UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Enabling Non-Volatile Memory for Data-intensive Applications

Permalink
https://escholarship.org/uc/item/9gr0q47b

Author
Liu, Xiao

Publication Date
2021

Peer reviewed|Thesis/dissertation

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

UNIVERSITY OF CALIFORNIA SAN DIEGO

Enabling Non-Volatile Memory for Data-intensive Applications

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

in

Computer Science (Computer Engineering)

by

Xiao Liu

Committee in charge:

Professor Jishen Zhao, Chair
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Tajana Rosing
Professor Steve Swanson

2021

Copyright
Xiao Liu, 2021
All rights reserved.

The dissertation of Xiao Liu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

To my parents.

EPIGRAPH

Oh, let the sun beat down upon my face.

With stars to fill my dreams.

I am a traveler of both time and space.

To be where I have been.

Sit with elders of the gentle race.

This world has seldom seen.

Talk of days for which they sit and wait.

All will be revealed.

— Robert A. Plant

TABLE OF CONTENTS

Dissertation Approval Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables xii

Acknowledgements xiii

Vita xvi

Abstract of the Dissertation xvii

Chapter 1 Introduction 1
1.1 Thesis Statement 3
1.1.1 Enabling NVM for Big Data Applications 3
1.1.2 Enabling NVM for Neural Network Applications 4
1.2 Thesis Outline 6

Chapter 2 Background and Motivation 8
2.1 The Hardware of Non-Volatile Memory 8
2.2 Data-intensive Applications 10
2.2.1 Big Data Applications 10
2.2.2 Neural Network Applications 12
2.3 NVM DIMM and Persistent Memory 13
2.3.1 Motivation: Lack of Reliability Support in NVM DIMM . . 15
2.4 RRAM-based NN Accelerators 16
2.4.1 Motivation: Missing Features Demanded by the Neural Network Applications 19

Chapter 3 Persistent Memory Workload Characterization: A Hardware Perspective . 21
3.1 Background and Motivation 23
3.1.1 Background on Persistent Memory Systems 23
3.1.2 The Need for Understanding Hardware Behavior 24
3.1.3 Row Buffer in Memory Devices and Its Impact on NVM Emulation 25
3.2 Methodology 26

3.2.1 System Platforms 27
3.2.2 File Systems 27
3.2.3 Persistent Memory Workloads 29
3.3 The Impact of Write Intensity 31
3.3.1 Impact of Write Intensity on System Performance 31
3.3.2 Bonnie Results 34
3.4 The Impact of NVM Access Locality 35
3.4.1 Buffer Hit Rate 35
3.4.2 Buffer Size 37
3.5 Microarchitecture Analysis 38
3.6 Implications on Persistent Memory System Design 40
3.7 Conclusion 42

Chapter 4 Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability 44
4.1 Background and Motivation 47
4.1.1 DRAM Main Memory 48
4.1.2 3D DRAM Last Level Cache 49
4.1.3 NVM Main Memory 49
4.1.4 Issues with Decoupled Reliability 50
4.2 Binary Star Design 52
4.2.1 Coordinated Reliability Scheme 53
4.2.2 Coordinating Wear Leveling and Consistent Cache Writeback 57
4.2.3 3D DRAM LLC Error Correction and Error Recovery . . . 59
4.2.4 Putting it All Together: An Example 60
4.3 Implementation 61
4.3.1 3D DRAM LLC Modification 62
4.3.2 Memory Controller Modifications 62
4.3.3 System Software Modifications 65
4.3.4 Binary Star Overheads 66
4.4 Experimental Methodology 67
4.5 Results 70
4.5.1 Reliability 70
4.5.2 Performance 72
4.5.3 Impact of the Periodic Forced Writeback Interval 74
4.6 Conclusion 77

Chapter 5 HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing 78
5.1 Background and Motivation 80
5.1.1 Neural Network Data Mapping on RRAM crossbar 80
5.1.2 Thermal Issues in RRAM Accelerator 81
5.2 HR^{3}AM Design 83

5.2.1 HR^{3}AM Overview 83
5.2.2 Heat-resilient Weight Adjustment 84
5.2.3 Dynamic Thermal Management 88
5.3 Evaluation 90
5.3.1 Methodology 90
5.3.2 Inference accuracy 91
5.3.3 Thermal status 92
5.3.4 Performance and energy 93
5.4 Conclusion 94

Chapter 6 Mirage: A Highly Paralleled And Flexible RRAM Accelerator 96
6.1 Background & Motivation 98
6.1.1 Shared Data Induced Data Dependency 100
6.1.2 Issues with Existing Hardware Assignments 102
6.2 Mirage 104
6.2.1 Finer-grained Parallel RRAM Architecture (FPRA) 105
6.2.2 Auto Assignment 110
6.3 Mirage Implementation 113
6.3.1 The Implementation of FPRA 113
6.3.2 Auto Assignment Tool Implementation 115
6.4 Experimental Setup 116
6.5 Evaluations 118
6.5.1 Performance 118
6.5.2 Flexibility 120
6.5.3 Power Efficiency 123
6.6 Conclusion 124

Chapter 7 Conclusion 125
7.1 Summary of the Thesis 125
7.2 Future Works 126

Bibliography 127

LIST OF FIGURES

Figure 2.1: Overview of RRAM-based DNN accelerator. 17

Figure 3.1: System platform configuration. 26
Figure 3.2: Throughput of FFSB on four systems. The x-axis represents the read to write ratio. The y-axis represents the throughput. 31
Figure 3.3: Normalized results of Bonnie. The y-axis stands for all the six metrics. The x-axis stands for the normalized number of each metric. 34
Figure 3.4: Filebench throughputs under various memory models. The y-axis represents the throughput. The x-axis represents memory models, which include DRAM, classic NVM with no buffer, and NVM with buffer hit rates under 50% to 90%. 36
Figure 3.5: Filebench throughputs of DRAM and NVMs with different row buffer sizes. The workload has 60% sequential read operations. 37
Figure 3.6: Filebench throughputs of DRAM and NVMs with different row buffer sizes. The workload has 80% sequential read operations. 37
Figure 3.7: Correlation between workload performance and five hardware counter statistics. The x-axis represents different metrics. The y-axis represents the correlation coefficient between the throughput and the metric. 38

Figure 4.1: Reliability and throughput of various LLC and main memory hierarchy configurations normalized to 28nm DRAM with RECC. We define "reliability" as the reciprocal of device failures in time (FIT) [1, 2], i.e., 1/FIT. Reliability data is based on a recent study [1]. 45
Figure 4.2: (a) Multiple reliability mechanisms for a cache line across LLC and main memory. (b) Fraction of clean LLC lines during application execution. We collected data for ten seconds after the benchmark finishes the warm up stage. 50
Figure 4.3: Basic architecture configuration. 53
Figure 4.4: Binary Star overview. (a) Periodic forced writeback and (b) Consistent cache writeback. (c) Binary Star error correction and error recovery mechanisms. A solid filled square represents a checkpointed consistent data block. A striped filled square represents a remapped inconsistent data block. 54
Figure 4.5: An example of running an application on Binary Star. 60
Figure 4.6: Modifications to NVM wear leveling. 63
Figure 4.7: State machine of the Binary Star software daemon. 65
Figure 4.8: Device failure rates of 3D DRAM for traditional RECC [3], RECC with IECC [1], Binary Star, and Chipkill with IECC [1, 4] vs. single cell bit error rate. Vertical lines represent 3D DRAM technology nodes (28nm and sub-20nm).[1] 71
Figure 4.9: System performance of different memory configurations with no ECC and Binary Star. We normalize the throughput of each configuration to the throughput of 3D DRAM LLC with DRAM. 73

Figure 4.10: Performance of Binary Star normalized to the performance of the 3D DRAM +DRAM configuration, as PCM latency is varied as a multiple of its value in Table 4.1. 73
Figure 4.11: Effect of varying the periodic forced writeback interval on (a) throughput, (b) probability of rollbacks (i.e., error rate on a dirty line), and (c) NVM writes. 75

Figure 5.1: Temperature impact on (a) RRAM cell conductance and (b) CNN applications relative inference accuracy. 81
Figure 5.2: Temperature distribution of a single RRAM chip when running VGG16, InceptionV3 and ResNet50. 82
Figure 5.3: Overview of HR^{3}AM structure. 83
Figure 5.4: Bitwidth downgrading hardware in the crossbar array. 86
Figure 5.5: Bitwidth downgrading operation flowchart. 87
Figure 5.6: Tile pairing (a) mechanism and (b) control logic. 88
Figure 5.7: Inference accuracy of four neural network models. 89
Figure 5.8: Comparison of thermal distributions of a RRAM chip when running three networks. 92
Figure 5.9: Comparison of maximum and average temperatures of RRAM chip when running three networks. The *x*-axis represents the temperature measurement time in milliseconds. 92
Figure 5.10: Normalized (a) throughput and (b) power consumption on HR^{3}AM. 94

Figure 6.1: The throughput of four neural networks. Numbers are normalized to the theoretical maximum throughput of the accelerator. 99
Figure 6.2: Shared data in a convolutional layer with four duplicate kernels. 100
Figure 6.3: An example of data sharing in the first two layers of a DNN model. 101
Figure 6.4: Single inference latency breakdown. 102
Figure 6.5: Comparison between user assignment and adjusted assignment on (a) the number of duplicate kernels in eight convolutional layers, and (b) inference speedup. 103
Figure 6.6: The overview of the Mirage architecture. 105
Figure 6.7: Two kernel batching types: (a) linear batching, and (b) square batching. . . 106
Figure 6.8: Data sharing aware memory in two layers of a DNN. 108
Figure 6.9: Batched window shifts in the input/output feature maps of a convolutional layer. The layer uses two batched 2×2×64-sized kernels with one stride size. Two batched kernels use a shared input of a 2×3×64-sized window. The numbers indicate the steps and the red arrow represents the direction. . . . 109
Figure 6.10: The work flow of the auto assignment. 110
Figure 6.11: The flow chart of searching algorithm. 112
Figure 6.12: Tile architecture. 114
Figure 6.13: State diagram of the intra-tile data synchronization. 115
Figure 6.14: Speedup of single inference latency with four RRAM-based accelerator configurations. 119

Figure 6.15: Idle ratio of four configurations. 119
Figure 6.16: Comparison of throughputs of three configurations. 120
Figure 6.17: Comparison of Mirage and w/ kernel duplication over (a) speedup, and (b) throughput when the chip size scales from 1× to 32×. 122
Figure 6.18: Speedup of three batching strategies with Mirage. 122
Figure 6.19: Power efficiency of three configurations. 123

LIST OF TABLES

Table 3.1: Profiled software and hardware events. (S) represents software events. The rest are hardware counter events. 28
Table 3.2: Storage workloads descriptions. 29

Table 4.1: Evaluated processor and memory configurations. 69
Table 4.2: Comparison of the evaluated resilience schemes. 72

Table 6.1: Hardware configuration of the Mirage. 118
Table 6.2: Number of duplicate kernels for each layer, and hardware utilization in four VGG configurations. 121

ACKNOWLEDGEMENTS

My PhD career is a life-changing experience. I am delighted and honored to meet with wonderful people through this adventure.

I would like thank my parents, Daping Liu and Yanmin Liu, for their unconditional love, compassion, and support. My parents have always been backing me through all the challenges during the pursuit of the degree.

I am deeply indebted to my advisor, Professor Jishen Zhao, for her generous support and endless patience as the chair of my committee. Her guidance has taught me not only what good research would be but also what a good researcher would be. Her invaluable mentorship reshapes me from an incompetent novice to be a qualified researcher.

I would also like to extend my gratitude to Professor Ryan Kastner, Professor Tajana Rosing, Professor Steve Swanson, and Professor Farinaz Koushanfar for their valuable comments and feedbacks as my committee members. Special thanks to Professor Tajana Rosing and Professor Steve Swanson. Aside from serving the committee, they provide strong support through collaborations with the projects.

I would like to express my appreciation to all the close collaborators: Professor Rachata Ausavarungnirun, Minxuan Zhou, Dr. Shengye Wang, Dr. David Roberts, Dr. Brandon Nesterenko, Professor Onur Mutlu, Professor Qing Yi, Professor Vijaykrishnan Narayanan, Sean Eilert, Ameen Akel, Pankaj Mehra, and Bhaskar Jupudi. Special thanks to Professor Rachata Ausavarungnirun for providing tremendous assistance through multiple of my publications. Thanks to Minxuan Zhou for introduction and discussion over the PIM-related topics. Thanks to Dr. Shengye Wang for teaching me over the robotics-related and software engineering topics. Thanks to Dr. David Roberts for teaching me over the memory errors related topcis. Thanks to Dr. Brandon Nesterenko for introduction on the HPC related topics. I would also like to thank my mentors, Dr. Maya Gokhale and Dr. Erich Haratsch, and colleagues during my summer internships.

I would like to thank all members of the STABLE for their feedback and support of my research. Special thanks to Matheus Ogleari and Mengjie Li, who provide enormous help in my first year at UC Santa Cruz. Thanks to Zixuan Wang and Theodore Michailidis for close collaboration over the NVM-related topics. I would also like to thank members of the NVSL, Dr. Jian Yang, Dr. Lu Zhang, Dr. Kunal Korgaonkar, Dr. Amirsaman Memaripour, Professor Joseph Izraelevitz, Morteza Hoseinzadeh, and Juno Kim, for many insightful discussion over the NVM-related topics. I also wish to thank other faculties of the CSE department: Professor Bryan Chin, Professor Hadi Esmaeilzadeh, and Professor Yiying Zhang. It was a pleasant experience to teach students and discuss researches with them.

Last but not least, I want to thank everyone, including all the mentioned people, who provide assistance and accommodation in the past two years during the pandemic. I never expect my PhD career would encounter such catastrophe. I feel lucky and deeply thankful to accomplish the degree at this special time.

Chapter 3 contains material from "Persistent Memory Workload Characterization: A Hardware Perspective.", by Xiao Liu, Bhaskar Jupudi, Pankaj Mehra, and Jishen Zhao, which appears in the Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC 2019). The dissertation author is the primary investigator and first author of this paper.

Chapter 4 contains material from "Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability.", by Xiao Liu, David Roberts, Rachata Ausavarungnirun, Onur Mutlu, and Jishen Zhao, which appears in the Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 52). The dissertation author is the primary investigator and first author of this paper.

Chapter 5 contains material from "HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing.", by Xiao Liu, Minxuan Zhou, Tajana S. Rosing, and Jishen Zhao, which appears in the Proceedings of the 2019 IEEE/ACM International Symposium on Low

Power Electronics and Design (ISLPED 2019). The dissertation author is the primary investigator and first author of this paper.

Chapter 2 and Chapter 6 contain material from "Mirage: A Highly Paralleled And Flexible RRAM Accelerator.", by Xiao Liu, Minxuan Zhou, Rachata Ausavarungnirun, Sean Eilert, Ameen Akel, Tajana S. Rosing, Vijaykrishnan Narayanan, and Jishen Zhao, which is submitted for publication. The dissertation author is the primary investigator and first author of this paper.

VITA

2015	Bachelor of Engineering, Zhejiang University
2021	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Xiao Liu, David Roberts, Rachata Ausavarungnirun, Onur Mutlu, and Jishen Zhao. ”Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability.” In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 807-820. 2019.

Xiao Liu, Minxuan Zhou, Tajana S. Rosing, and Jishen Zhao. ”HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing.” In 2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), pp. 1-6. IEEE, 2019.

Xiao Liu, Bhaskar Jupudi, Pankaj Mehra, and Jishen Zhao. ”Persistent Memory Workload Characterization: A Hardware Perspective.” In 2019 IEEE International Symposium on Workload Characterization (IISWC), pp. 249-252. IEEE, 2019.

ABSTRACT OF THE DISSERTATION

Enabling Non-Volatile Memory for Data-intensive Applications

by

Xiao Liu

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2021

Professor Jishen Zhao, Chair

The emerging Non-Volatile Memory (NVM) technologies are reforming the computer architecture. NVM holds advantages includes a byte-addressable interface, low latency, high capacity, and in-memory computing capability. However, data-intensive applications today demand compound features rather than just better performance. For instance, big data applications would require high availability and reliability. The neural network applications require scalability and power efficiency. Despite all the advantages of NVM, simply attaching the NVM to the memory hierarchy are unable to meet these demands. The decoupled reliability schemes among NVM and other devices fail to provide sufficient reliability. The vulnerability against overheating and hardware underutilization limit the performance and scalability of the in-memory computing

NVM.

Using the NVM for the data-intensive application requires redesign and customization. In this thesis, we focus on discussing the architecture designs that enable NVM for data-intensive applications. Our study includes two major types of data-intensive applications – big data applications and neural network applications.

We first conduct a characteristic study against the persistent memory applications. Persistent memory implements over the NVM-based main memory and guarantees crash consistency. We explore the performance interaction across applications, persistent memory system software, and hardware components. Based on our characterization results, we provide a set of implications and recommendations for optimizing persistent memory designs.

Second, we propose Binary Star for the generic data-intensive applications, which coordinates the reliability schemes and consistent cache writeback between 3D-stacked DRAM last-level cache and NVM main memory to maintain the reliability of the memory hierarchy. Binary Star significantly reduces the performance and storage overhead of consistent cache writeback by coordinating it with NVM wear leveling.

For neural network applications, our first design explores the thermal effect over one representative NVM – resistive memory (RRAM). We find heat-induced interference decreases the computational accuracy in the RRAM-based neural network accelerator. We propose HR^{3}AM, a heat resilience design, which improves accuracy and optimizes the thermal distribution. Results show that HR^{3}AM improves classification accuracy and decreases both the maximum and average chip temperatures.

Lastly, we present Mirage to improve parallelism and flexibility for pipeline-enabled RRAM-based accelerators. Mirage is a hardware/software co-design that addresses the data dependencies and inflexibility issues of existing accelerators. Our evaluation shows that Mirage achieves low inference latency and high throughput compared to state-of-the-art RRAM-based accelerators.

Chapter 1

Introduction

The rapid growth of chip technology has innovated a variety of new applications over the last decade. Data-intensive applications are almost everywhere: it covers the cloud backend of mobile app services to the self-driving control of the autonomous vehicles. Many organizations and individuals fulfill their jobs and render their services through data-intensive application over the cloud-native hardware. The fast response, great scalability, and easy maintenance of the modern applications have improved the productivity of various tasks.

Data-intensive applications demands reliability, scalability, and availability for the computing system. Frist, because of the value of the data and importance of the service, users demands high reliability for the data-intensive applications [5]. Such reliability include crash consistency towards power failure, error correction against memory related errors, fast reboot against software generated errors and undoing unintentional manipulation. Second, data-intensive application also demands high scalability. As many services have millions of active the users, the load size of the data-intensive applications can be massive. Scalability requires the application to provide same Quality of service (QoS) to users under different load sizes. The QoS includes critical performance metrics such as response time, tail latency, and maximum bandwidth. The Last but not least, data-intensive application demands high availability. A available data-intensive application

benefits both users and the administrator. The available application should be convenient and usable for the user. The less expertise application required from the user, the wider range users can use the application, the more availability it can provide for the users. For the administrator perspective, a available application should simple enough and easy for them to maintenance, upgrades and evaluation [6].

The emerging Non-Volatile Memory (NVM) could address these demands. Compared to DRAM and SSD, NVM has its unique performance characteristics. The byte-addressable feature and low access latency enable NVM to be an alternative for the main memory. The non-volatility provided by the NVM makes the memory-level crash consistency possible, which is also known as "persistent memory". These performance characteristics are vastly different from either fast, low capacity, and volatile SRAM and DRAM, or the slow, high capacity, and non-volatile SSD and disk. The nature of NVM leads to different reliability characteristics. Unlike existing storage media, NVMs use two states of non-silicon-based material to store a binary digit. Because of its features, the NVM is free of many silicon-based errors on the DRAM and SSD. With better reliability, NVM is also projected to scale below 10nm easier than the DRAM [7]. NVM also provides in-memory computing ability. The in-memory computing NVM is able to conduct computations for the neural network. The in-memory computing provides massive parallelism utilizing capacity of the NVM. It also reduces the data movement between memory and processors and achieves better energy efficiency.

With the arrival of 3DXpoint [8], the wide use NVM could be expected in many computational systems within a few years. The system level supports for NVM are fairly sufficient. There are extension to the instruction sets [9], file systems [10, 11, 12, 13, 14], programming models [15, 16, 17, 18], compiler [19, 20], and etc. With these designs, the deployment of NVM can be as convenient as the the deployment of DRAM and SSD.

Despite all the advantages of NVM, enabling NVM for the data-intensive applications is non-trivial. The data-intensive application requires resilience against system crash and errors,

which are not supported with plain NVM. The persistent memory application demands memory-supported crash consistency. To support that, it requires the modification to the memory controller to support writeback reordering and forced writeback, and software support on operating system and programming models. Generic big data applications require strong resilience against memory errors. When attaching NVM to the current system, it requires coordination among the reliability schemes among NVM and other type of devices to achieve high reliability. The in-memory computing NVM enabled neural network applications also requires customized designs to achieve high performance and reliability. The solutions in traditional memory hierarchy are not applicable to the in-memory architecture. This novel architecture also suffers from issues in the reliability (e.g., thermal-related and endurance-related) and performance (e.g., underutilization), which requires re-evalution of the issues and correlated design.

1.1 Thesis Statement

Based on our analysis towards characteristics of the NVM and demands of the data-intensive applications, our thesis statement is: **utilizing the novel NVM, the customized architecture designs can provide desired features with sufficient performance, for various modern applications.** In this thesis, we discuss two major types of data-intensive applications: big data applications and neural network applications. Due to the uniqueness of each type of applications, there is not an unified NVM-based design that benefits both of them. However, all the designs of the thesis follow the same principle: the design is aware of the characteristics of the NVM, and redesigns the existing architecture such that the system benefits from the NVM.

