques for reliable neuromorphic computing systems*, in *Computer-Aided Design (ICCAD), 2014 IEEE/ACM International Conference on*, pp. 63–70, Nov, 2014.

[119] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov, *Training and operation of an integrated neuromorphic network based on metal-oxide memristors*, *Nature* (2014).

[120] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, *Operating system support for improving data locality on cc-numa compute servers*, in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pp. 279–289, 1996.

[121] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, *Page placement strategies for GPUs within heterogeneous memory systems*, in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pp. 607–618, 2015.

[122] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, *Proceedings of the IEEE* **86** (Nov, 1998) 2278–2324.

[123] K. Chen, S. Li, N. Muralimanohar, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, *Cacti-3dd: Architecture-level modeling for 3D die-stacked DRAM main memory*, in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 33–38, 2012.

[124] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, *Cacti-io: Cacti with off-chip power-area-timing models*, in *Proceedings of the International Conference on Computer-Aided Design*, pp. 294–301, 2012.

[125] L. Gao, F. Alibart, and D. B. Strukov, *A high resolution nonvolatile analog memory ionic devices*, in *4th Annual Non-Volatile Memories Workshop, NVMW 2013*, 2013.

[126] C. Xu, D. Niu, N. Muralimanohar, N. P. Jouppi, and Y. Xie, *Understanding the trade-offs in multi-level cell ReRAM memory design*, in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pp. 1–6, 2013.

[127] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, *Rram-based analog approximate computing*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **34** (Dec, 2015) 1905–1917.

[128] J. Li, C.-I. Wu, S. Lewis, J. Morrish, T.-Y. Wang, R. Jordan, T. Maffitt, M. Breitwisch, A. Schrott, R. Cheek, H.-L. Lung, and C. Lam, *A novel reconfigurable sensing scheme for variable level storage in phase change memory*, in *Memory Workshop (IMW), 2011 3rd IEEE International*, pp. 1–4, IEEE, 2011.