1.1.1 Enabling NVM for Big Data Applications

Because of the similarity to the DRAM, applications can use NVM as a part of the main memory to cache data to the lower level storage. We first look into the crash consistency

applications – persistent memory applications. Persistent memory application requires data consistency against system failure at anytime. We first conduct characteristics study against the persistent memory applications. In this study, we analyze the performance and hardware behaviors of persistent memory systems by running various storage workloads with a variety of configurations (e.g., read/write intensity, file size, and number of threads) on traditional and persistent memory file systems. To examine the performance impact of persistent memory systems and workload configurations, we profile the statistics of a variety of software events and hardware performance counters available in modern processors. As a result, we find the implications and recommendations that might benefit persistent memory designs.

Then we consider more general applications – big data applications. These applications require large amount of memory but also desire resilience against errors in the memory system. We propose Binary Star, a coordinated memory hierarchy reliability scheme with NVM-based main memory for these applications. Binary Star consists of 1) coordinated reliability mechanisms between 3D DRAM LLC and NVM, and 2) NVM wear leveling with consistent cache writeback. The key insight of our design is that traditional memory hierarchy designs typically strive to *separately* optimize the reliability of LLCs and main memory with *decoupled* reliability schemes, yet coordinating reliability schemes across the LLC and main memory enhances the reliability of the *overall* memory hierarchy, while at the same time reducing unnecessary ECC, multiversioning, and ordering control overheads that degrade performance. We show that Binary Star provides benefits across various types of workloads, including in-memory databases, in-memory key-value stores, online transaction processing, data mining, scientific computation, image processing, and video encoding.

1.1.2 Enabling NVM for Neural Network Applications

Applications can also use NVM as the media for the novel PIM architecture. The PIM architecture avoids unnecessary data movement between CPU and memory, and parallels

operations in the memory, which is a good fit the neural network applications. Under the PIM architecture, NVM is used as the computational media. We specifically look into the resistive RAM (RRAM) based PIM architecture. RRAM-based PIM has been widely used in accelerating neural network (NN) applications.

We first look into the thermal issue of RRAM-based accelerators. We discover that the thermal issue potentially reduces the inference accuracy of neural networks, such that the accuracy loss can be as high as 90%. We introduce a novel heat resilient design HR^3AM for the RRAM crossbar. HR^3AM solves the issue from two aspects: adapting weights to the reduced conductance and optimizing chip thermal distribution. Therefore we correspondingly propose two mechanisms in HR^3AM from these two aspects. The first mechanism *bitwidth downgrading* aims to improve accuracy. It adapts weights to the decreased conductance range by reducing bitwidth per RRAM cell. The second mechanism *tile pairing* aims to tackle the thermal issue. It optimizes the chip thermal status by matching hot spot tiles with idle tiles.

Lastly, we look into the underutilization in the RRAM-based accelerators. We discovered the data dependency induced pipeline stalls, and inflexible hardware assignment in pipeline design of the RRAM-based accelerators. These two issues lead the underutilization of the accelerator, which can causes up to 50% idle time during the execution. To resolve these issues, we introduce Mirage, a RRAM-based accelerator that provides 1) high parallelism by resolving the data dependency due to shared data in duplicate kernels, and 2) high flexibility by enabling automatic hardware assignment for general DNN applications. Mirage consists of two primary designs: Finer-grained Parallel RRAM Architecture (FPRA) and Auto Assignment (AA), to deal with issues raised by the shared data and hardware assignment, respectively. FPRA is based on the insight that the proper arrangement of the duplicate kernels and unified data buffering of input and output data can resolve the data dependency caused by shared data. FPRA introduces *kernel-batching* to group the computation of the duplicate kernels of the same layer, and *data sharing aware memory* to uniformly manage the input and output data of duplicate kernels. AA

utilizes the network parameters and accelerator hardware configurations to automatically generate a hardware assignment. AA follows two rules that we observed from the DNN models: 1) layers at the front need to have same or higher level of parallelism of the layers deeper in the network , and 2) consecutive layers with same level of parallelism suffer from the least amount of stall brought by data dependency. With the greedy strategy that follows these rules, AA searches for the best assignment for all the layers under the available hardware.

1.2 Thesis Outline

The thesis is organized as follows: Chapter 2 discusses the background of NVM, modern applications, and exiting architecture designs for NVM. Chapter 3 presents our study over characteristics of the persistent memory. Chapter 4 introduces the design and evaluation of the Binary Star. Binary Star is a coordinated memory hierarchy reliability scheme built on the NVM-based heterogeneous memory. Chapter 5 presents the design and evaluation of the HR^{3}AM. HR^{3}AM is heat resilient design for the RRAM-based accelerator. Chapter 6 introduces the design and evaluation of the Mirage. Mirage is a software/hardware co-design that provides high parallelism and high flexibility for the RRAM-based accelerator. Chapter 7 summarizes the thesis and discusses the potential future research inspired by this thesis.

Acknowledgements

This chapter contains material from “Persistent Memory Workload Characterization: A Hardware Perspective.”, by Xiao Liu, Bhaskar Jupudi, Pankaj Mehra, and Jishen Zhao, which appears in the Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC 2019). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability.”, by Xiao Liu, David Roberts, Rachata Ausavarungnirun, Onur Mutlu, and Jishen Zhao, which appears in the Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 52). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing.”, by Xiao Liu, Minxuan Zhou, Tajana S. Rosing, and Jishen Zhao, which appears in the Proceedings of the 2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED 2019). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Mirage: A Highly Paralleled And Flexible RRAM Accelerator.”, by Xiao Liu, Minxuan Zhou, Rachata Ausavarungnirun, Sean Eilert, Ameen Akel, Tajana S. Rosing, Vijaykrishnan Narayanan, and Jishen Zhao, which is submitted for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 2

Background and Motivation

In this chapter, we discuss the background and motivation of the dissertation. Section 2.1 provides an overview of the hardware technologies of the NVM. Section 2.2 provides an introduction of the data-intensive applications. Section 2.3 discusses the NVM DIMM and persistent memory – a crash consistent architecture based on NVM. Section 2.4 introduces the architecture of the RRAM based NN accelerator, which uses NVM as the in-memory computing media.

2.1 The Hardware of Non-Volatile Memory

Three representative NVMs are phase change memory (PCM), spin-transfer torque magnetic memory (STT-RAM), and resistive memory (RRAM). Despite that different types of NVMs bases on different underlying physics, all the NVMs use high resistance state and low resistance state and use electrical stimulus to switch between the two states [21].

Typical PCM uses $Ge_2Sb_2Te_5$ (GST) chalcogenide, which obtains the crystalline state and the amorphous state. The crystalline state has a low resistance, while the amorphous state has a high resistance. To set a PCM storage element as the amorphous state, the controller uses a short current pulse to heat the chalcogenide to 650°C and freezes rapidly [22]. To set the element to the crystalline state, the controller uses a long current pulse to heat the chalcogenide in the

same way but cools down slowly [22]. PCM is the most popular NVM technology because of the advantages on scalability, cell size, read latency, and write power efficiency [23, 7]. The industry has actively engaged in the development of PCM and proposed many prototypes [24, 25, 26]. It is believed that the 3D-Xpoint is a type of the PCM [27].

STT-RAM uses magnetic tunnel junction. The structure of STT-RAM has two layers: a free layer and a reference layer [28]. The reference layer has a fixed magnetic orientation, while the free layer's magnetic orientation decides the resistance of the unit. When both layers have the same orientation, the unit is at low resistance. When they have opposite orientations, the unit is at high resistance. When manipulating the resistance, the transistor release a large current to alternate the magnetic orientation of the free layer [29]. STT-RAM has one of the best read/write latency and endurance among NVMs. However, large cell sizes limit the capacity of STT-RAM [21]. These features make STT-RAM a desired replacement to the SRAM or embedded DRAM [30, 31].

The RRAM, or memristor, shares some similarities with PCM. RRAM uses two metal layers and an oxide layer in between. The state of RRAM depends on the formation and rupture of the oxide layer. The state of RRAM can be changed by an external voltage. A positive voltage change sets the RRAM as the high resistance, and a negative voltage sets the RRAM as the low resistance [32]. Similar to the PCM, RRAM also has a low access latency, small cell size, and low write energy [21].

All three types of NVMs are able to form one transistor one resistor (1T1R) structure, where the transistor is silicon and the resistor is NVM cell. The 1T1R structure can scale and form an array. Similar to the DRAM array, the controller uses the word line to select a line from NVM array. Two vertical lines, source line and bit line, are used for read and write operations [21]. The 1T1R array forms the basic structure of the NVM chip, which can be integrated into an NVM DIMM (details in Section 2.3).

PCM and RRAM can also compose the “crossbar” architecture, which can be used for

in-memory computing. The crossbar architecture uses the one selector and one resistor (1S1R) structure. Because the 1S1R structure is smaller than the 1T1R structure, the crossbar array is able to achieve higher density [33]. In-memory computing architecture commonly uses the structure of the crossbar array. We provide details of the in-memory computing NVM architecture in Section 2.4.

2.2 Data-intensive Applications

The data-intensive applications arise with the rapid growth in cloud computing today. The data-intensive applications cover a wide range of scenarios: backend of web and mobile services, shared services platforms, machine learning applications, IoT applications, etc. The data-intensive applications intensively evolve in transferring, generation, and processing the data. The data-intensive applications are the essential part of many core services. Data-intensive applications require strong scalability, reliability, and availability [34]. The scalability guarantees the services are accessible to millions of users around the world. The reliability guarantees that any threat to the services (e.g., cyber-attack, power loss, and system errors) would not harm the availability of the service. The availability provides usability to the users and maintainability to the administrators. In this thesis, we focus on discussing the two most popular applications: big data applications, and neural network applications.

2.2.1 Big Data Applications

Big data applications are the applications that are frequently accessing and manipulating a large volume of data. Typical big data applications are data mining, cloud computing, and streaming applications [35, 36]. Big data applications store the data with databases and file systems and use caches and buffer to accelerate data access. The big data application processes the data with the batched or streaming form. The key factors of big data applications are the

complexity and size of the data. Therefore, the memory hierarchy is decisive to the performance of the data-intensive applications.

Big data applications are in demand for larger memory capacity [37]. With the rapid growth on the scale of the applications, the system needs to keep increasing the capacity of the memory to ensure the performance and quality of the service of the application. We have seen the exponent growth in the amount of the data during the past two decades [38, 35] Applications, such as cloud computing and scientific computing, demand frequent access to terabytes or even petabytes of data [39]. Attaching more block-level devices can satisfy the data's capacity but cannot provide the same level of performance to the application. The big data applications demand the scalable storage hierarchy: the capacity, latency, and bandwidth on all levels of the storage hierarchy need to match the data sizes.

In addition to the memory capacity, big data applications are also in demand for reliability against errors. As the main memory-related error dominates the computing system [2, 40, 41], these errors could potentially corrupt data, interrupt or completely halt the processing, and invalid the generated results. For these applications, NVM can certainly provide much a larger main memory capacity compared to the slow scaling DRAM. Nevertheless, providing reliability with low overhead would require a better understanding of architecture.

Persistent memory applications. Certain big data applications require strong consistency of data, such that it has to satisfy the ACID (Atomicity, Consistency, Isolation, Durability) features. The NVM provided memory-level crash consistency that guarantees the ACID, also known as "persistent memory". Persistent memory applications with strong consistency are able to survive system crashes at any time. The persistent memory applications can be implemented with persistent memory libraries [42]. Persistent memory applications can also be block devices and databases applications but applied to persistent memory file system [43, 44, 45, 46, 47]. Due to the fast speed of NVM, the persistent memory applications obtain crash consistency with less overhead compared to the applications that use lower level storage. Compared to regular applica-

tions, the persistent memory applications frequently accessing the memory and use fence and flush instructions to force ordering of the data writeback. We provide a discussion of persistent memory designs in Section 2.3.

2.2.2 Neural Network Applications

The deep neural network (DNN) has made significant progress in prediction accuracy over the last decade. The evolution of DNN broadens the use of neural network to artificial intelligence, pattern recognition, image classification, and natural language processing. The DNN also booms the research in the various area of self-driving, face recognition, recommendation system, image processing, and more.

One preventative DNN is the convolutional neural networks (CNN). CNNs are widely used in image and pattern recognition. When CNN is used for image recognition, the input is the pixels of the image on all the channels. The output is a vector that the value on each dimension represents the probabilistic values of the input identified as the object represented by that dimension. The intermediate data between layers during the processing is also referred to as the *feature map*. CNN uses four major types of layers: convolutional layer, pooling layer, ReLU (rectified linear unit) layer, and fully connected layer.

The convolutional layers are the majority of the layers in the CNN. The convolutional layer uses a set of kernels, which are small matrices, to conduct convolution with the input feature map [48]. Three hyperparameters decide how to process the entire image. The *depth* decides the number of neurons in the depth dimension that are active. The *stride* decides how many pixels kernel moves at a time. The *padding* decides whether the padding should be zero and the spatial size of the padding. The computation of the convolutional layer is essentially multiplication and summation, which take up most computations in CNN processing.

The *pooling layer* is the layer that conducts non-linear downsampling. The purpose of using pooling layer is to reduce the size of the feature map, such that the amount of data and

computation is reduced. Therefore, the pooling layer controls the overfitting of the network model. Maxpooling is commonly used in CNN, which generates the highest value from the input. The typical pooling layer is behind the convolutional layer, and in front of the ReLU layer.

The *ReLU layer* uses the activation function to remove all the negative values. Typical ReLU layer uses tanh and sigmoid functions. The ReLU layer helps linearize the behavior of the model, which is easier to be optimized [49]. Because the ReLU layer prevents vanishing gradients problem, it is critical to the learning process of the DNN.

The *fully connected layers* in the CNN are the same as fully usually connect to the very end of the CNN. In the fully connected layers, neurons are connected to all the activations of the previous layer. The computation of activation in the fully connected layer is matrix multiplication and addition to the bias offset.

2.3 NVM DIMM and Persistent Memory

Deploying NVM over the DIMM is a common use of the NVM. With NVM DIMM, NVM can be accessed through the memory bus and form hybrid or heterogeneous memory with the DRAM. The high bandwidth and low latency of the memory bus match up with the NVM performance. JEDEC released NVDIMM-P standard recently, which supports persistent memory and interconnections with DDR4 or DDR5 DRAM [50]. Intel's Optane DC Persistent Memory Module (referred to as 3D-Xpoint DIMM) is an NVM DIMM based on 3D-Xpoint technology. The 3D-Xpoint DIMM can operated under two modes: *Memory mode* and *App Direct mode*. The memory mode uses 3D-Xpoint DIMM as the main memory and DRAM DIMM on the same channel as the cache to the 3D-Xpoint DIMM. The app direct mode allows bypass the DRAM DIMM and direct access to the 3D-Xpoint.

Recent studies provide a complete characterization against the 3D-Xpoint DIMM performance [51, 52, 53]. A single 3D-Xpoint DIMM is able to provide 2.3 GB/s read bandwidth

and 6.6 GB/s write bandwidth at maximum. The read latency of 3D-Xpoint DIMM ranges from 169 ns to 305 ns. The latency of flush instructions ranges from 62 ns to 90 ns. The actual write latency is immeasurable due to the existence of an internal buffer [53].

The reliability of the NVM DIMM is promising. Memory errors can be classified into soft errors and hard errors. The soft errors dominate the memory errors on many systems [2]. Most NVM technologies experience fewer soft errors than the DRAM [54, 55]. The hard errors in NVM are mostly related to the limited endurance. The life cycle of NVM at least $10^4\times$ longer than the NAND SSD [22, 56]. Products include 3D-Xpoint DIMM have equipped wear-leveling and over-provisioning to combat hard errors [51].

While the NVM provides non-volatility, "persistent memory" provides data persistence. Due to the volatile cache, persistent memory design guarantees the data to reach the NVM in the correct order. Various architectural designs have been proposed to support persistent memory in academia [31, 57, 58, 59]. The Intel extends the x86 and adds *clwb* and fence instructions to support persistent memory [9]. Intel recently supports Asynchronous DRAM Refresh (ADR) feature in the recent Xeon processor [60]. The ADR guarantees that all the data reaches 3D-Xpoint DIMM will be flushed into NVM during the power loss.

Two mainstream persistent memory system support solutions are the programming libraries and file systems. Even though the users can directly apply flush and fence instructions, the engineering effort to transform a non-persistent memory application to persistent memory application is painstaking. Both programming library and file system ease the use of persistent memory. The various persistent memory programming libraries proposed [15, 61, 62, 63, 18]. When using the library, users declare a memory space as the persistent region similar to the standard memory allocation. The libraries manage and allocate the pool of persistent memory space. The libraries also conduct garbage collections to avoid memory space leakage and fragmentation. The user can access the persistent region with provided persistent access API or atomic section to specify memory accesses to be persistent. When using the libraries, all the cache flushes and memory

barriers are added to the persistent memory accesses of the application automatically. To ensure data consistency, the libraries generally maintain a log that keeps track of all the modifications to the persistent region. Each time system restarts, the system uses the log to verify all the ongoing modifications to the region and guarantees the data is not corrupted.

The file system supported persistent memory requires less programming effort than the libraries [64, 11, 65, 12, 13, 66, 14]. Because the traditional file system is designed for non-volatile block devices, many features such as journaling already guarantee data persistence. The persistent memory file system has three major differences compared to the block device file system. First, persistent memory file system supports direct access (DAX), which the NVM access to bypass the page cache. The performance gap between DRAM and NVM is much less compared to the gap between NVM and SSD. Bypassing the page cache brings a significant increase in the performance [67]. Second, persistent memory file system allows access to NVM through load and store with memory-mapped (mmap) DAX mode. When the NVM is mapped to the memory space under DAX mode, applications can direct access NVM with load and store, and bypass all the system caches. This feature can benefit certain applications when all the overhead of system caches are removed. Third, the persistent memory uses 8-byte atomicity. This finer granularity provides more flexibility and higher performance for the garbage collection.

2.3.1 Motivation: Lack of Reliability Support in NVM DIMM

Many features of the NVM DIMM fit with the demands of big data applications. The non-volatility and reliability provide resilience against errors. The capacity and performance provides scalability for increased data volume. However, we find two desired research opportunities with the current reliability supports for big data applications.

First, there lacks profiling of persistent memory applications from the hardware perspective. Existing studies of persistent memory applications profiling are from the software perspective, which only provides insights based on operations throughput, the latency of software

requests, and counters of software events [68, 69, 51]. The profiling related to hardware components, including cache hierarchy, TLB, memory controllers, and buffers, are missing. Because persistent memory application uses forced cache writebacks and memory fences to reorder the data writeback, many behaviors of these hardware components is different from when system dealing with non-persistent memory applications. The hardware perspective-based profiling is critical to the performance of the persistent memory application.

Second, existing design can provide the capacity demanded by the big data application, but not the reliability. The lack of coordination between reliability schemes between NVM and other devices leads to compromised reliability. The existing reliability scheme, such as error correction code, only prevents the errors within each device. However, the memory errors increase significantly as the technology node scales down [1, 37]. To prevent an increased amount of errors, the overhead of reliability schemes is non-negligible [70, 71, 72]. The decoupled reliability schemes ignore the feature of the memory hierarchy that certain errors could be corrected with data copies on the other devices. As a result, such decoupled reliability schemes lead to performance overhead and compromised reliability for the big data application.

2.4 RRAM-based NN Accelerators

Several RRAM-based architectural designs for accelerating DNN have been proposed recently [73, 74, 75, 76, 77, 78]. The most significant difference between RRAM-based accelerator and generic computation units or CMOS-based DNN accelerators is that the RRAM-based accelerators conduct computation within RRAM cells using analog signals. RRAM-based accelerators use memristors to form a crossbar structure to perform multiply-accumulate operations within RRAM itself.

Accelerator architecture. As Figure 2.1 (a) shows, the memristors are placed in a grid such that they are horizontally connected by the *word lines* and vertically connected by the *bit lines*.

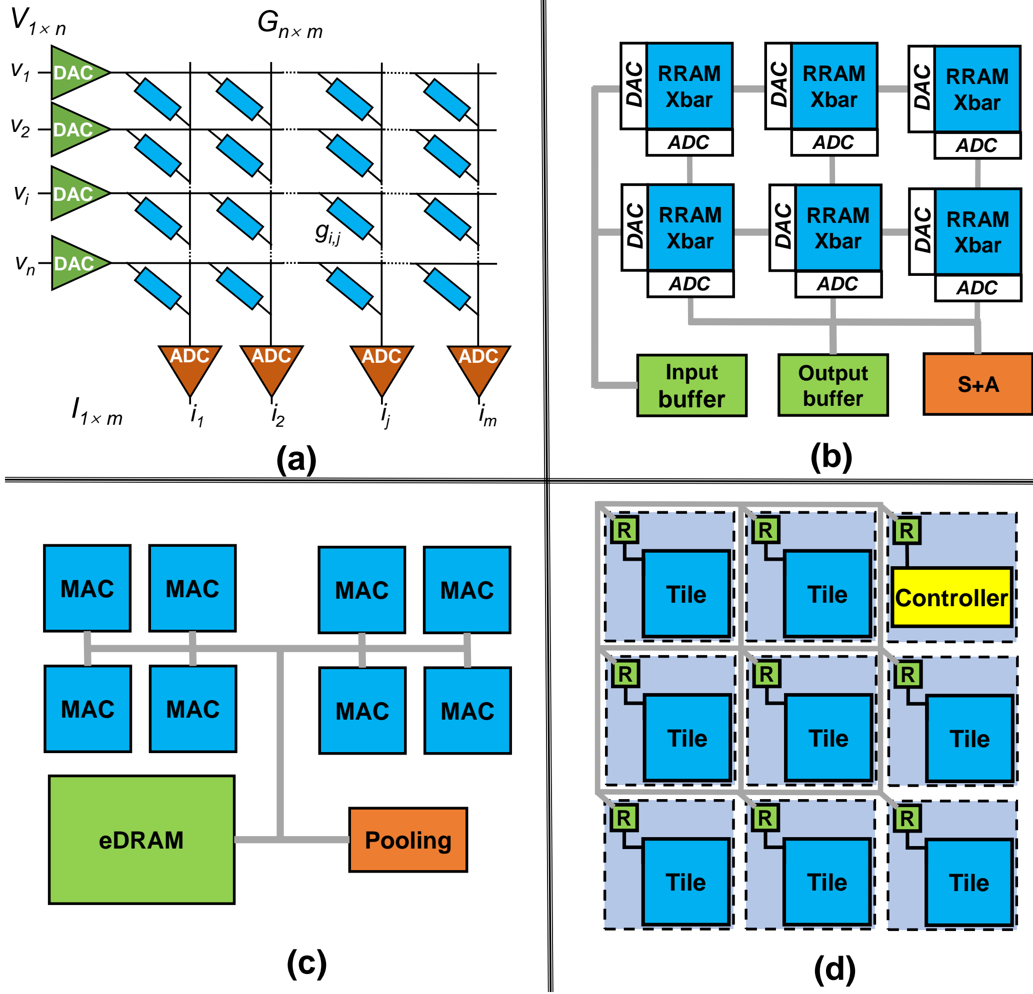


Figure 2.1: Overview of RRAM-based DNN accelerator.

For each memristor, the output current on the word line is the product of memristor conductance and input voltage on the bit line (Kirchoff's Law). The currents from all the memristors connected with the same word line are summed up. Hence, the input voltages vector $V_{1\times n}$ is able to conduct matrix multiplication with the memristors conductance matrix $G_{n\times m}$ to get the output currents vector $I_{1\times m}$, i.e., $I_{1\times m} = V_{1\times n} \times G_{n\times m}$. The size of an RRAM crossbar array is limited by the device yield, device variability, and array parasitics [79]. The size of RRAM crossbar array can vary from 12×12 [80] to 500×661 [81].

Crossbar arrays require auxiliary hardware components to perform matrix multiplication (as shown in Figure 2.1 (b)). First, it requires digital to analog converters (DACs) and analog to

digital converters (ADCs) because the calculation is in analog form. The analog signal is only used within crossbar arrays, while the digital signal is used everywhere else. Second, data buffers are also required to store input and output data temporarily. Typically, input buffers and output buffers are separated. Third, the shift-and-add (S+A) unit is required to combine the results of multiple crossbar arrays. Since a single memristor can only represent a finite number of bits, the matrix multiplication of high precision numbers (e.g., 32-bit integer) must spread its high and low bits across several crossbar arrays [74]. The shift-and-add (S+A) unit shifts the output according to the bits location of the input, and adds it to the outputs generated by other crossbar arrays. Designs such as [77] propose to conduct analog summation to reduce the overhead from the intensive A/D conversions. These crossbar arrays, along with shared components, form a *multiply accumulator* (MAC).

To process a neural network, a MAC has additional components to deal with non-matrix-multiplication operations in the network (as shown in Figure 2.1 (c)). First, a pooling unit is required to perform pooling operations for the pooling layers. Second, activation units are required to perform activation functions for the activation layers. Such units include sigmoid, ReLU, tanh, etc., are used based on the type of the neural networks. Thrid, embedded DRAM (eDRAM) stores and transfers data for the MACs. A buffer connects all the MACs and the data cache for data forwarding. All the MACs, including the pooling unit, activation units and eDRAM, form a *tile*.

As shown in Figure 2.1 (d), a tile is the highest-level computational component of an RRAM-based accelerator. Tiles are interconnected with routers to form a 2D mesh-connected network-on-chip (NoC). Tiles use routers to forward input and output data to each other. The chip controller is in charge of pre-processing and I/O operations, which connects to the NoC through the router.

Inference using RRAM-based accelerators. When processing a DNN inference, the crossbar arrays process the convolutional and fully connected layers while the pooling units

process the pooling layers [75, 74]. The RRAM-based accelerator processes the convolutional layer in the *weight stationary* form [82] by assigning weights of the kernel as the conductance of the crossbar array (i.e., $G_{n\times m}$ on Figure 2.1 (a)). The three-dimensional kernel flattens to a single dimension which fits into a column of the crossbar array. The output on the bit line directly corresponds to the output of the convolution. Weights of a kernel can be stored over the different columns of a crossbar array. Considering the limitations on the size of the crossbar array and the bitwidth of RRAM cells, all the weights of the same kernel are usually spread across multiple crossbar arrays (i.e., a column of crossbar arrays shown in Figure 2.1 (b)).

When performing a convolution in the weight stationary form, the accelerator feeds input from the feature map (i.e., $V_{1\times n}$ in Figure 2.1 (a)) using the size of the kernel and stride size while keeps the kernel weights in the fixed crossbar arrays. When the kernel is mapped to several different arrays, the final outputs are summed using the intermediate outputs generated by these arrays (i.e., S+A on the Figure 2.1 (b)).

2.4.1 Motivation: Missing Features Demanded by the Neural Network Applications

Despite the advantages of the RRAM-based NN accelerator, there are two major missing demands with the existing architecture.

First, existing RRAM hardware lacks resilience against interference. The hardware of RRAM faces interference from thermal, crosstalk, and write disturbance [21, 83, 84]. These interferences could impact the accuracy of the crossbar multiplication. However, there lacks a quantification of the impact on the accuracy of the neural networks from interferences during the runtime of RRAM-based neural network. Therefore, the corresponding countermeasures have not been proposed as well.

Second, the pipeline design in the existing RRAM architecture is incomplete. For instance, the data forwarding of how feature map data is shared and distributed among kernels belong to

the same layer is unidentified. The data forwarding is critical and can cause pipeline idles when the computation and data transfer are incompatible. The usability of the existing accelerator is also missing. There lacks an approach to assignment the hardware to each layer in the network. For the regular users who have no knowledge of the architecture details, handling the hardware assignment is impractical and can lead to underutilization of the hardware.

Acknowledgments

This chapter contains material from "Mirage: A Highly Paralleled And Flexible RRAM Accelerator.", by Xiao Liu, Minxuan Zhou, Rachata Ausavarungnirun, Sean Eilert, Ameen Akel, Tajana S. Rosing, Vijaykrishnan Narayanan, and Jishen Zhao, which is submitted for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 3

Persistent Memory Workload Characterization: A Hardware Perspective

Deploying byte-addressable NVMs on memory bus enables a new data storage concept, called "persistent memory" [85]. Representative NVMs include spin-transfer torque RAM [86], phase-change memory [87], resistive RAM [88], battery-backed DRAM [89, 90], and 3D XPoint™memory that is announced by Intel and Micron [8, 91]) – They combine the benefits of both worlds: the byte-addressability and fast load/store interface of memory with the data recoverability of storage. As such, NVM allows applications to directly load and store data in the main memory. Due to this game-changing feature, NVMs are poised to radically improve data manipulation performance and energy efficiency. NVMs also motivates a new concept – persistent memory, which guarantees that data is recoverable (consistent and durable) through power outages and system crashes; persisting data through slow storage I/O interface is no longer required.

Traditional computer systems manage data persistence by software mechanisms of file systems and databases [92, 93]. Yet main memory access is manipulated by both virtual memory software and hardware control units, such as CPU caches, buffers, and memory controllers. As

such, to fully exploit the potential of persistent memory, it is beneficial to design system software and applications with exposure to hardware details.

However, most previous persistent memory systems are motivated and evaluated by software-based performance profiling and characterization [68, 69, 11, 67]. These profiling schemes provide system performance implications in terms of operation throughput, software request latency, and software events (e.g., context switches, system calls, and thread migrations). But they treat hardware components, such as CPU cache hierarchy, TLBs, memory controllers, and buffers, as a black box. As a result, very little information about hardware components is available for persistent memory software or software/hardware coordinated designs.

In this work, we intend to provide implications on optimizing persistent memory system designs from a hardware perspective. We analyze the performance and hardware behaviors of persistent memory systems by running various storage workloads with a variety of configurations (e.g., read/write intensity, file size, and number of threads) on traditional and persistent memory file systems. To examine the performance impact of various persistent memory system and workload configurations, we profile the statistics of a variety of software events and hardware performance counters available in modern processors. As a result, we find the following implications and recommendations that might benefit persistent memory software designs.

- We identify that persistent memory system performance is more closely correlated to the configurations and behaviors of several particular hardware components, such as last-level cache (LLC) access and instruction and data translation look-aside buffers (TLBs), than others. As such, persistent memory system designers may prioritize the optimizations on the access to the hardware components with larger correlation than others.
- We show that persistent memory system designs perform differently when running different types and configurations of applications on top. Performance varies significantly across read- and write-intensive workloads, due to the latency gap between DRAM and NVM devices, along with the asymmetrical read and write latency of NVMs.

- We show that the recently-introduced direct access (DAX) support in Linux essentially enhances file systems to the performance benefit of persistent memory. But the current design can be susceptible for the limitations on flexibility and performance offered to applications.
- Whereas NVM devices do not require row buffers (a set of sense amplifiers and latches that temporarily store data values) as DRAM does, buffers, if present on NVM chips, can significantly mitigate the performance penalty of long latency, low bandwidth, and asymmetric read/write latency in NVM access. As such, system software and applications with high memory access locality can exploit such buffers to optimize system performance.
- We demonstrate that data atomicity support, such as journaling, imposes considerable performance overhead even with the latest persistent memory file system optimization mechanisms. To further optimize performance, persistent memory systems can adapt to the different levels of atomicity based on the application's requirement.

3.1 Background and Motivation

3.1.1 Background on Persistent Memory Systems

Modern computer systems decouple fast data access and data persistence by adopting separate main memory (DRAM) and storage (disks and flash) components. This decades-long model is being enriched by a new tier of memory, "persistent memory", which offers a single-level, flat data storage model by incorporating the persistence property of storage and the fast load/store access of memory [94, 95, 96, 97]. Persistent memory can be implemented by attaching byte-addressable NVMs – such as spin-transfer torque RAM [98], phase-change memory [87], resistive RAM [88], battery-backed DRAM [89], and 3D XPoint™memory [8] – directly to the processor-memory bus. As the appearance of NVMs on the market in 2019 [91], persistent memory can radically change the landscape of software and hardware design in the future computer systems.

As a hybrid of memory and storage, persistent memory systems need to maintain data

persistence (crash consistency) [85] – just like storage systems – to ensure that data is recoverable in the the face of impending system crashes or power outages. Data persistence support can be implemented at various memory system components, such as libraries [61, 99], in-memory data structures [100, 101], operating systems [95], file systems [64, 67, 102, 11, 103, 104], database systems [105, 106], and microarchitectures [31, 107, 108, 57, 109, 110].

Among these, persistent memory file systems are promising solutions due to their ease of use by application programmers and legacy application codes [64, 67, 11, 103, 104]. With minor or no modifications, various applications can directly execute on top of persistent memory file systems without being aware of the difference between persistent memory and traditional disk-based storage devices. To offer optimal application performance, persistent memory file systems need to be carefully designed to fully exploit the characteristics of memory and storage systems incorporated with NVM devices [68].

3.1.2 The Need for Understanding Hardware Behavior

Most recent persistent memory file system designs are motivated and evaluated by pure software performance profiling and characterizations [11, 104, 61]. The profiling can provide many insights in the system operation. But it remains unclear how well the system designs along with the applications running can utilize the underlying hardware components in computer systems. For instance, traditional disk-based file system designs focus on optimizing the data transfers between main memory and storage components, mainly managed by software mechanisms. However, by attaching to the processor-memory bus, persistent memory systems need to maintain data transfers between CPU caches and main memory; such most components in this cache-memory hierarchy are managed by hardware. Memory access heavily relies on various hardware components, such as CPU caching, buffering, address translation, memory controllers and interfaces, and memory device architecture. Without understanding the behavior of these components, persistent memory file system designs can fall sub-optimal. In this work, we expose

a large room for future persistent memory design, once we take into account the detailed hardware behaviors.

3.1.3 Row Buffer in Memory Devices and Its Impact on NVM Emulation

Row Buffer in Memory Devices. Memory (either DRAM or NVM) is physically organized as two-dimensional arrays (rows and columns) of bitcells. With DRAM, each read or write access requires buffering of an entire row in the array in a row buffer structure (a set of sense amplifiers and latches that temporarily store data values). The typical size of a DRAM row buffer can be up to 8KB [111]. Accessing row buffer has a much shorter latency than memory array, whereas a miss in the row buffer needs to be serviced by multiple steps: writing the buffered row back to memory array, reading the new row to be accessed from the memory array into the row buffer, and then perform the access. As a result, DRAM accesses with high locality – consecutive accesses to the same row (i.e., hit in the row buffer with "open" row policy [112]) – can lead to much better access performance than those miss in the row buffer. Based on our industrial collaborator's exploration, NVM devices may not require row buffers due to their discrepant cell device and sensing mechanisms compared to DRAM. But several memory microarchitecture studies indicate that small buffers in NVM (e.g., 4Kb) – where multiple adjacent columns share one sense amplifier by a multiplexer – can significantly improve memory access performance [113, 114], given that NVMs typically impose much longer access latency and lower bandwidth than DRAM. As such, we investigate the performance impact of buffers on NVM devices in persistent memory systems and demonstrate the substantial benefits of employing them.

Impact on NVM Emulation. As NVM devices are yet to be available on the market. Most previous persistent memory studies emulate NVM using DRAM [94, 115]. However, the impact of the buffers on NVM devices is not fully considered. For example, PMEP [94] collects an average memory access latency – which is a mix of row-buffer hits and misses – and then

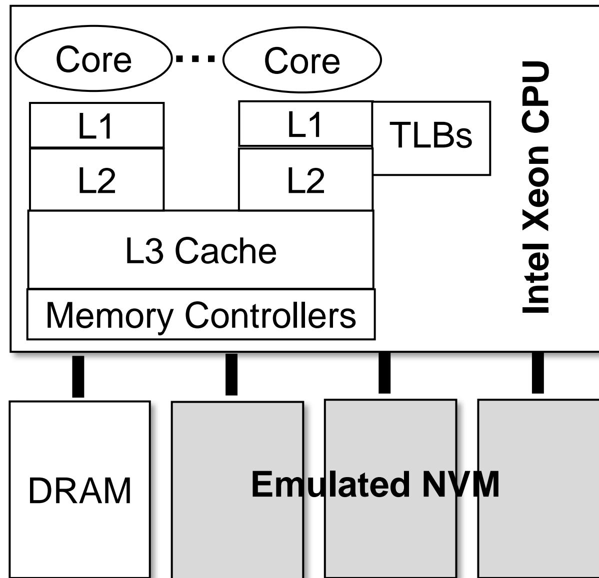


Figure 3.1: System platform configuration.

multiplies it by a ratio of NVM/DRAM latency [94]. Yet hits in the buffers on NVM chips can effectively close the latency gap between DRAM and NVM access. Naïvely treating all persistent memory access with NVM latency penalty can guide persistent memory file system designers to consider NVM's long read and write latency in an over-pessimistic manner. To examine such impact, we take into account the impact of buffer in NVM devices in our studies. In particular, we distinguish the memory access latency of buffer hits and misses, and then only apply the NVM long read/write latency penalty to those buffer misses.

3.2 Methodology

To investigate representative file systems and storage workloads, we chose one popular traditional file system, two latest persistent memory file system implementations, and exercised them with six widely-used storage benchmarks running on a workstation configured with an emulated NVM.

3.2.1 System Platforms

To perform our measurements, we use a workstation configured with Intel Xeon E5-2620 v3 processor (Figure 3.1). The processor contains six 2.4GHz two-way multithreaded cores. Each core has the private 32KB 8-way set associative L1 data and instruction caches, and a private 256KB 8-way set associative L2 caches. All the cores share a 15MB 20-way set associative last-level cache (LLC). To access hardware performance counters, we use Linux `perf` utility [116] to profile various hardware counter (and also software events) as listed in Table 3.1. We run each workload multiple times, calculated the mean and the standard deviation of each software and hardware counter event.

The workstation runs Ubuntu 14.04.6 operating system LTS distribution; we upgrade the kernel to 4.4.0 version. The workstation has four DDR4-2133 RDIMMs. We emulate 12GB of NVM on DRAM with various read and write latencies[1]. To configure the NVM partition, we use `memmap` GRUB (boot loader) parameter to mark the specific 12GB range of DRAM as the persistent memory partition mounted as `pmem0`. We further ensured that other operating system functions do not use the persistent memory partition. We update the new boot loader configuration and recompiled the kernel by enabling the parameters that support DAX and NV-DIMM.

3.2.2 File Systems

Our evaluation includes three file systems. Two of them are variants of ext4, and the last is a state-of-the-art persistent memory file system[2].

Ext4. Ext4 [93] is a popular file system used in Linux. Ext4 is an extent-based file system with reduced metadata overhead. Ext4 employs journaling to guarantee the persistence of both data and metadata updates. It supports three modes, which provides different levels of

[1]We also employed 128GB memory to study some of our workloads. However, the shape of the results is consistent with the two different memory sizes. Therefore, we only show our results with 12GB emulated NVM.

[2]We do not show analysis of Intel persistent memory file system (PMFS), because it is no longer being maintained [10].

Table 3.1: Profiled software and hardware events. (S) represents software events. The rest are hardware counter events.

Cache Events	L1 data cache loads, L1 data cache load misses, L1 data cache stores, L1 data cache store misses, L1 data cache prefetches, L1 data cache prefetch misses, L1 instruction cache loads, L1 instruction cache load misses, L1 instruction cache prefetches, L1 instruction cache prefetch misses, LLC loads, LLC load misses, LLC stores, LLC store misses, LLC prefetch misses
TLB Events	dTLB loads, dTLB load misses, dTLB stores, dTLB store misses, dTLB prefetches, dTLB prefetch misses, iTLB loads, iTLB load misses
Other Events	Page faults (S), context switches (S), CPU migrations (S), branch loads, branch load misses

atomicity for data and metadata. 1) `Data-writeback` mode does not perform any data journaling. This mode is intended for optimizing performance, and it fails to recover the data across system crashes. 2) `Data-ordered` mode only records metadata in the journal. This mode provides decent performance (lower than data-writeback mode), yet it has the advantages to replay the journal and check for any inconsistencies. 3) `Data-journal` mode writes both metadata and data into the journal. This mode ensures data to be recoverable by taking notable performance degradation.

Ext4-DAX. Latest persistent memory file systems can bypass the page cache allocated in DRAM and directly access NVM via loads/stores by using a technique called *Direct Access* (DAX) [90]. DAX maps storage components directly into userspace. DAX handles page faults as exceptions and directly copies the pages from NVM to DRAM main memory. Currently, several traditional file systems, including XFS, ext4, and Btrfs, can be configured to use DAX mode. This work focuses on ext4. We leave the investigation of the other two file systems (XFS and Btrfs) as future work. Note that ext4-DAX uses journaling to persist metadata updates only; it does not support file content journaling. To ensure a fair comparison between ext4 and ext4-DAX, we configure both in `data-ordered` mode, where only metadata is journaled.

Table 3.2: Storage workloads descriptions.

Name	Thread Count	File Count	Total Size (GB)	Behavior
Fileserver (Filebench)	50	10k	5.7	Emulates the basic file access pattern.
Webproxy (Filebench)	100	10k	10	Emulates a webproxy server.
Varmail (Filebench)	16	1000	10	Emulates the access pattern of a simple mail server.
File compression & decompression.	1	2	4.6	Conducts zip and unzip operations over a folder.
FFSB	20	110	11	Emulates multi-thread application access to many small size files under the same folder.
Bonnie	1	1	5	Performs a pre-defined set of tests on a single file.

NOVA. NOn-Volatile memory Accelerated log-structured file system (NOVA) is one of the latest persistent memory file systems [11]. It adopts three optimizations for persistent memory. First, NOVA is a log-structured file system (LFS), which naturally logs the data updates. Therefore, it only journals metadata to ensure data persistence. The journal process only records the directory's log tail pointer and new inode's valid bit. Second, NOVA provides optimizations for garbage collection. NOVA contains two types of garbage collections: fast and thorough garbage collections. Fast garbage collection reclaims a page if it is entirely dead. Thorough garbage collection reclaims two pages by transferring data to a new page , if they both have more than 50% spaces. Third, NOVA uses a red-black tree as the free list to improve allocation and deallocation speed. The tree provides effective merging and fast deallocation.

3.2.3 Persistent Memory Workloads

Table 3.2 includes all the workloads for the experiments. They includes three workloads from filebench suite [43], two microbenchmarks – FFSB [117] and Bonnie [45]. Before each

experiment, we exercise the DRAM and persistent memory partitions by writing, appending, and deleting files of various sizes. We also experiment with various file sizes in each workload.

Filebench. Filebench [43] is a benchmark suite designed for evaluating file systems. It can generate various workloads to emulate a variety of real-world applications. We generated three different workloads, include fileserver, webproxy, and varmail, to emulate the typical file system access patterns of the file, web proxy, and email servers.

- *Fileserver* emulates the basic file server access pattern, including a sequence of creates, deletes, appends, reads, writes, and metadata operations, which is performed by multiple "user" threads.
- *Webproxy* emulates a plain webproxy server. We employ 100 threads to perform a mix of create, append, read, and delete operations on 10,000 files with an average size of 1MB. This workload is characterized by fairly flat namespace hierarchy with a directory width of 1,000,000, i.e., all files are deployed in a large directory. The workload has 100 threads with a 5:1 read/write ratio.
- *Varmail* emulates the access pattern of a mail server. The operations of this workload include create-append-synchronization, read-append-synchronization, reads, and deletes. We configure a directory with a width of one million. It stores 1000 files with a median file size of 100MB. This workload has 16 threads. The read/write ratio is 1.

File Compression and Decompression. We employ `zip`, and `unzip` to investigate the performance of file compression and decompression in persistent memory systems. The zip operation compresses the archives, while the unzip operation extracts the archives. We use them to compress and decompress two MPEG-4 files; each has a size of 2.3GB.

FFSB. The Flexible File System Benchmark (FFSB) [117] generates customizable workloads to measure file system performance. It employs *pthreads* to support multiple groups of threads that can access multiple files simultaneously. We use it to generate a set of workloads that perform a mix of reads, writes, re-writes and appends operations. We set up various read/write ratios and random access pattern throughout the experiment. We employ 20 threads to perform

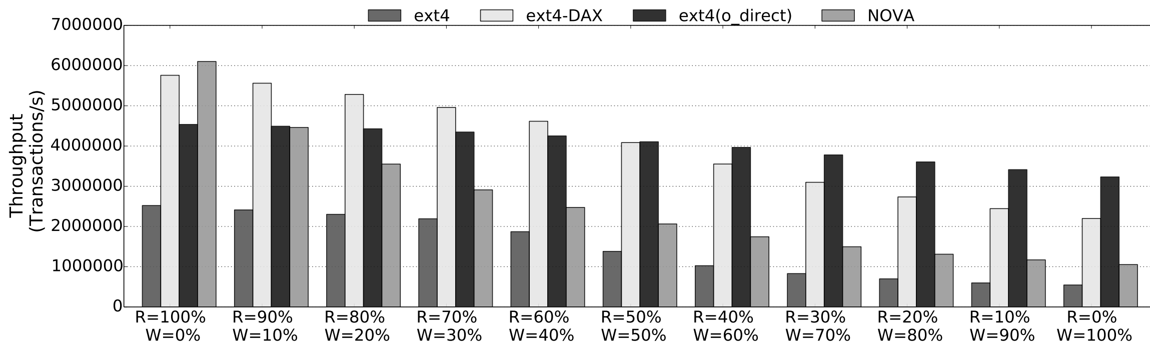


Figure 3.2: Throughput of FFSB on four systems. The *x*-axis represents the read to write ratio. The *y*-axis represents the throughput.

random read and write operations on 110 files, the size of a single file is 100MB.

Bonnie. Bonnie64 [45] measures file system performance by invoking `putc()` and `getc()` macro operations to file systems. In addition, it can create four child processes that execute 4000 seek operations to random locations in the file. 10% of the seek operations may change the block that they conduct read and re-write. Our experiments perform `putc()` and `getc()`, and random seek operations in a total size of 5GB files. `putc()` imitates write operation while `getc()` imitates read operation.

3.3 The Impact of Write Intensity

To investigate the impact of storage workloads' data access behavior on persistent memory performance, we evaluate the throughput of FFSB and Bonnie by varying their write intensity. The two workloads provide us with the flexibility of configuring read/write ratio during their execution.

3.3.1 Impact of Write Intensity on System Performance

Figure 3.2 shows transaction throughput of FFSB configured with various read/write ratios ranging from from 100% read (*R=100%, W=0*) to 100% writes (*R=0, W=100%*). Because

the throughput deviation is within 5% across multiple runs, we only show an average throughput in Figure 3.2. In addition to ext4, ext4-DAX, and NOVA, we also study ext4 with direct_IO reads and writes by enabling `o_direct` flag [118] (later represented by **ext4-o_direct**). O_direct has been introduced to Linux since kernel version 2.4.10. It attempts to minimize I/O caching and fits the situation when the application has self-caching [118]. Like the DAX, o_direct bypasses the page cache to exploit the byte-addressability of NVMs. However, the limitation of using o_direct flag with an `open` system call is that workloads must issue IO operations with the size as the multiples of 512 byte [118]. We make the following major observations from our results.

First, *the overall throughput decreases as write intensity increases.* For the write request, the file system takes several operations, includes reserve space, rearrange existing data, and commit new data. These operations typically take a longer time to process than the read request operations. As Figure 3.2 shows, the throughputs of all the file systems decrease as the write intensity increases. However, different file systems show a vastly different downtrend.

Second, *system throughput with NOVA decreases much faster than with the other file systems, when the write intensity increases*. NOVA performs the best with read-intensive workloads (2.1× throughput of ext4 and 1.5× throughput of ext4-DAX, when the workload only issues read requests). But ext4-o_direct performs the best with write-intensive workloads (in particular, NOVA only achieves 27% of ext4-o_direct throughput and 39% of ext4-DAX throughput, when the workload only issues write requests). As discussed in Section 3.2, ext4-DAX does not support data journaling mode. Therefore, we configure all ext4 variations with metadata journaling only; data is directly updated in place. Therefore, the principal cause of this drop is that NOVA supports atomicity of both data and metadata, whereas ext4 variants only ensure metadata atomicity.

NOVA handles a write request in the following way: 1) allocate new data blocks, 2) copy user data, 3) append to the log, 4) update the radix tree, 5) free old blocks by invoking the garbage collector. This results in a substantial performance drop of NOVA when compared to other file systems (which do an in-place update of the data) for write-intensive workloads. Moreover, due

to FFSB's preallocation, all the write operations in the FFSB benchmark are rewritten to blocks. As a result, NOVA is inferior to ext4-DAX for the write-intensive workloads.

To further investigate the journaling overhead, we experiment the write-intensive workload on ext4 under `data-journal` mode. This has the same level of write protection as NOVA. By doing it, throughput decreases to 23.6% of ext4 with metadata journaling. It is about 60% of NOVA throughput. Because ext4-DAX with data journaling mode is still under development, we assume that it will decrease similarly to ext4. Therefore, we estimate throughput by linear decreasing. Results show throughput of ext4-DAX with data journaling could be around 180% to 89% of NOVA's. Even under the same degree of write atomicity, ext4-DAX still has equivalent or better performance than NOVA on the write-intensive workload.

Third, *DAX feature provides considerable performance benefits for NVM file systems.* As Figure 3.2 shows, ext4-DAX performs 2$\times$ of ext4 across all the workload configurations. DAX bypasses the page cache, which is a buffer allocated in DRAM. Such buffering can improve the performance of slow block-based storage. Due to NVM's properties: no seek overhead and DRAM comparable latency, page cache is nothing but an unnecessary copy of the data in the hybrid memory system. To bypass the page caches, DAX directly maps the NVM device to the userspace. For read-intensive workloads, the performance of ext4-DAX is 3$\times$ of ext4 performance. While for write-intensive workloads, it is 1.3$\times$. Because we use DRAM as emulated NVM, DAX could show more benefits over the actual NVM device because of the longer write latency.

Finally, *the throughput trend of ext4-o_direct is the most stable.* Throughput difference between the 100% write configuration and the 100% read configuration is only 28%, while ext4, ext4-DAX, and NOVA have 78%, 62%, NOVA's 82.7%, respectively. Ext4-o_direct also performs the best among write-intensive configurations (write ratio above 50%). DAX is theoretically superior to o_direct, because of it is intentionally optimized for NVM. However, our results show the opposite way. A possible reason is, that workloads can achieve direct loads and stores to the storage when using `mmap` with DAX. Without mmap, DAX still requires a copy between memory

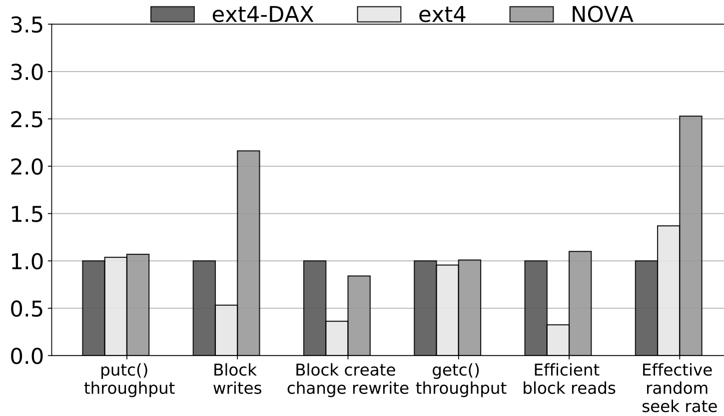


Figure 3.3: Normalized results of Bonnie. The *y*-axis stands for all the six metrics. The *x*-axis stands for the normalized number of each metric.

and *pmem0*. Currently, in the FFSB workload, there is no support to perform memory-mapped file IO. So we use the default setting, and the results show o_direct perform better. The combination of mmap and DAX could provide the most benefit for NVM file systems.

Furthermore, we use `fio`[44] benchmark that adopts mmap to investigate DAX and o_direct. However, the performance results of both modes are similar. We leave the further evaluation of DAX with mmap as the future work.

3.3.2 Bonnie Results

Figure 3.3 shows the results of Bonnie64 workload. We configure the write-to-read ratio as 1:1 throughout the experiments. We show the results of six metrics: `putc()` throughput, block writes, block create change rewrite, `getc()` throughput, efficient block reads, and effective random seek rate. All the metrics results are normalized to the results on the ext4-DAX mounted system.

We make two observations from the result. First, we can conclude that NOVA read fast because of the effective random seek rate, not the block read speed. NOVA has the best random seek rate, about 2.5× of the ext4 and 1.8× of the ext4-DAX. While the block read speeds between NOVA and ext4-DAX are similar. Second, NOVA has a faster block write speed while

ext4-DAX has better block create and rewrite speed. NOVA block write speed is 2.1× of the ext4-DAX's. While the ext4-DAX block create speed is 1.2× of the NOVA's. As we previously discussed, NOVA pays the data journaling overhead while ext4-DAX does not. Even though NOVA has a bigger advantage on write than ext4-DAX on the rewrite, Bonnie conducts rewrites more than writes and appends. Therefore, NOVA and ext4-DAX have similar `putc()` throughputs (difference within 7%).

3.4 The Impact of NVM Access Locality

In this section, we examine the performance impact of adopting hardware buffers in NVM components in persistent memory systems. As discussed in Section 3.1.3, hit on the row buffer could be a critical factor for the latency model. Row buffer is the cache for memory rows in DRAM. In the future NVM, there will be a similar structure. We name it "buffer" in this work. Like row buffer in the DRAM, the buffer acts as a cache to the NVM rows. The NVM's buffer could be different from the row buffer in DRAM because of different write powers [22, 119]. Therefore we conduct a study to explore the impact of the buffer size on the performance. Besides, we also discuss the impact of the buffer hit rate on the performance.

3.4.1 Buffer Hit Rate

We present the result of a sensitivity study of buffer hit rate in Figure 3.4. We estimate the performance based on NVM architecture in [22]. The result includes two baselines: DRAM and classic NVM. The classic NVM ignores the existence of buffer. We study the buffer hit rate from 50% to 90%; the row buffer size is set to 2Kb across the experiment.

We get the following observations from the results. First, the system throughput increases rapidly as the buffer hit rate increases. Each time the hit rate increase 10%, NOVA throughput increases 1.9% while ext4 and ext4-DAX throughput increase 1.4%. Second, NVM with 90%

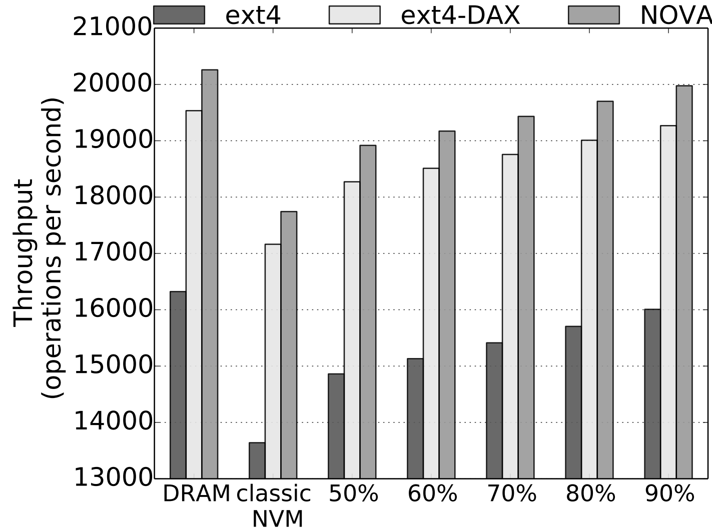


Figure 3.4: Filebench throughputs under various memory models. The *y*-axis represents the throughput. The *x*-axis represents memory models, which include DRAM, classic NVM with no buffer, and NVM with buffer hit rates under 50% to 90%.

hit rate has a minor performance difference to the DRAM performance. We discover that the performance difference between DRAM and 90% buffer-hit NVM is within 2% among all the file systems. Certain types of workloads can achieve 90% hit rates. For example, the stream read and write workload could have a decent buffer hit rate. We conduct the following estimation of buffer hit rate. We assume the memory access granularity is the cache size (64B). A single row buffer misses every time the workload accesses the subsequent buffer size data when accessing a continuous address space. Therefore, when the file size is much larger than the buffer size, the buffer hit rate is close to 96.9%.

We conduct a further investigation with zip workload, a typical stream application, to show that 90% hit rates are ordinary on real workloads. We employ Pin [120] to measure the row buffer locality for the workload. We use Pin measured page locality to estimate the buffer hit rate because of the similarity between page size and buffer size. The result shows that buffer locality is above 98% across three file systems. The variance between different systems is less than 0.01%. We can expect that the performance difference between NVM and DRAM-based

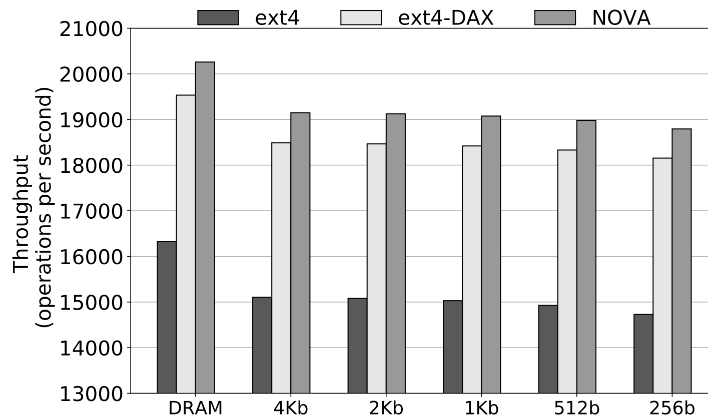


Figure 3.5: Filebench throughputs of DRAM and NVMs with different row buffer sizes. The workload has 60% sequential read operations.

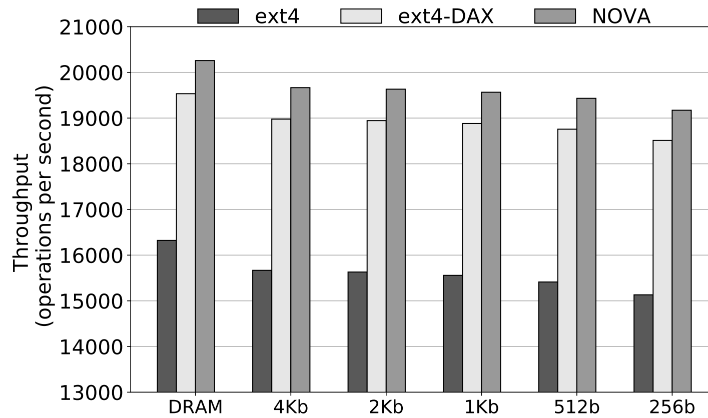


Figure 3.6: Filebench throughputs of DRAM and NVMs with different row buffer sizes. The workload has 80% sequential read operations.

systems is insignificant when running these workloads.

3.4.2 Buffer Size

We present result of a sensitivity study of buffer size in Figure 3.5 and Figure 3.6. We use the same performance model from the buffer hit rate study. We evaluate the buffer size from 256b to 4Kb. Our evaluation uses two different workload sequential read ratios: 60% (Figure 3.5) and 80% (Figure 3.6). We observe two findings from the result. First, the throughput increases with

the row buffer size. When buffer size increases, random memory accesses have a higher chance of hitting the address range within a buffer size. Therefore, the buffer hit rate increases with the buffer size. The increased buffer hit rate reduced the memory access cycles and results in high throughput.

Second, the shrunk row buffer size could potentially hurt NVM performance in the future. Under both 60% and 80% sequential read tests, the throughput drops 2% when the buffer size shrinks 50%. For the future NVM, buffer size might be $1/2$ or $1/8$ of regular row buffer size in DRAM [22], which could lead to 2%–8% performance loss. The future designs for the NVM buffer could explore the eviction and prefetch strategies to compensate for this performance loss.

3.5 Microarchitecture Analysis

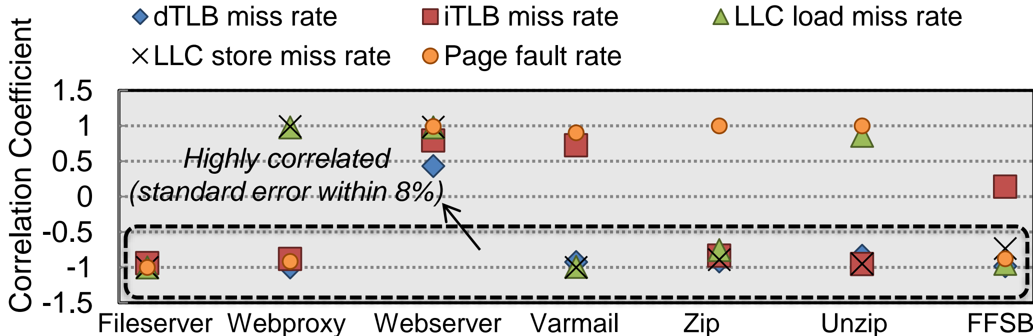


Figure 3.7: Correlation between workload performance and five hardware counter statistics. The *x*-axis represents different metrics. The *y*-axis represents the correlation coefficient between the throughput and the metric.

Furthermore, we examine the relationship between persistent memory system performance and various hardware and software event counters. Our experiments across six workloads identify that four hardware counter and one software events are most closely correlated to persistent memory system throughput. Figure 3.7 plots the statistical correlation between throughput and these five metrics across six benchmarks. *x*-axis represents each benchmark and *y*-axis is the correlation coefficient between throughput and each metric. These metrics are usually considered

the primary source of performance overhead. We make the following observations based on the abnormality of the results.

First, dTLB miss rate shows a high negative correlation with the throughput among all the benchmarks. This indicates that the address cache miss contributes more to the performance overhead than the data cache miss. Increase the size of the TLB cache could provide performance gain for the persistent memory system. Second, the page fault rate is least correlated to the performance. As the Figure 3.7 shows, the correlation between page fault and throughput varies drastically among different workloads. In this work, higher page faults do not necessarily lead to bad performance. The reason for this behavior is that all the page faults collected by *perf* are reported as minor page faults in the DAX featured file systems. DAX handles page fault as an exception. Because the NVM has the same level of latency as DRAM, the system does not suffer from much overhead as it does in conventional systems.

Last, LLC load and store miss show an insignificant correlation to the performance. However, this relates to the limitation of the perf tool. In the current system, all the misses on the LLC visit DRAM because the main memory is a pure DRAM. However, LLC miss could access NVM in hybrid main memory. Even though we partition DRAM to simulate hybrid memory in the current system, the perf tool cannot distinguish the LLC leads to NVM access or DRAM access. Therefore, the perf tool measured LLC miss rate is not equal to the NVM access. The perf tool should be extended to support profiling for the specific type of memory.

Based on these observations, we make the following suggestions. For the persistent file system design, we suggest dTLB miss rate be another guideline besides throughput for the design of the persistent memory file system. We also suggest page fault can be ignored, especially for the DAX supported file systems. We suggest an extension to the existing tools to support the hybrid memory for the hardware metrics profiling tools. This extension categorizes the DRAM and NVM access generated hardware metrics.

3.6 Implications on Persistent Memory System Design

Based on our observations, we provide the following implications and recommendations for persistent memory system design for reaping the full potential of this new data storage component.

First, *persistent memory system performance highly depends on workload data access patterns.* In our study, no file system always wins across various workloads and configurations. Different workloads have disparate read/write intensity, data access frequency, and data access granularity. Due to the latency gap between DRAM and NVM devices, along with the asymmetrical read and write latency of NVMs, system performance can vary significantly across read- and write-intensive workloads. For example, as shown in Section 3.3.1, the performance of all file systems decreases as workload write intensity increases, but the decreasing rate varies; therefore, the persistent memory file system design with the best performance also changes. **Recommendation:** Persistent memory system design needs to be aware of the type and configuration (especially memory access patterns) of applications running on top of it.

Second, *DAX substantially enhances file systems to leverage the performance benefit of persistent memory, yet may not be the only option.* Our results show that file systems with DAX (ext4-DAX and NOVA) provides much better performance than traditional ext4 file system by bypassing the page cache. However, current DAX design has several limitations. 1) DAX needs to be configured as a global parameter of file systems, regardless of the workload running on top of it. This can significantly reduce the flexibility of the file systems. 2) As shown in Section 3.3.1, DAX does not always out-perform the interfaces and mechanisms already existing in traditional system design; extensive tuning on system software and applications can be required to exploit the performance benefit of DAX best. Furthermore, DAX may not be the only option for supporting persistent memory in legacy file systems. For example, Linux interface o_direct (enabling direct_IO reads and writes) also allows workload's data access to bypass the page

cache, whereas is not susceptible to the two limitations that DAX has. This interface can be triggered by the workload. It also provides comparable or even better performance than DAX as we demonstrate. Note that enabling direct_IO was not recommended in traditional systems, because it would disable all the optimization mechanisms. However, this can change in the emerging persistent memory systems with discrepant characteristics than traditional systems. **Recommendations:** Persistent memory system design needs to provide a flexible interface for workloads to enable/disable DAX. The performance and flexibility of DAX design can also benefit from the automatic tuning of system configurations.

Third, *journaling or logging on data ensures data atomicity, but imposes substantial performance overhead even with the latest persistent memory system optimization*. As shown in Section 3.2.2, the performance of ext4 `data-ordered` mode is faster than `data-journal` mode, whereas the latter enforces data atomicity so allows better data recoverability. Even the latest developed persistent memory file systems, e.g., NOVA, can suffer from considerable performance overhead by enforcing the atomicity and consistency of data. **Recommendation:** As different applications may have different data reliability and recoverability requirement, persistent memory system performance can benefit from providing the options of multiple levels of atomicity guarantees. That way, persistent memory systems can achieve high performance with sufficient data reliability.

Fourth, *persistent memory system performance is highly correlated to the configuration and behavior of a small number of hardware components.* As shown in Section 3.5, across various workloads, persistent memory system performance is always closely related to the configuration and behavior of certain hardware components, such as last-level cache (LLC) and instruction/data TLB accesses. The rationale behind is that these hardware components are the most closely coupled with memory access among the components in the processor. **Recommendation:** In order to fully optimize persistent memory performance, system designers may want to spend most of their effort on optimizing the access to these highly correlated hardware components

instead of others.

Fifth, *Buffers on NVM devices can substantially impact persistent memory system performance and the accuracy of NVM emulation.* Whereas NVM devices do not require row buffers as DRAM does, the performance of persistent memory systems can substantially benefit from adopting buffers on NVM devices. Such buffer can significantly mitigate the performance penalty of the long latency, low bandwidth, and asymmetric read/write latency in NVM access. Besides, NVM emulation that ignores the buffers can affect the accuracy of the results and conclusions on NVM access performance. **Recommendations:** NVM hardware design can improve system performance by incorporating a buffer on NVM devices. System software and application design can leverage such buffers by exploiting the locality in data access at the granularity of the buffer size. To accurately emulating NVM access, NVM modeling and emulation tools need to be developed in a way that is aware of such buffers.

3.7 Conclusion

In this dissertation, we study the performance and hardware behavior of persistent memory systems by characterizing various workloads. We conduct experiments over three file systems using nine benchmarks. We provide several implications based on our observations in our experiments. First, persistent memory system performance is closely related to hardware behavior. Based on our results, a small number of hardware counter statistics have strong correlations with performance throughput. Second, systems that support atomic updates of data for persistent memory have more performance overhead. For example, ext4 when mounted in `data-journal` mode, the throughput is much lower when compared to other modes. Future file system design of NVM can provide a cache-bypass interface for the workload, thereby leveraging the underlying NVM. It is evident from our results that ext4-DAX outperformed ext4 in terms of performance for all the workloads. Third, buffers on NVM chips can act as a critical factor to enhance NVM

performance. In contrast to current NVM modeling techniques, considering buffer hit rate results in reduced overhead. These insights provide meaningful details for optimizing persistent memory system software and hardware designs.

Acknowledgments

This chapter contains material from “Persistent Memory Workload Characterization: A Hardware Perspective.”, by Xiao Liu, Bhaskar Jupudi, Pankaj Mehra, and Jishen Zhao, which appears in the Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC 2019). The dissertation author is the primary investigator and first author of this paper.

Chapter 4

Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability

Modern cloud servers adopt increasingly larger main memory and last-level caches (LLC) to accommodate the large working set of various in-memory computing [121, 122], big-data analytics [123, 124], deep learning [125], and server virtualization [123] applications. The memory capacity demand requires further process technology scaling of traditional DRAMs, e.g., to sub-20nm technology nodes [126, 1]. Yet, such aggressive technology scaling imposes substantial challenges in maintaining reliable and high-performance memory system operation due to two main reasons. First, resilience schemes to maintain memory reliability can impose high-performance overhead in future memory systems. Future sub-20nm DRAMs require stronger resilience techniques, such as in-DRAM error correction codes (IECC) [70, 71, 72, 1, 4, 127] and rank-level error correction codes (RECC) [128, 129, 130], compared to current commodity DRAMs. Figure 4.1 shows that such error correction code (ECC) schemes impose significant performance overhead to the memory hierarchy that employs sub-20nm DRAM as main memory.

Second, even with strong resilience schemes, future memory systems do not appear to be as reliable as the state-of-the-art. In fact, due to increased bit error rates (BERs), even when we combine RECC and IECC, sub-20nm-DRAM-based memory systems with 3D-stacked DRAM LLC can have lower reliability than a 28nm-DRAM-based memory system that employs only RECC [1], as seen in Figure 4.1 (when comparing the triangle and the circle).

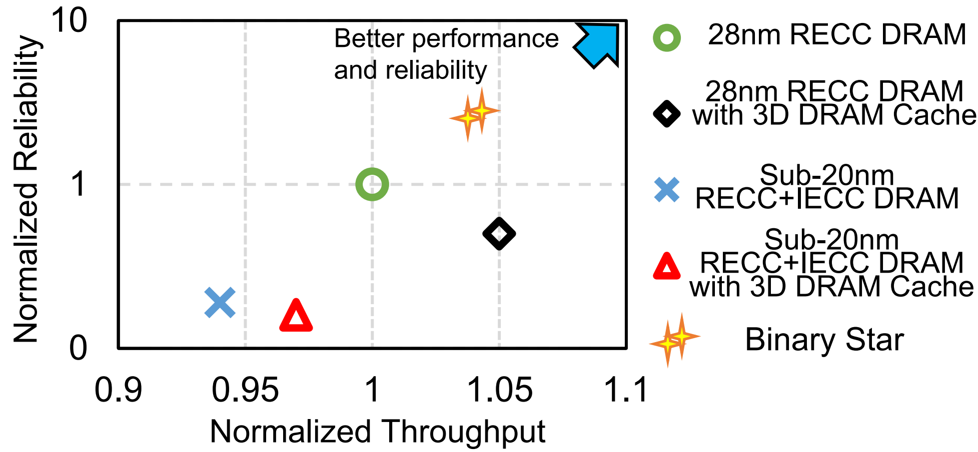


Figure 4.1: Reliability and throughput of various LLC and main memory hierarchy configurations normalized to 28nm DRAM with RECC. We define "reliability" as the reciprocal of device failures in time (FIT) [1, 2], i.e., 1/FIT. Reliability data is based on a recent study [1].

To address the DRAM technology scaling challenges, next-generation servers will adopt various new memory technologies. For example, Intel's next-generation Xeon servers and Lenovo's ThinkSystem SD650 servers support Optane DC PM [131, 132], which demonstrates the practical use of large-capacity NVMs in servers. Intel's Knights Landing Xeon Phi server processors integrate up to 16GB multi-channel DRAM (MCDRAM) [133], which is a type of 3D-stacked DRAM,[1] used as the LLC. 3D DRAM and NVM promise much larger capacity than commodity SRAM-based caches and DRAM-based main memory, respectively.

However, reliability and performance issues remain. For example, compared with a 28nm DRAM (represented by the circle in Figure 4.1), a sub-20nm DRAM (the cross in the figure) has both worse performance and reliability. When 3D DRAM LLC is used, the sub-20nm 3D-DRAM-

[1]We use *3D DRAM* and *3D-stacked DRAM* interchangeably in this dissertation.

cached DRAM (represented by the triangle in Figure 4.1) outperforms the sub-20nm DRAM (the cross in the figure), but suffers from degraded reliability. We make similar observations when we compare the 28nm 3D-DRAM-cached DRAM (the diamond in Figure 4.1) and the 28nm DRAM (the circle in the figure).

Most NVM technologies are less vulnerable to soft errors than DRAM [54, 55]. However, many of these technologies have much lower endurance compared to DRAM [22, 56]. As a result, most NVM-based main memory needs to adopt hard error protection techniques, imposing both performance and storage overheads compared to DRAM-based main memory [3, 134, 56]. NVM also enables persistent memory techniques [110, 135], which allows data structures stored in NVM to be recovered after system failures. However, the performance of persistent memory systems is degraded by the high performance and storage overheads of multiversioning and write-order control mechanisms (e.g., cache flushes and memory barriers) [136, 135, 61].

Our goal in this work is to achieve both high reliability and high performance, when we scale the capacity of LLC and main memory using new 3D DRAM and NVM technologies. To achieve our goal, we propose Binary Star,[2] a coordinated memory hierarchy reliability scheme, which consists of 1) coordinated reliability mechanisms between 3D DRAM LLC and NVM (Section 4.2.1), and 2) NVM wear leveling with consistent cache writeback (Section 4.2.2). The key insight of our design is that traditional memory hierarchy designs typically strive to *separately* optimize the reliability of LLCs and main memory with *decoupled* reliability schemes, yet coordinating reliability schemes across the LLC and main memory enhances the reliability of the *overall* memory hierarchy, while at the same time reducing unnecessary ECC, multiversioning, and ordering control overheads that degrade performance. As illustrated in Figure 4.1, Binary Star (represented by a double star) achieves much better reliability than state-of-the-art resilience schemes with various LLC and main memory configurations. Even with a

[2] A "binary star" is a star system consisting of two stars orbiting around their common barycenter. Our coordinated memory hierarchy reliability design appears like a star system that consists of two "stars" – a 3D DRAM LLC and an NVM main memory.

slower PCM as the main memory, Binary Star's performance is comparable to the combination of 28nm DRAM main memory and 3D DRAM LLC. We show that Binary Star provides benefits across various types of workloads, including in-memory databases, in-memory key-value stores, online transaction processing, data mining, scientific computation, image processing, and video encoding (Section 4.5).

This dissertation makes the following key contributions:

- We identify the inefficiency of traditional reliability schemes that are decoupled and uncoordinated across the memory hierarchy (i.e., between the LLC and main memory).
- We propose "Binary Star", the first coordinated reliability scheme across 3D DRAM LLC and NVM main memory. Our design leverages consistent cache writeback in NVM to recover errors in 3D DRAM LLC. The LLC requires only error detection to perform consistent cache writeback. This significantly reduces the performance and storage overhead of consistent cache writeback.
- We develop a new technique that coordinates our consistent cache writeback technique with NVM wear leveling to reduce the performance, storage, and hardware implementation overheads of memory reliability schemes.
- We develop a set of new software-hardware cooperative mechanisms to enable our design. Binary Star is transparent to the applications and thus requires no application code modification.

4.1 Background and Motivation

Existing memory hierarchy designs typically strive to optimize the reliability of individual components (either caches or main memory) in the memory hierarchy [126, 127, 1]. These designs lead to decoupled and uncoordinated reliability across different memory hierarchy levels. As a result, existing designs suffer from high performance overhead and high hardware implementation cost. In this section, we motivate our coordinated reliability mechanism by first discussing

the inefficiency of existing reliability schemes for DRAM in Section 4.1.1, 3D DRAM LLC in Section 4.1.2, and NVM in Section 4.1.3. Then, we illustrate issues with decoupled, uncoordinated reliability schemes in Section 4.1.4.

4.1.1 DRAM Main Memory

DRAM has been used as the major technology for implementing main memory. However, DRAM scaling to sub-20nm technology nodes introduces substantial reliability challenges [1, 126, 137, 138].

DRAM errors. DRAM errors can be roughly classified into transient errors and hard errors. Transient errors are caused by alpha particles and cosmic rays [139], as well as retention time fluctuations [140, 141, 71, 72, 142]. These transient errors are less likely to repeat and can be avoided after rewriting the corresponding erroneous bits. Hard errors are caused by physical defects or failures [129, 143, 2]. These hard errors repeatedly occur in the same memory cells due to permanent faults.

Challenges with process scaling. Process technology scaling can compromise DRAM reliability. Randomly distributed single-cell failure (SCF) is the primary source of device failures [2]. The frequency of SCF increases as DRAM cell size reduces, because as smaller transistors and capacitors are more vulnerable to process variation and manufacturing imperfections [141, 1]. Redundant rows and columns can be used to fix SCFs and keep DRAM device yield high [1]. Recent studies show that in-DRAM ECC (IECC), which integrates ECC engines inside the DRAM device, is a promising method to address DRAM reliability issues [144, 70, 1]. However, IECC leads to significant storage overhead in DRAM chips [4, 70, 127, 1]. Furthermore, IECC needs to be combined with RECC [1], to achieve reliability close to commodity DRAM [2], imposing substantial performance overhead (See Section 4.5).

4.1.2 3D DRAM Last Level Cache

Using large 3D DRAM as an LLC can accommodate the increasing working set size of server applications and reduce the performance overhead of capacity scaling. However, the Through Silicon Vias (TSVs) connecting the layers of 3D DRAM can increase the BER by introducing additional defects [145]. Previous studies adopt two-level error correction/detection schemes to improve 3D DRAM reliability [146, 147, 148, 149]. However, these usually require heavy hardware modifications and significant storage overhead.

4.1.3 NVM Main Memory

To tackle the scalability issues with DRAM technologies, NVM (e.g., PCM [22, 87], STT-RAM [150, 98], and RRAM [88]) introduces a new tier in the memory hierarchy, due to its promising scalable density and cost potential [151, 152, 153]. Many data center server software and hardware suppliers are adopting NVMs in their next-generation designs. Examples include Intel's Optane DC PM [131], Microsoft's storage class memory support in Windows OS and in-memory databases [154, 155], Red Hat's persistent memory support in the Linux kernel [156], and Mellanox's persistent memory support over the network fabric [157].

NVM errors. Similar to DRAM, NVM is also susceptible to transient and hard errors. However, the sources of these errors are different from DRAM. Alpha particles and cosmic rays are less of an issue with NVM. Instead, NVM transient errors are largely caused by resistance drift [55]. In fact, resistance drift can dominate bit errors in MLC NVMs (e.g., MLC PCM [54]). Resistance drift is a less critical issue with SLC NVMs, as it can be addressed by infrequent scrubbing (e.g., every several seconds) [54, 55]. NVM hard errors are often caused by limited endurance. For example, PCM cells can wear out after 10^7-10^9 write cycles [22, 158]. Certain NVM technologies, such as certain types of PCM, may have much lower transient BER compared to DRAM before the cells wear out [159, 158].

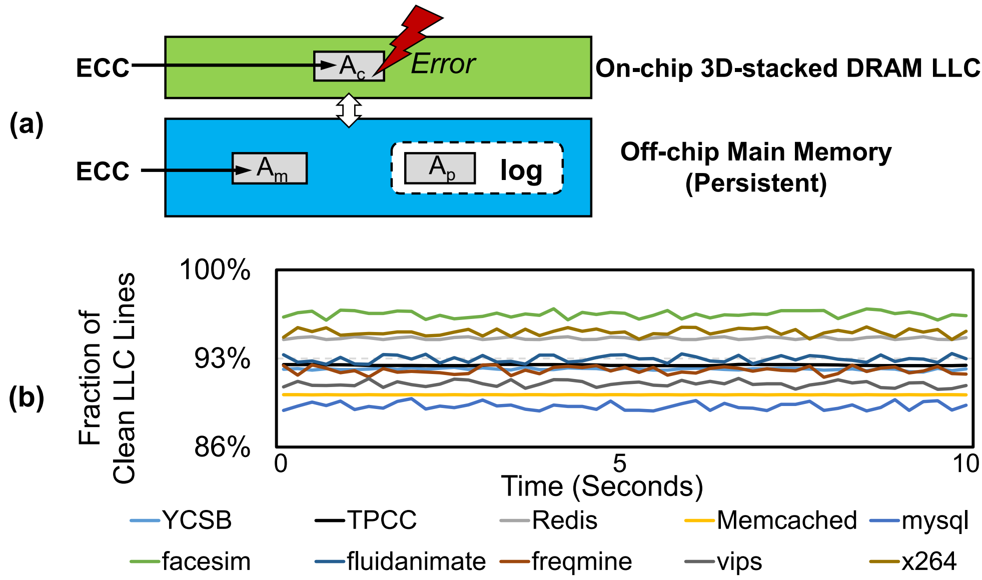


Figure 4.2: (a) Multiple reliability mechanisms for a cache line across LLC and main memory. (b) Fraction of clean LLC lines during application execution. We collected data for ten seconds after the benchmark finishes the warm up stage.

To mitigate transient and hard errors, previous works propose techniques to reduce MLC NVM BER at low performance and storage overheads [160, 55, 54]. SLC NVM main memory can be effectively protected using existing ECC mechanisms, as done in state-of-the-art DRAM [159, 55]. In addition to ECC, scrubbing can be used to address NVM transient errors (resistance drift) [54, 55]. To mitigate hard errors caused by endurance issues, previous studies adopt wear leveling [22, 119, 161], which scatters NVM accesses across the whole device to ensure that all bits in the device wear out in a balanced manner. Hard errors (bits that are already worn out) can be remapped [3, 134], which disables the faulty bits, redirecting accesses to alternative physical locations in the memory.

4.1.4 Issues with Decoupled Reliability

A modern system that employs a large 3D DRAM as LLC and a high-capacity NVM as main memory typically utilizes two *separate* uncoordinated reliability schemes for DRAM and NVM [146, 162, 3, 134]. The key issue with such decoupled reliability across the LLC and main

memory is redundant protection across the memory hierarchy: the same piece of data can be repeatedly (and unnecessarily) protected multiple times. Figure 4.2 (a) shows an example. A clean cache line-*A* is protected by ECCs in both 3D DRAM LLC and main memory (represented by A_c and A_m, respectively). In fact, as shown in Figure 4.2 (b) (our methodology, system, and workload configurations are described in Section 4.4), a large portion of the 3D DRAM LLC lines are clean throughout execution time (after warm-up) with an *identical* copy sitting in main memory, and both copies are protected with ECCs. In this case, carefully enforcing the reliability of only one of the data copies is sufficient. For instance, without ECC in LLC, we can simply invalidate a clean LLC line A_c in case it has a transient error; the error will be naturally recovered by servicing a cache miss to *A* with an ECC-protected copy A_m from main memory. As such, the LLC only needs to detect whether or not A_c has an error. However, it is challenging to fully exploit the coordination between cache and main memory. For example, what if cache line A_c is dirty? Once A_c has errors, we lose the new value of that cache line because the copy in memory (A_m) does not have the up-to-date value. Even if A_c is *not* dirty, A_c with A_m can lead to inconsistent data structures in the program: A_c may belong to a data structure comprising multiple cache lines. If we load the old value A_m to recover the LLC line, the data structure will become inconsistent: the old value of A_m will be inconsistent with new values of other dirty LLC lines that belong to the same data structure. Section 4.2 discusses our solution to the reliability and consistency problems of both clean and dirty LLC lines.

While there are several prior works that handle some SRAM cache and DRAM main memory reliability issues together [163, 164, 165], these are not desirable for the combination of 3D DRAM and NVM. On the one hand, these works are hard to scale to the 3D DRAM size. On the other hand, because these designs require frequently clearing dirty cache lines [163, 165] or off-loading SRAM cache ECC to the main memory [164, 165], these mechanisms further consume NVM's short endurance and low bandwidth.

Instead of coordinating the reliability schemes of caches and main memory, persistent

memory designs enhance the reliability of the main memory when a system completely loses data in caches due to a system crash or power failure [31, 166]. Persistent memory allows data in NVM to be accessed via load/store instructions like traditional main memory, yet recoverable across system reboots [64, 96]. As shown in Figure 4.2 (a), persistent memory systems typically maintain an additional copy of data (e.g., A_p that can be a log entry or a checkpoint), which is used to recover original data (A_m) in NVM main memory if needed after a system crash. This process can be done through controlled data versioning (e.g., by logging [61] or shadow paging [64]), or via write-ordering (e.g., by cache flushes and memory barriers [64]). However, relying on persistent memory mechanisms for error recovery purposes has at least two major disadvantages: first, all these techniques require code modifications, which require significant software engineering effort [108]; second, applications with no persistence requirements suffer from substantial performance and storage overheads in main memory access [108, 135, 61].

In summary, we need to address the overhead and scalability issues with (i) the decoupled reliability schemes across caches and main memory in state-of-the-art memory systems that combine 3D DRAM and NVM, and (ii) the challenges of exploiting persistent memory techniques in order to efficiently maintain the reliability of memory hierarchy. **Our goal** in this work is to design a comprehensive coordinated memory hierarchy reliability scheme that is efficient and effective for memory hierarchies that combine multiple technologies.

4.2 Binary Star Design

To address the aforementioned reliability challenges, we propose Binary Star. Binary Star 1) coordinates reliability schemes across 3D DRAM LLC and NVM main memory, and 2) uses an efficient hardware-based 3D DRAM LLC error correction and a software-based error recovery mechanism.

3D DRAM LLC and NVM technologies allow the memory hierarchy to continue to scale

in a cost-efficient manner. Figure 4.3 shows our basic architectural configuration. We assume that higher-level caches are SRAM. The LLC in the processor is a 3D DRAM. We use SLC PCM, which is a well-understood NVM technology, as a representative for NVM main memory to make our design and evaluation concrete. Our design principles can be applied to various NVM technologies. In this section, we describe our design principles and mechanisms, using Figure 4.4 to illustrate them. We leave the description of the implementation of our mechanisms to Section 4.3.

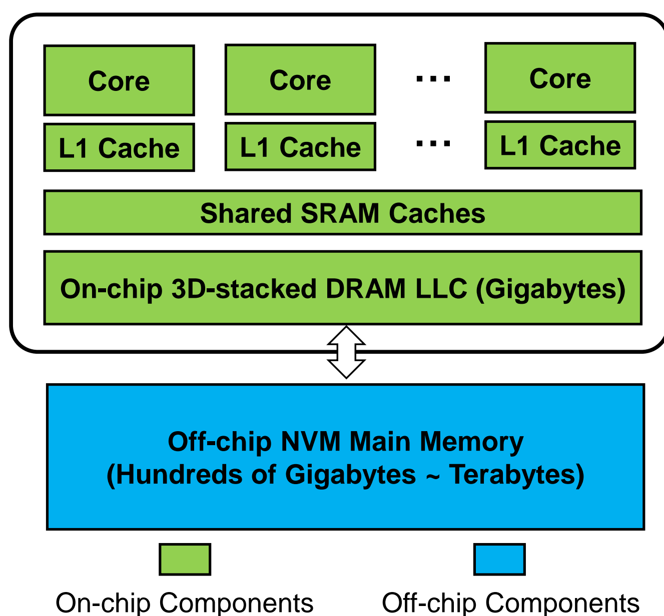


Figure 4.3: Basic architecture configuration.

4.2.1 Coordinated Reliability Scheme

Our coordinated reliability scheme strives to optimize the reliability of the 3D DRAM LLC and NVM hierarchy as a whole, instead of separately optimizing the reliability of individual components at different hierarchy levels, at low performance and storage cost. Our scheme is enabled by the following key observation:

Observation 1: *Errors in the LLC can be corrected by consistent data copies in NVM*

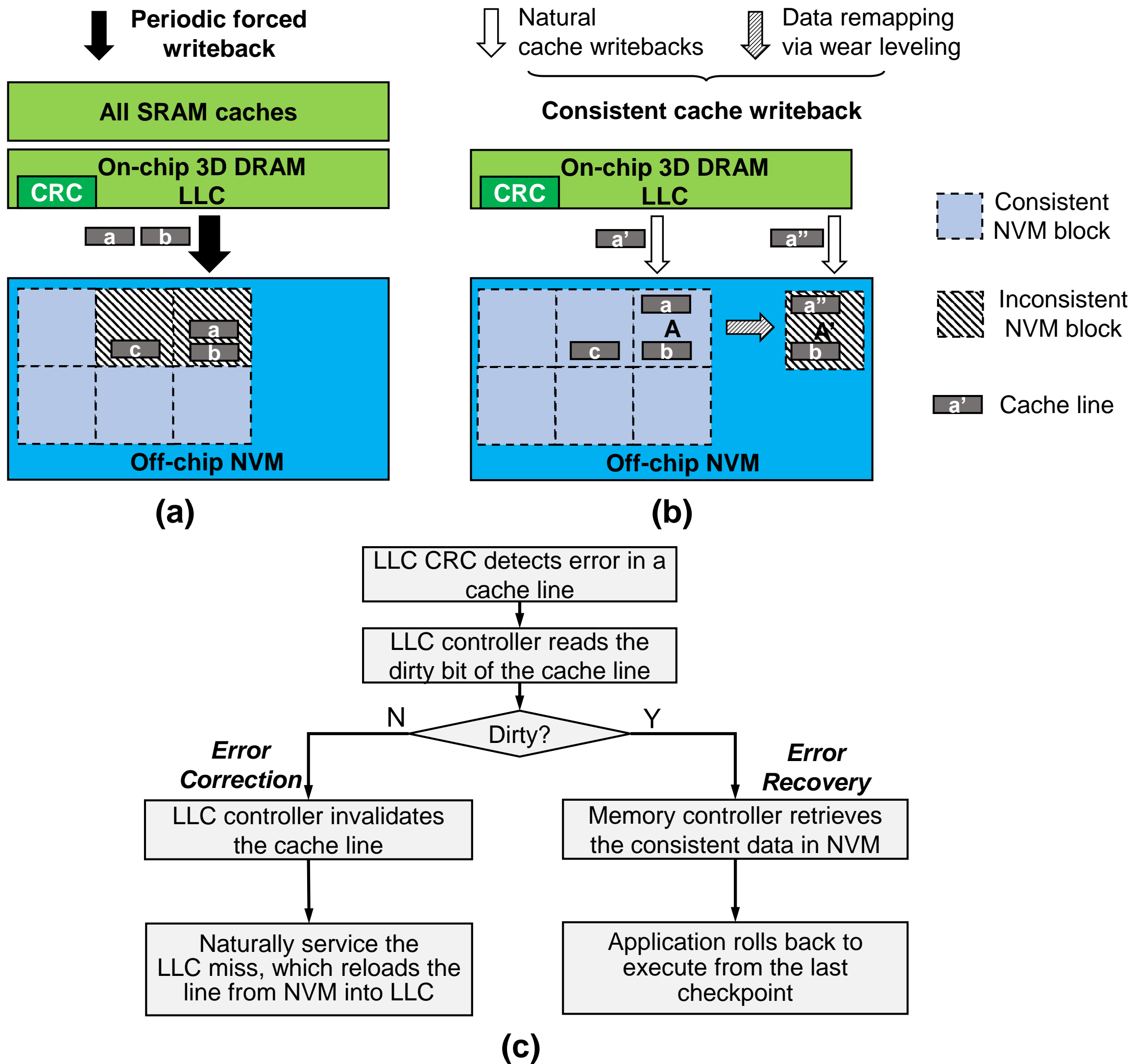


Figure 4.4: Binary Star overview. (a) Periodic forced writeback and (b) Consistent cache writeback. (c) Binary Star error correction and error recovery mechanisms. A solid filled square represents a checkpointed consistent data block. A striped filled square represents a remapped inconsistent data block.

main memory.

The data stored in the LLC is a subset of the data in main memory. Therefore, as long as the main memory maintains a consistent data copy at a known point in time, errors in the corresponding LLC line can be corrected (or ignored) using the consistent data copy in memory. This observation makes error correction codes in the LLC unnecessary. Instead, Binary Star

coordinates 3D DRAM LLC and NVM reliability in the following manner: the LLC adopts only cyclic redundancy check (CRC) codes [167] to ensure errors can be detected; NVM maintains consistent data copies that can be used to correct (or avoid) the detected errors in the LLC. While it is straightforward to implement CRC in the LLC, maintaining consistent data in NVM for error recovery purposes is complicated for two reasons. First, storing and updating multiple versions of data can incur significant storage and performance overhead [31, 108]. Second, LLC writebacks can corrupt consistent versions of data in NVM: For example, a data structure update that spans multiple cache lines can result in multiple dirty cache lines; yet only one or several of the dirty cache lines may be written back to NVM main memory at a given time, leaving the state of the data structure in main memory inconsistent.

We need to avoid such inconsistency of data in NVM main memory, in order to use the main memory contents to correct errors that happen in LLC lines. To this end, we propose *periodic forced writeback* and *consistent cache writeback* (depicted in Figure 4.4 (a) and (b), respectively).

Periodic forced writeback. During the execution of an application, portions of updated data might remain in the dirty cache lines, causing inconsistent data to be present in the NVM. Because dirty cache lines contain all the most recent updates, Binary Star periodically forces the writeback of dirty cache lines from all cache levels into the NVM main memory (e.g., cache lines *a* and *b* in Figure 4.4 (a)). These written-back data along with previously written-back dirty cache lines (e.g, cache line *c* in the figure.) generate a set of consistent data, i.e., a *checkpoint of data*, in the NVM by ensuring that all updates to data are propagated to the NVM. When taking the checkpoint, Binary Star marks all updated NVM blocks as consistent (i.e., the striped squares in Figure 4.4 (a) turn into solid squares after the checkpoint is taken). To recover to this consistent checkpointed state in the presence of LLC errors, Binary Star maintains a single checkpoint and saves the state of an affected process to ensure that the process is able to recover to the point *immediately after* the periodic forced writeback. The checkpointed consistent data in

NVM stays intact in NVM until the end of the next forced writeback period. This checkpointed consistent data in NVM can be used to recover from LLC errors detected by the CRC between two consecutive forced writeback periods. In our evaluation, we perform detailed sensitivity studies (Section 4.5.3) and demonstrate that 30 minutes is the best periodic forced writeback interval for the Binary Star system with our benchmarks.

Consistent cache writeback. To maintain consistency of the checkpointed data *between two forced writeback periods*, we propose a consistent cache writeback mechanism as shown in Figure 4.4 (b). This mechanism redirects natural LLC writebacks to NVM locations that are different from the locations of the checkpointed data, so that checkpointed data stays consistent.

Application Transparency. Maintaining data consistency through persistent memory typically requires support for (i) a programming interface (e.g., transactional interface), (ii) multi-versioning (e.g., logging or copy-on-write), and (iii) write-order control (e.g., cache flushes and memory barriers). These can impose non-trivial performance and implementation overheads on top of a traditional memory hierarchy [136, 31]. Furthermore, persistent memory systems need to redirect data upon each persistent data commit [136, 110, 61]. Our design does not impose large overheads due to three major reasons. First, the granularity of our consistent cache writeback mechanism is only one cache line. Therefore, we do not require a transactional interface to the application to determine the commit granularity. Second, our consistent cache writeback mechanism is used to update the application's working data, rather than the checkpointed consistent data. Therefore, we do not need to enforce write-order control of consistent cache writeback. Finally, our design only needs to ensure that natural LLC writebacks do not overwrite the original, i.e., checkpointed copy of consistent data, which is reserved to be used to correct errors in LLC. As such, we do *not* redirect each natural LLC writeback to a new NVM location. Rather, in our technique, the LLC writebacks to the same data block can repeatedly *overwrite* the same NVM block[3] that is outside of the checkpointed consistent data. As shown in Figure 4.4 (b),

[3] A block is the minimal granularity of remapping and data consistency ensured in the NVM in our design. The

the up-to-date version (e.g., cache lines a'' and b) in NVM is visible to applications, while the checkpointed consistent version (e.g., the original data a) is hidden from applications.

4.2.2 Coordinating Wear Leveling and Consistent Cache Writeback

Our consistent cache writeback mechanism requires redirecting a natural LLC writeback to an NVM location that is different from the location of the checkpointed data. However, doing so with a traditional memory system design would introduce extra effort to (i) maintain address remapping with metadata and free data block management and (ii) manage alternative memory regions to store redirected natural LLC writebacks. These can impose substantial performance and NVM storage overheads. Instead of explicitly maintaining data consistency for each cache writeback, Binary Star leverages the remapping techniques that are already employed by existing NVM wear leveling mechanisms. Our method redirects the natural LLC writebacks to memory locations different from the checkpointed data locations during consistent cache writeback. We make the following key observation:

Observation 2: *NVM wear leveling naturally redirects and maintains the remapping of data updates to alternative memory locations.*

Note that it is difficult to leverage NVM wear leveling with regular persistent data commits on a system with persistent memory. To provide crash consistency, persistent memory systems typically require each data commit to be redirected to a new location, e.g., a new log entry [61], a newly allocated shadow copy [64], a new checkpoint [108], or a new version managed by hardware [31, 108]. However, NVM wear leveling redirects data updates *only when* the original physical memory location is repeatedly overwritten for a given number of times [119]. Therefore, the remapping of data typically happens infrequently. If we would like to exploit wear leveling to provide consistency in persistent memory, wear leveling needs to be synchronized with each persistent data commit. This likely imposes a substantial performance overhead because it leads

block size is larger than a cache line and depends on the granularity of wear leveling mechanism used.

to prohibitively frequent data remapping and metadata updates.

In contrast, Binary Star's consistent cache writeback mechanism can effectively take advantage of existing NVM wear leveling mechanisms. During the interval between two consecutive occurrences of periodic forced writeback, Binary Star's consistent cache writeback mechanism needs to redirect NVM updates of each data block only once. After the first update to the consistent data block is redirected due to a natural LLC writeback (e.g., the update of cache line a', which is overwritten by the later update a'', as shown in Figure 4.4 (b)), subsequent natural LLC writebacks of the same data block can repeatedly overwrite the same redirected data location (e.g., the updated version a'', as shown in Figure 4.4 (b)). Therefore, Binary Star does *not* need to synchronize wear leveling with each consistent cache writeback. Instead, Binary Star must trigger one single extra *data remapping* via the wear leveling mechanism at only the *first* update to a consistent data block; wear leveling can operate as is during the rest of the execution.

By coordinating wear leveling with consistent data writeback, Binary Star enables better utilization of wear leveling. Traditional NVM wear leveling remaps data of frequently accessed NVM blocks to infrequently accessed blocks [119], without being explicitly aware of the consistency of data. Our design exploits this *remapping* capability by exposing data consistency to the wear leveling, such that we can guarantee data consistency by using the data remapping capability of wear leveling (as illustrated in Figure 4.4 (b)). In particular, we coordinate NVM wear leveling with our consistent cache writeback mechanism in the following manner:

- *Binary Star-triggered wear leveling*: When the memory controller receives an LLC writeback (either a natural LLC writeback or a periodic forced writeback) to a checkpointed consistent data block, Binary Star actively triggers wear leveling to remap the LLC writeback to another NVM block. Such remapped updates, along with the original blocks in the current checkpointed consistent data, form the next checkpoint of consistent data. As a result, Binary Star ensures that the *current* version of the checkpointed consistent data block remains intact.
- *Natural wear leveling*: During the rest of the time, wear leveling can operate as is. Note that

Binary Star does *not* prevent natural wear leveling from remapping the checkpointed consistent data blocks. In case a checkpointed consistent data block is remapped, Binary Star simply updates the metadata to reflect the new location of the block (Section 4.3).

- *NVM updates without remapping*: NVM wear leveling is performed periodically [119]. During the interval when wear leveling is not performed, natural LLC writebacks to the blocks outside of the checkpointed consistent data blocks can directly overwrite the same location.

4.2.3 3D DRAM LLC Error Correction and Error Recovery

Our 3D DRAM LLC uses CRC codes that can only detect errors. Binary Star recovers from them using the copies of cache lines that are in the NVM main memory. Based on our coordinated reliability scheme, there can be two types of data copies in NVM for a given block: 1) the checkpointed consistent copy, which is generated by periodic forced writeback and which contains the consistent data, and 2) the latest updated copy that is managed by Binary Star-triggered wear leveling. Once CRC detects an error in a given 3D DRAM LLC cache line during an access to the 3D DRAM LLC, Binary Star performs error correction and recovery via the following steps that we also illustrate in Figure 4.4 (c): First, the LLC controller reads the dirty bit of the erroneous cache line.

If the LLC line is clean, Binary Star performs error correction in hardware as follows: 1) the LLC controller invalidates the cache line, 2) the corresponding LLC access turns into an LLC miss, which is serviced as a natural LLC miss without any hardware modifications. The LLC miss generates an access to the corresponding most recently updated data block in the NVM. The corresponding data block could reside in either the checkpointed consistent data block or the remapped data block.

If the LLC line is dirty, Binary Star determines that it is an uncorrectable error. If such an error is detected, Binary Star triggers error recovery through the system software by 1) reverting the memory state to the checkpointed consistent version of application data, and 2) rolling back

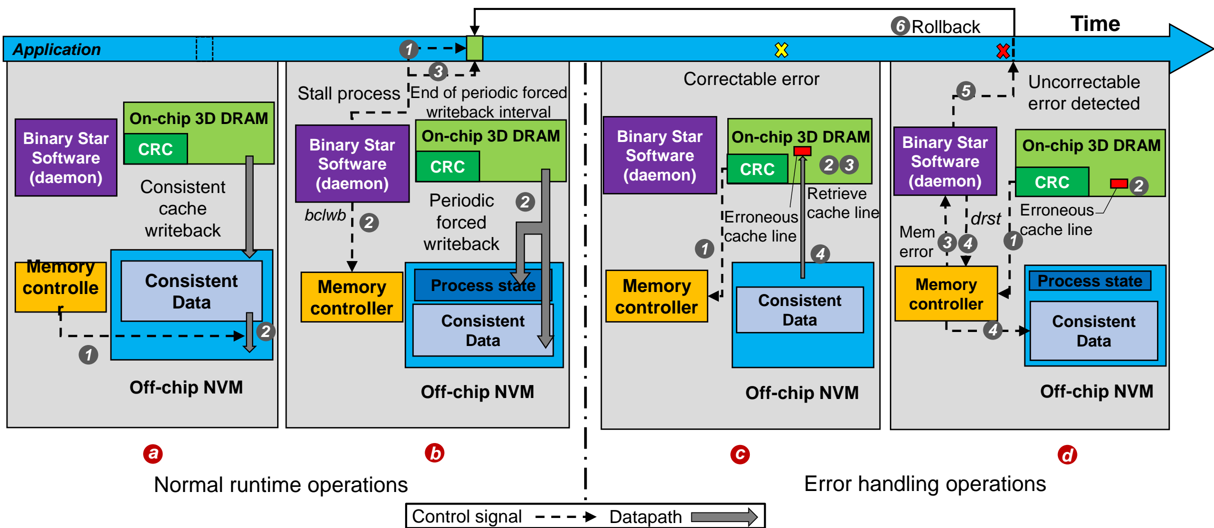


Figure 4.5: An example of running an application on Binary Star.

the application to execute from the checkpoint.

4.2.4 Putting it All Together: An Example

With these design components, we demonstrate a simple example to show the detailed operation of Binary Star when running an unmodified application. Figure 4.5 depicts an example of four cases (ⓐ to ⓓ) that a running application may encounter when our proposed mechanisms are employed. These four cases are software and hardware operations that conduct consistent cache writeback (ⓐ) and periodic forced writeback (ⓑ) during normal execution, as well as handling of correctable errors (ⓒ) and handling of uncorrectable errors (ⓓ), when an error is detected.

- Binary Star hardware conducts consistent cache writeback during normal application execution (ⓐ). Consistent cache writeback is transparent and asynchronous to the application. When the LLC has to evict a cache line, the memory controller identifies if the cache line belongs to a consistent block (❶). The cache line is directly written into NVM if it belongs to an inconsistent block, otherwise the wear leveling remaps the consistent block to a new block and writes the cache line into it (❷). Section 4.3.2 provides the details.

- **Binary Star daemon** conducts periodic forced writeback during normal execution (ⓑ). The daemon stalls the application (❶), saves the process state, and executes the **Binary Star cache line writeback (*bclwb*)** instruction, which enables the memory controller to write back all dirty cache lines to main memory without invalidation and mark the corresponding NVM data blocks as consistent (❷). This procedure creates a new checkpoint and invalidates the previous one. The daemon resumes application execution once the periodic forced writeback finishes (❸).
- A correctable error (ⓒ), i.e., an error in a clean LLC line, is transparent to the software. Hardware detects and corrects these errors: **memory controller** detects the error using CRC (❶), reads the dirty bit and identifies that this is a correctable error (❷), invalidates the LLC line (❸), and issues a cache miss to retrieve the correct copy of the cache line from NVM (❹).
- An uncorrectable error, i.e., an error in a dirty LLC line, requires software and hardware operations to recover the LLC line (ⓓ). **LLC controller** detects the error using CRC (❶), reads the dirty bit and identifies that this is an uncorrectable error (❷), and sends a signal to the Binary Star daemon (❸). The daemon stalls the application and triggers the rollback procedure (❹). The daemon uses the **Binary Star data reset (*drst*)** instruction to reclaim the previously checkpointed consistent data blocks (i.e., the checkpoint) in NVM (❺). The whole procedure rolls the memory hierarchy back to the consistent checkpointed state and resumes normal application execution (❻).

4.3 Implementation

This section discusses our Binary Star implementation, including modifications we make to the 3D DRAM LLC, the memory controller and the system software.

4.3.1 3D DRAM LLC Modification

We replace the ECC in 3D DRAM LLC with a 32-bit CRC (CRC-32) [167] for each 64B LLC line. This design allows Binary Star to detect up to eight-bit errors as well as a portion of nine or more bit errors, and all burst errors up to 32 bits long [167]. Binary Star also modifies the finite state machine in the LLC controller to support the steps required to handle LLC error detection, correction, and rollback as listed in Section 4.2.3.

4.3.2 Memory Controller Modifications

Binary Star modifies the state-of-the-art area- and performance-efficient Segment Swap design [119] for NVM wear leveling. Segment Swap counts the number of writes to each segment, and periodically swaps a hot segment with a cold segment.[4]

Modifications to the wear leveling mechanism. To coordinate the NVM wear leveling mechanism with our consistent cache writeback mechanism, we modify Segment Swap [119] design as shown in Figure 4.6. In the original design, the mapping table maps a "virtual" segment to a "physical" segment. In our design, we maintain a set of metadata with each segment that includes 1) a *consistent bit* that indicates whether the segment belongs to the consistent checkpoint, 2) a *free bit* that indicates whether the segment is free, and 3) a *segment number* that identifies the segment that stores the corresponding checkpointed consistent data. The metadata is stored in a reserved NVM space similar to previous works [119, 161]. During periodic forced writeback, the memory controller uses the segment numbers to release each segment belonging to the previous checkpointed consistent data (e.g., segment *A* ❶ in Figure 4.6) by resetting the *free bit* and adding them to the free list. After all dirty cache lines are written back, the memory controller also marks the current inconsistent segment as consistent (segment *A'* ❷) and deletes the stored segment number. During consistent cache writeback, NVM wear leveling performs remapping

[4] Segment is the granularity of wear leveling. In Binary Star, we set the segment size to be one Mbyte, similarly to the original design [119].

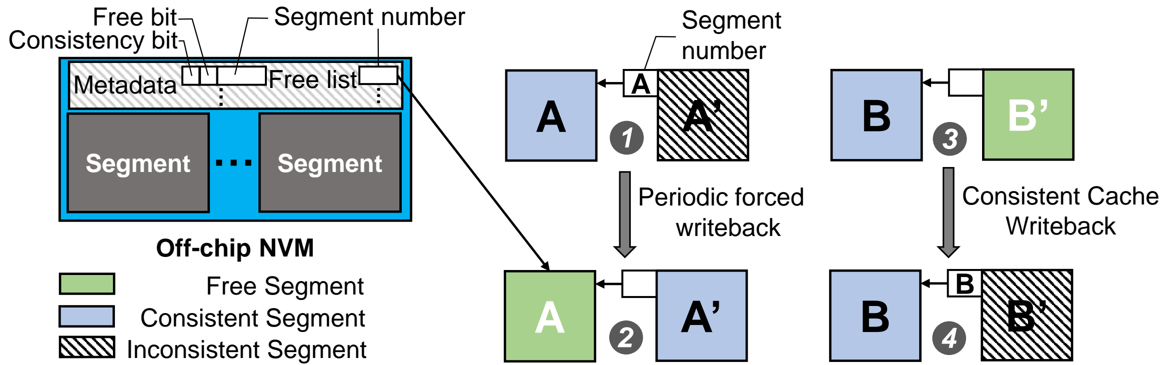


Figure 4.6: Modifications to NVM wear leveling.

by selecting a free segment, copying data from the consistent segment (segment *B* ❸) to it, and marking it as an inconsistent segment (segment *B'* ❹).

Modification to NVM write control. We modify the memory controller's wear leveling control logic to enable our consistent cache writeback mechanism (Section 4.2.2). When the memory controller performs a cache writeback, the wear leveling control logic reads the *consistent bit* in the metadata space of NVM to determine the consistency state of a segment that the cache line belongs to. If the cache line belongs to an inconsistent segment, the controller directly writes data into NVM. If the cache line belongs to a consistent segment, the segment is remapped via the wear leveling mechanism. In this case, the controller selects a free segment from the free list. The free list keeps track of all the free segments' numbers and is stored in the memory controller. The writeback data, along with the original data in the old segment, is moved to the new free segment. The memory controller also changes the entry in the mapping table to the new segment [119].

New instructions. To coordinate between the memory controller and the system software, we add two instructions to 1) support periodic forced writeback, and 2) retrieve and roll back to checkpointed consistent data.

The first instruction is called the *Binary Star cache line writeback instruction* (*bclwb*). Binary Star leverages the *bclwb* instruction to write back all dirty cache lines. Similar to an existing cache line writeback instruction (*clwb*) [166], *bclwb* allows Binary Star to write back

each dirty cache line to NVM without invalidating the cache line. The *bclwb* instruction not only performs the same cache line writeback functionality as *clwb*, but also triggers the memory controller to modify wear leveling metadata to guarantee consistency. During the execution of a *bclwb* instruction, after data writeback, the memory controller marks the current segment as consistent. The memory controller also tracks the old segment with the segment number, frees it by setting the free bit, and adding it to the free list.

The second instruction is called the *Binary Star data reset instruction* (*drst*) and is used to retrieve checkpointed consistent data. *drst* has only one operand: memory address. *drst* is used for handling error-triggered rollback. When the memory controller receives a *drst* command, the memory controller checks the *consistent bit* of the associated segment's memory address to see whether or not the segment is consistent. The memory controller does nothing if the segment is consistent. Otherwise, the memory controller finds the corresponding consistent segment using the stored consistent segment number. The memory controller then invalidates the current segment by setting the free bit, and adding the segment number to the free list.

Reducing latency by controlling NVM writebacks. Because the latency of Binary Star-triggered wear leveling can degrade the performance of a latency-critical application, Binary Star provides two optional optimizations. First, when the NVM bus is idle, Binary Star allows the memory controller to issue preemptive consistent cache writeback to remap a consistent segment in advance. Second, when read requests are delayed by the writebacks, Binary Star allows the memory controller to postpone LLC writebacks, which has been shown to provide performance benefits in a previous study [168].

NVM over-provisioning. NVM preserves a certain amount of physical space specifically for wear leveling purposes [169]. Over-provisioned space guarantees that wear leveling can always find a free segment when the logical space is full. To ensure that Binary Star-triggered wear leveling always finds a free segment, this over-provisioned space has to be large enough to provide a sufficient number of free segments. Our experiments (Section 4.2.2) show that

10% extra space for over-provisioning is sufficient to accommodate a 30-minute periodic forced writeback interval with all of our benchmarks.

4.3.3 System Software Modifications

We implement software support as part of the operating system to leverage the hardware support.

Binary Star daemon. We develop a Binary Star daemon, which is in charge of periodic forced writeback and rollback. Because the periodic forced writeback and rollback are infrequent operations, we assume a single daemon can manage all processes on a Binary Star enabled machine. Figure 4.7 shows the state machine of the daemon. Binary Star daemon can be implemented as a kernel loadable module.

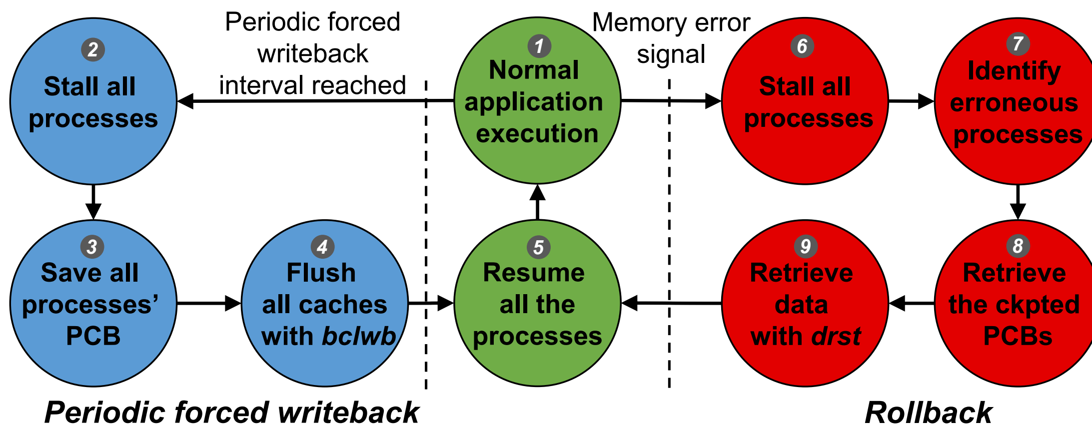


Figure 4.7: State machine of the Binary Star software daemon.

The Binary Star daemon operates in normal mode until either 1) the periodic forced writeback interval is reached, or 2) a memory error signal is received. To determine if the periodic forced writeback interval is reached, we use a simple timer, which consumes minimal CPU resources (➊). When the timer reaches a pre-defined time interval, the daemon sends interrupt signals to stall all the running processes and starts the periodic forced writeback procedure.

The periodic forced writeback procedure consists of five steps. First, the daemon sends

an interrupt to each of the executing processes and stalls them (❷). Second, a copy of each process control block (PCB) is flushed into the memory, so that the process can be restored (❸). Each PCB contains process-related state, such as process ID, stack, and page table. Instead of introducing extra implementation cost, Binary Star leverages the PCB state to rollback a process to the previously written-back checkpoint in the same way as existing designs [170]. Third, the daemon flushes all the dirty caches lines (including SRAM and 3D DRAM caches) to the memory, by using the *bclwb* instruction (❹). Fourth, when all cache line writebacks are completed, the daemon resumes all the stalled processes (❺).

The second key task of the Binary Star daemon is to manage error-triggered rollback. When the daemon receives an unrecoverable memory error signal from the memory controller, it starts the rollback procedure. First, the daemon sends an interrupt to stall all executing processes (❻). Second, the daemon identifies the error-affected processes, inspects page tables of each process and identifies the process that triggered the error (❼). Third, the daemon retrieves all the PCB data, such that the affected processes' PCBs are rolled back to the checkpointed PCBs (❽). Fourth, the daemon retrieves the data of all error-affected processes (❾). The daemon rolls back memory data by retrieving the old page table from the PCB and using the *drst* instruction to fetch previously checkpointed consistent data. The daemon also employs *drst* to free all the remapped inconsistent segments to avoid memory leakage. Finally, the daemon resumes the execution of all the processes (❺).

4.3.4 Binary Star Overheads

Overhead in 3D DRAM Cache. The CRC-32 code requires 32 bits per 64-byte cache line, i.e., 6.25% storage overhead, which is 50% lower than the storage overhead of traditional ECC. The area and performance overheads of CRC encoding and decoding engines are similar to the existing ECC encoding and decoding engines.

Overhead in NVM. Our modification to NVM wear leveling requires three bytes per

segment. With a 512GB NVM and a 1MB segment size, this introduces 0.0003% storage overhead. The free segment list takes 1.1875MB additional space at most. Because NVM typically over-provisions its capacity by roughly 10% to support wear leveling [169], our design reuses the over-provisioning space and, as such, avoids additional storage overhead.

Software overhead. For software support, Binary Star only requires a loadable module to the operating system. An application requires no modification to utilize Binary Star.

Overhead of periodic forced writeback and uncorrectable error-triggered rollback. We test periodic forced writeback and rollback using several workloads on an emulated NVM. Typically, the periodic forced writeback takes 50μs to 3.6 seconds. Because Binary Star rollback reuses data hidden by wear leveling and avoids large amounts of data movement, the rollback takes only dozens of microseconds. The Binary Star daemon has negligible performance impact on the operating system and applications because of the low frequency of periodic forced writeback and the very small likelihood of error-triggered rollback (See Section 4.5.3).

4.4 Experimental Methodology

We evaluate both the performance and reliability of Binary Star by comparing it with four baseline memory configurations: 1) DRAM memory with 3D DRAM LLC (**3D DRAM+DRAM**), 2) DRAM memory without 3D DRAM LLC (**DRAM**), 3) PCM as main memory with 3D DRAM LLC (**3D DRAM+PCM**), and 4) PCM as main memory without 3D DRAM LLC (**PCM**).

For all four memory configurations, we use various state-of-the-art ECC mechanisms, which we describe below. With DRAM, we evaluate both 28nm and sub-20nm process technologies. We use Failures In Time (FIT) as our main reliability metric.

Performance simulation. We evaluate the performance of Binary Star modifications using McSimA+ [171], which is a Pin-based [120] cycle-level multi-core simulator. Our infrastructure models the row buffer as well as row buffer hits and row buffer misses. We use

the FR-FCFS memory scheduling policy [172, 173] for Binary Star and all our baselines. We configure the simulator to model a multi-core out-of-order processor with 3D DRAM LLC and NVM DIMM (for Binary Star) as described in Table 4.1. The higher-level L1 and L2 caches are SRAM-based. We model the 3D DRAM LLC based on the HMC 2.1 specification [174]. We use PCM timing parameters [22] in our basic configuration for the NVM DIMM. We adopt the state-of-the-art NVM error protection design [3] in our evaluations.

Reliability simulation and modeling. We evaluate the reliability of our design using a combination of reliability simulation and theoretical calculation. We employ FaultSim [175], a configurable memory resilience simulator, to compare the reliability and storage overhead of various memory technologies and reliability schemes.

To evaluate the error correction and recovery ability of Binary Star, we calculate the device failure rates using a method proposed in previous work [1]. A device failure occurs when one or more erroneous bits within any 64-bit line of a $\times 8$ device are uncorrectable after the corresponding ECC or correction/recovery mechanisms are applied. We calculate the device failure rate based on the error correction/recovery capability (the number of bits that can be corrected) of each mechanism given the number and distribution of single cell errors [1]. We assume that single cell errors are randomly distributed. We omit column, row, and connection faults for the same reasons discussed in previous studies [1]. We only consider errors that are found after device shipment time [1]. We evaluate various reliability schemes, including no-ECC, RECC only [130], combined RECC (rank-level ECC) and IECC (in-DRAM ECC) [1], combined Chipkill and IECC [1, 4], and Binary Star (which combines CRC-32 and our error correction and recovery techniques described in Section 4.3). For both reliability simulation and theoretical analysis, we adopt PCM and sub-20nm DRAM device reliability parameters published in previous works [159, 1, 175, 2]. We implement 3D DRAM Chipkill with multiple DRAM cubes. For a fair comparison, we use multiple DRAM cubes across all the reliability experiments. However, we note that Binary Star is applicable to both single-cube and multiple-cube 3D DRAM. Because

Table 4.1: Evaluated processor and memory configurations.

Processor	
Cores	4 cores, 3.2 GHz, 2 threads/core, 22nm
IL1 Cache	32KB per core, 8-way, 64B cache lines, 1.6ns latency
DL1 Cache	32KB per core, 8-way, 64B cache lines, 1.6ns latency
L2 Cache	32MB shared, 16-way, 64B cache lines, 4.4ns latency
LLC	3D DRAM, 16 GB, CRC-32 (for Binary Star), tCAS-tRCD-tRP: 8-8-8, tRAS-tRC: 26-34 [146, 174]
Main Memory	
DRAM	512 GB DDR4-2133 timing, 2 128-bit channels, ***ECC schemes***: non-ECC, RECC [130], combined RECC and IECC [1], combined Chipkill and IECC [1, 4]
PCM	512 GB and 52GB over-provisioning space, 36ns row-buffer hit, 100/300ns read/write row-buffer conflict [22], 2 channels, 128 bits per channel, Single error correction, double error detection (SECDED) + NVM remap [3]

3D DRAM does not have ranks, RECC on 3D DRAM is the same as ECC on a single-cube as in various previous works [176, 147].

Benchmarks. We evaluate our design with various data-intensive workloads. Redis [177] and Memcached [178] are in-memory key-value store systems widely used in web applications. Our Redis benchmark has one million keys in total and simulates the case when Redis is used as an LRU cache; Our Memcached benchmark has 100K ops (read/write), with 5% write/update operations. TPC-C [179] is a popular on-line transaction processing (OLTP) benchmark; we run 400K transactions, mix of reads and updates (40%). YCSB [180] is an open-source specification and program suite for evaluating retrieval and maintenance capabilities of computer programs; we run eight million transactions, randomly performing inserts, updates, reads, and scans. MySQL [46] is one of the most widely used relational database management systems (RDBMSs), which is typically used in online transaction processing; we use a table of 10 million tuples and conduct complex OLTP operations (using sysbench [181]), including point, range, update, delete, and insert queries. We exercise these server workloads using four clients and scale the memory footprint. We also evaluate Binary Star with several memory-intensive benchmarks

from the PARSEC 3.0 benchmark suite [182], including *facesim*, *freqmine*, *fluidanimate*, *vips*, and *x264*. We configure the dataset to be the same as the capacity of our main memory configuration (512GB) on all the benchmarks.

We evaluate application performance using instruction throughput (instructions per cycle) for PARSEC 3.0 [182] benchmarks and operational throughput (the number of executed application-level operations, e.g., inserting or deleting data structure nodes, per second, not including logging) for all other benchmarks.

4.5 Results

This section presents the results of our Binary Star evaluation. Section 4.5.1 shows Binary Star improves reliability over various error correction baselines. Section 4.5.2 shows the performance of Binary Star compared to multiple memory configurations. Section 4.5.3 presents a study of the impact of Binary Star's periodic forced writeback interval on various metrics.

4.5.1 Reliability

Figure 4.8 shows the projected 3D DRAM device failure rates of different resilience schemes as the single cell bit error rate varies from 10^{-14} to 10^{-4}. The x-axis represents the bit error rate (BER) of a single cell and the y-axis is the corresponding device failure rate. We make three major observations. First, for sub-20nm DRAM with 10^{-5} single cell bit error rate [1], Binary Star reduces the device failure rate by 10^{12} times compared to IECC with Chipkill [1] and by 10^{16} times compared to IECC with RECC [1]. Second, even when the sub-20nm single cell error rate is lower (i.e., 10^{-6}), Binary Star still provides better reliability than other schemes: Binary Star reduces the device failure rate by 10^{16} times compared to IECC with Chipkill, and by 10^{22} times compared to IECC with RECC. These results indicate that Binary Star is effective for technologies that exhibit lower error rates than those projected in [1]. Third, compared to the

commonly-employed RECC DRAM, Binary Star greatly lowers the device failure rate on both 28nm and sub-20nm technology nodes (i.e., at 10^{-10} and 10^{-15} single cell BERs). We conclude that Binary Star provides strong error protection on 3D DRAM LLCs, regardless of technology node and single cell error rate.

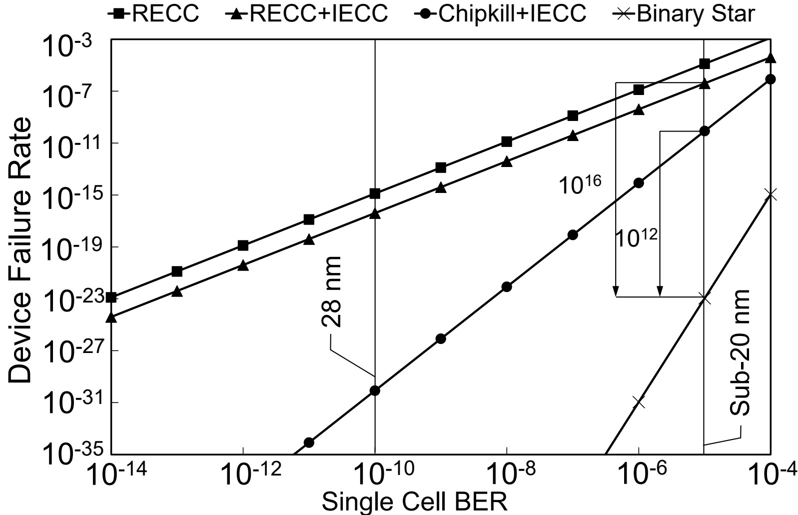


Figure 4.8: Device failure rates of 3D DRAM for traditional RECC [3], RECC with IECC [1], Binary Star, and Chipkill with IECC [1, 4] vs. single cell bit error rate. Vertical lines represent 3D DRAM technology nodes (28nm and sub-20nm).[5]

Table 4.2 summarizes the FIT and storage cost of five state-of-the-art baselines and Binary Star. We show results when the device single cell BER is 10^{-6} (left) and 10^{-5} (right). We make three observations. First, Binary Star provides a clear advantage against all state-of-the-art baselines. Compared to IECC+Chipkill, which is the best IECC based scheme, Binary Star reduces FIT by 14.4×. Compared to the IECC+RECC protected sub-20nm DRAM system, Binary Star reduces FIT by 29.7×. Second, compared to the NVM based main memory system that uses RECC 3D DRAM cache and PCM, Binary Star experiences 58.5% fewer failures. Third, Binary Star incurs lower space overhead compared to all other state-of-the-art designs, with only 6.25% additional space in 3D DRAM and only 12.79% additional space in PCM, respectively. We conclude that Binary Star is the most efficient reliability mechanism for hybrid emerging

[5]The sub-20nm single bit error rate is based on [1]. The single cell error rates of future sub-20nm products can vary and depend on the specific technologies.

technologies of 3D DRAM cache and NVM main memory.

Table 4.2: Comparison of the evaluated resilience schemes.

System	FIT	Storage cost	
		DRAM LLC	Main memory
No-ECC [2] 28nm DRAM	44032-66150	N/A	0%
RECC [183] 28nm DRAM	8806-13230	N/A	12.5%
IECC+RECC [1] sub-20nm 3D DRAM + DRAM	78211-117912	18.75%	18.75%
IECC+Chipkill [1, 4] sub-20nm 3D DRAM + DRAM	37949-59518	18.75%	18.75%
RECC [183, 3] sub-20nm 3D DRAM + PCM	6352-9963	12.5%	12.79%
Binary Star sub-20nm 3D DRAM + PCM	2637-3968	6.25%	12.79%

4.5.2 Performance

Figure 4.9 shows the performance of three baseline memory configurations, all of which have *no ECC*, and Binary Star normalized to the baseline with 3D DRAM cache and DRAM main memory.[6] We make two observations based on this data. First, Binary Star's performance is very close to the baseline with 3D DRAM. Binary Star performs within 0.2% of a baseline with 3D DRAM and PCM Binary Star also performs within 1% of the performance of the baseline with 3D DRAM cache and DRAM main memory. Second, we observe that using 3D DRAM as the LLC to PCM main memory improves performance by 10% over the PCM-only baseline. We observe that the performance improvements are significant on workloads that exhibit high LLC locality (e.g., memcached, facesim, vips and x264). In summary, we find that Binary Star provides

[6]We include the CRC encoding and decoding performance overheads based on [1]. We normalize the throughput of each configuration to the throughput of 3D DRAM LLC with DRAM. We find that the error detection and recovery overheads are negligible compared to the CRC encoding and decoding overheads, because errors happen rarely (See Section 4.5.1).

comparable performance to the baseline with 3D DRAM cache and DRAM main memory but at significantly higher reliability.

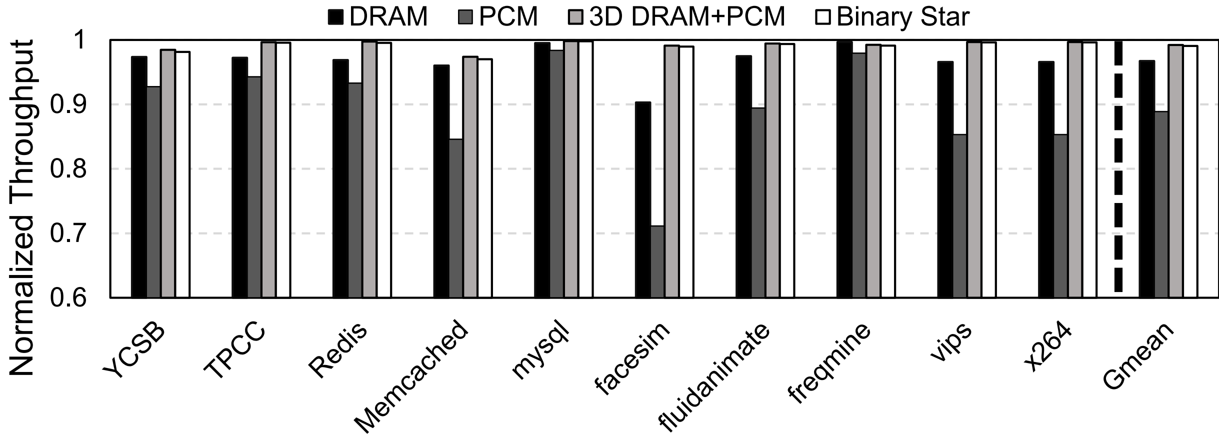


Figure 4.9: System performance of different memory configurations with no ECC and Binary Star. We normalize the throughput of each configuration to the throughput of 3D DRAM LLC with DRAM.

Effect of NVM wear-out. To understand the effect of NVM wear-out-induced latency on performance, Figure 4.10 provides the performance of each workload with different NVM access latencies. In this experiment, we vary the latency of PCM to 0.5×, 1×, 2× and 4× of the default PCM latency provided in Table 4.1. All throughput values are normalized to the highest performance memory configuration (3D DRAM+DRAM). As Figure 4.10 shows, Binary Star retains 99% of the performance of the baseline with a 0.5× PCM latency and its average

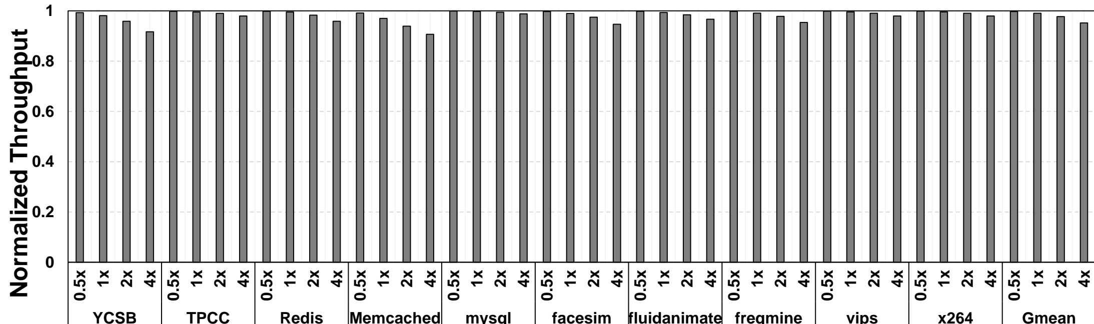


Figure 4.10: Performance of Binary Star normalized to the performance of the 3D DRAM +DRAM configuration, as PCM latency is varied as a multiple of its value in Table 4.1.

performance degradation is only 3% with a 2$\times$ PCM access latency. In the unrealistic case where the PCM access latency is quadrupled (4$\times$), the average performance degradation of Binary Star is only 5% lower than the baseline.

4.5.3 Impact of the Periodic Forced Writeback Interval

Binary Star's periodic forced writeback interval is critical to throughput, number of rollbacks triggered by uncorrectable errors, NVM lifetime, and capacity.

Throughput. Figure 4.11 (a) shows the normalized system throughput of different workloads for six different periodic forced writeback intervals, ranging from one second to one hour. We normalize the throughput of each workload to the throughput of the same workload under the 3D DRAM LLC with DRAM configuration. We make two observations. First, throughput decreases when the interval decreases. As the periodic forced writeback interval decreases, both periodic forced writeback and consistent cache writeback happen more often and occupy more NVM bandwidth, leading to reduction in throughput. Second, the periodic forced writeback interval impacts the efficiency of consistent cache writeback. As the interval decreases, the locality of NVM access reduces such that Binary Star-triggered wear leveling happens more frequently. Therefore, Binary Star triggered wear leveling overlaps much less with endurance-triggered wear leveling, leading to throughput loss.

Rollback Triggered by Uncorrectable Errors. Figure 4.11 (b) shows the rollback rate for six different intervals. Rollback is triggered by uncorrectable errors that occur when accessing dirty cache lines in 3D DRAM. As such, the results also reflect the error rate on dirty cache lines. As the forced writeback interval decreases, dirty cache lines remain in LLC for less time. Therefore, the possibility of uncorrectable-error-triggered rollback reduces. When the interval decreases to one second, the likelihood of error-triggered rollback is $< 10^{-11}$. However, the throughput loss is too much in this case, as shown in in Figure 4.11 (a). At a 1800-second

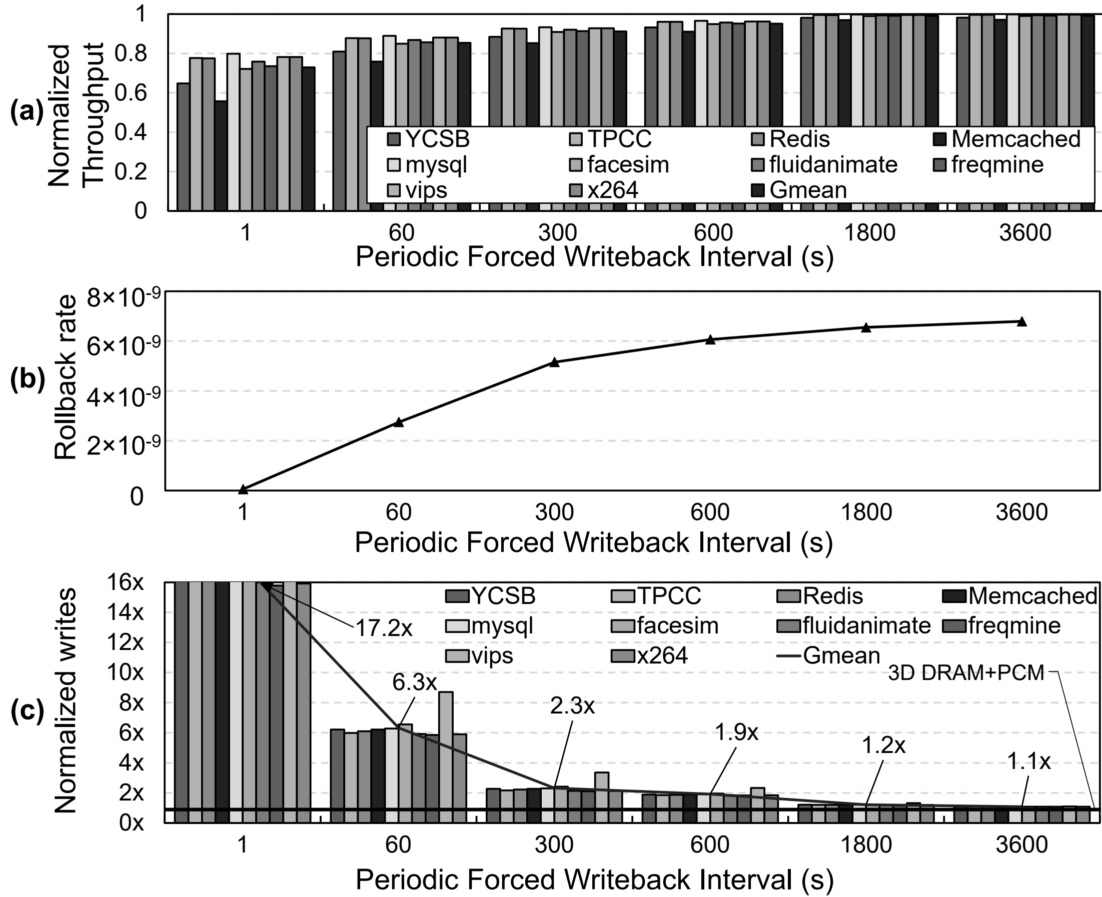


Figure 4.11: Effect of varying the periodic forced writeback interval on (a) throughput, (b) probability of rollbacks (i.e., error rate on a dirty line), and (c) NVM writes.

(30-minute) periodic forced writeback interval, the rollback rate is 10^{-9}, which is close to the error rate of current DRAM without ECC.

Endurance. Figure 4.11 (c) shows the normalized number of NVM writes for six different periodic forced writeback intervals, normalized to the NVM writes in the 3D DRAM+PCM baseline for each benchmark. We make three observations. First, the number of writes increases drastically to 17.2× of the PCM-only baseline when the interval is only one second. Such a short interval leads to a low likelihood of rollback, but can cause up to 94% faster NVM wear-out due to the larger number of writes. Second, when the interval gets larger, the number of NVM writes gradually becomes close to that in the 3D DRAM+PCM baseline. Third, the increase in

the interval length has diminishing benefit. We find that increasing the interval from 30 minutes to one hour only leads to a 10% reduction in the number of NVM writes, but forces applications to roll back consistent data from earlier in the past (up to one hour instead of up to 30 minutes) when an uncorrectable error happens, leading to high re-execution overheads.

Lifetime impact of coordinated wear leveling. We use the same methodology as previous work [160] to evaluate the NVM lifetime impact of our design. We determine the end of NVM lifetime as the time when NVM capacity drops below 90% of the original capacity (i.e., when the over-provisioned space is saturated). We make two observations when the periodic forced writeback interval is 30 minutes (1800 seconds). First, Binary Star triggers 20% additional NVM writes, compared to the 3D DRAM+PCM configuration (Figure 4.11 (c)). This leads to an 8.4% NVM lifetime reduction compared with the baseline wear leveling mechanism [119]. As the baseline wear leveling mechanism already offers $10\times$ NVM lifetime improvement on native NVM without wear leveling [119], our design provides $9.2\times$ better lifetime than native NVM, and our coordinated wear leveling mechanism has minimal impact on NVM lifetime. Second, using 3D DRAM as LLC significantly prolongs the lifetime, because it reduces writes to NVM by 90.1% in the NVM baseline and 87.8% in Binary Star, respectively.

Capacity. One disadvantage of consistent cache writeback on NVM is that Binary Star gradually consumes more space over time because the consistent cache writeback triggered block remapping consumes free blocks and holds consistent blocks. To free up space for incoming remapping operations, periodic forced writeback releases space for previous checkpointed consistent data (as discussed in Section 4.3.2). For the above experiments, we guarantee that all the workloads have sufficient memory space while keeping 10% of the NVM space for wear leveling.

Overall, the periodic forced writeback interval length causes a trade off between performance, reliability, NVM endurance and the effective NVM capacity. In this work, we choose 30 minutes (1800s) as the interval, such that periodic forced writeback only incurs less than 1% average throughput loss, with only a 10^{-9} chance of rollback and 5 to 7.5 FIT per GB of NVM.

4.6 Conclusion

We propose Binary Star, the first coordinated memory hierarchy reliability scheme that achieves both high reliability and high performance with a 3D DRAM last-level cache and nonvolatile main memory. Binary Star coordinates 1) the reliability mechanisms between the last-level cache and main memory, leveraging consistent cache writeback and periodic forced writeback to the main memory to allow the elimination of ECC in the last-level cache, and 2) the wear leveling scheme in main memory with consistent cache writeback to significantly reduce the performance and storage overhead of consistent cache writeback. As a result, we can eliminate the error correction codes in the last-level cache, while achieving higher reliability than using costly sophisticated ECC mechanisms. Our work rethinks the reliability support in the memory hierarchy in light of the next-generation computing systems that may adopt two key new memory technologies, i.e., 3D DRAM caches and byte-addressable nonvolatile memories. We hopefully provide a critical step in reaping the full reliability advantages of these emerging technologies beyond simply replacing the existing memory technologies.

Acknowledgments

This chapter contains material from “Binary Star: Coordinated Reliability in Heterogeneous Memory Systems for High Performance and Scalability.”, by Xiao Liu, David Roberts, Rachata Ausavarungnirun, Onur Mutlu, and Jishen Zhao, which appears in the Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 52). The dissertation author is the primary investigator and first author of this paper.

Chapter 5

HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing

Neuromorphic computing has attracted increasing attentions today because the applicable scenarios of the neural network continue to expand. Convolutional neural networks (CNNs) are widely adopted because of their high prediction accuracy. As network sizes grow, general-purpose hardware platforms, such as CPU and GPU, are insufficient for delivering decent performance and power-efficiency for neuromorphic computing applications. There have been a wide variety of neuromorphic computing accelerators, which utilize fast and power-efficient emerging technology to accelerate large-scale CNNs.

Resistive RAM (RRAM), or memristor, have been widely investigated to serve as a CNN accelerator [74]. RRAM can form a crossbar array to perform matrix multiplication by exploiting analog characteristics of RRAM cells. Matrix multiplications are the majority of operations in convolution. Optimizing matrix multiplications is essential to improving the performance of CNN. In a RRAM based CNN accelerator, weights of the matrix are programmed into the conductance of each RRAM cell in each crossbar. The input data is converted to analog signals and transferred to crossbar bitlines. The output data is generated and converted to digital

signals on wordlines. Matrix multiplication achieves higher speed and parallelism on RRAM crossbars than conventional hardware [184]. RRAM crossbars also have cost and energy efficiency advantages [74].

However, the thermal issue of RRAM crossbars potentially reduces the inference accuracy of neural networks. Prior work discovered RRAM conductance is sensitive to temperature [185]. RRAM ON state conductance (G_{ON}) and OFF state conductance (G_{OFF}) varies as the temperature changes. The conductance range $[G_{OFF}, G_{ON}]$ sharply declines on the hot cells. This variation drastically decreases weight precision when it is represented in the form of conductance. When the chip continuously operates, the accumulated heat affects more RRAM cells. This causes many weights to be misrepresented during inference. Eventually, neural networks suffer from loss of accuracy. Our experiments show that the accuracy loss at high temperature can be as high as 90%.

A few research paid attention to the problem. Most existing works focus on handling device defects and variations in RRAM crossbars [186, 187, 188]. However, because these schemes are based on permanent errors from process variation, they do not dynamically adjust to the defected cells. Temperature aware row adjustment (TARA) [189] proposed a heat resilient design but suffers from several major drawbacks. TARA still needs to be aware of the exact weights and adjusts row order based on each weight's magnitude, which is not scalable with large scale networks. Besides, TARA provides no optimization over thermal issue. As network size expands, the thermal issue gets worse and row adjustment becomes less effective. In this dissertation, we introduce a novel heat resilient design HR^{3}AM for the RRAM crossbar. HR^{3}AM solves the issue from two aspects: adapting weights to the reduced conductance and optimizing chip thermal distribution. Therefore we correspondingly propose two mechanisms in HR^{3}AM from these two aspects. The first mechanism *bitwidth downgrading* aims to improve accuracy. It adapts weights to the decreased conductance range by reducing bitwidth per RRAM cell. The second mechanism *tile pairing* aims to tackle the thermal issue. It optimizes the chip thermal

status by matching hot spot tiles with idle tiles. The contributions of this dissertation include:

- We evaluate the impact of heat on the inference accuracy of several large scale neural networks. Our results show the inference accuracy of several distinguished CNNs drop to less than 10% of theoretical accuracy.

- We propose HR^3AM, which contains two mechanisms: *bitwidth downgrading* and *tile pairing* to improve inference accuracy and tackle the thermal issue, respectively. Bitwidth downgrading increases the accuracy of weights represented in conductance. Tile pairing releases the heat on hot spot tiles.

- We evaluated HR^3AM with four state-of-the-art CNN models. The results show our design guarantees 87.8% of the theoretical inference accuracy. Our design also shows a 6.2K decrease on maximum temperature and a 6K decrease on average temperature for the entire chip.

5.1 Background and Motivation

5.1.1 Neural Network Data Mapping on RRAM crossbar

The neural network consists of various types of layers. Most of layers in a deep neural networks are convolutional layers (e.g. 13 out of 16 in VGG16), which conduct a large number of matrix multiplications. Utilizing RRAM crossbar through matrix multiplications in the neural network can increase parallelism and decrease latency. Encoding a weight value into the RRAM crossbar requires conversion between the value and the cell conductance, which follows the formula [184], where $\alpha = \frac{G_{max}-G_{min}}{w_{max}-w_{min}}$ and $\beta = G_{max} - \alpha * w_{max}$:

$$G = \alpha * W + \beta \tag{5.1}$$

α linearly scales weights to match range of conductance and β adds the offset to remove negative values in weights. The precision of weights is limited by the bitwidth of RRAM cells. Single cell only achieves at most seven bits accuracy [190]. To achieve higher precision, each weight can be programmed into multiple RRAM cells [74].

Each layer of the neural network is serialized. Therefore we can only perform parallel operations within the single neural network layer. Each layer of the neural network should be mapped to a group of RRAM tiles [74]. Because tiles belonging to the different layers are independent, layers can form a pipeline to achieve better throughput [74]. The pipelined RRAM crossbar chip processes multiple inputs concurrently with different layers.

5.1.2 Thermal Issues in RRAM Accelerator

RRAM cells originally obtain ON and OFF states. Each RRAM cell can represent multiple bits by setting an intermediate state that has the conductance between ON and OFF states [184]. Each status has its unique conductance range that does not overlap with others. However, temperature changes has a significant impact on the RRAM conductance between OFF and ON states [191]. Increased temperature leads to a narrower range of conductance.

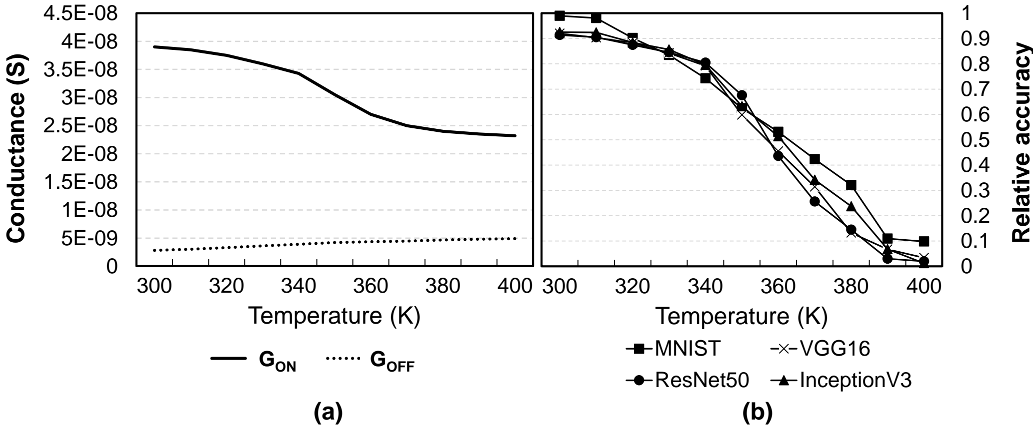


Figure 5.1: Temperature impact on (a) RRAM cell conductance and (b) CNN applications relative inference accuracy.

Figure 5.1(a) depicts this effect on OFF and ON conductance (G_{OFF} and G_{ON}). The

ON state has a weak metallic-like characteristic [191] such that G_{ON} significantly drops as the temperature rises. While G_{OFF} increases with temperature because of increased current [191]. The conductance range $[G_{OFF}, G_{ON}]$ drops by 50% when the temperature rises from 300K to 400K. The conductance range starts to drop sharply after 330K, which is a common operational temperature for many chips. We observe the asymmetry of conductance variation: G_{ON} drops rapidly while the G_{OFF} increases slowly.

The shifting conductance leads to weight misrepresentation of the neural network. The weight conversion (Equation 5.1) between weights and RRAM conductance is based on ideal condition at room temperature. When the temperature changes, certain weight values fall into unavailable conductance range, the weight conversion maps these weight values to the nearest available conductance. These weight values are misrepresented and share conductance with others. We modeled the effect and tested on several large scale neural networks [192, 193]. Figure 5.1(b) shows the relative reference accuracy of four neural networks under 300K to 400K. To get relative accuracy, we normalize the ReRAM generated accuracy to the software generated accuracy. We assume entire chip is effected by the same temperature. The accuracy drops as the temperature raises. Despite minor variance between different network models, the inference accuracy of all experiments eventually drop below 10% at 400K.

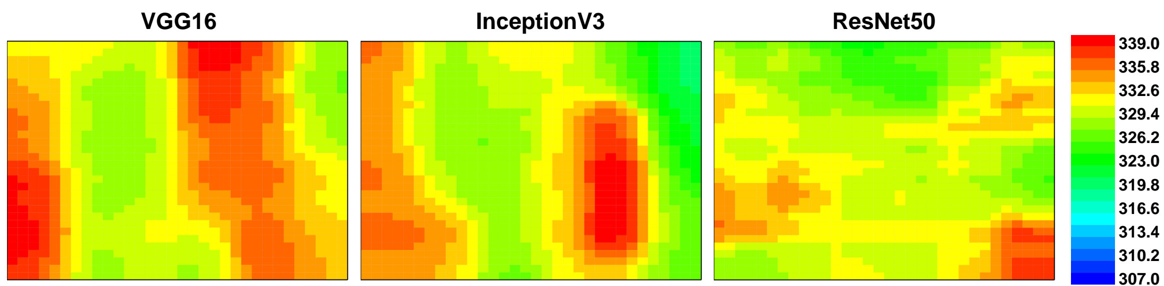


Figure 5.2: Temperature distribution of a single RRAM chip when running VGG16, InceptionV3 and ResNet50.

When pipelining multiple CNN applications, each application runs in different layers during runtime. Because different CNN layers have various sizes, the amount of data and matrix

multiplication operations conducted by different neural network layers varies. When each layer is bonded to certain groups of tiles on the chip, power consumption varies between tiles and dynamically change with time, which may cause a non-uniform thermal distribution on RRAM chip. We generate the steady state temperature distributions of entire RRAM chip with three CNN models (VGG16, InceptionV3, ResNet50) conducting inference for 10000 ImageNet [194] figures. As Figure 5.2 shows, the temperature difference can be as high as 17.16K. The distribution also shows several distinctive hot spots. Design-time mechanisms are not sufficient to solve such dynamic issues. Furthermore, identifying hot spot tiles and mitigating heat on hot spot tiles could optimize the thermal status. In the next section, we propose two mechanisms, one can dynamically mitigate the negative impact of overheating, and the other can reduce hotspots on the RRAM chip.

5.2 HR^{3}AM Design

5.2.1 HR^{3}AM Overview

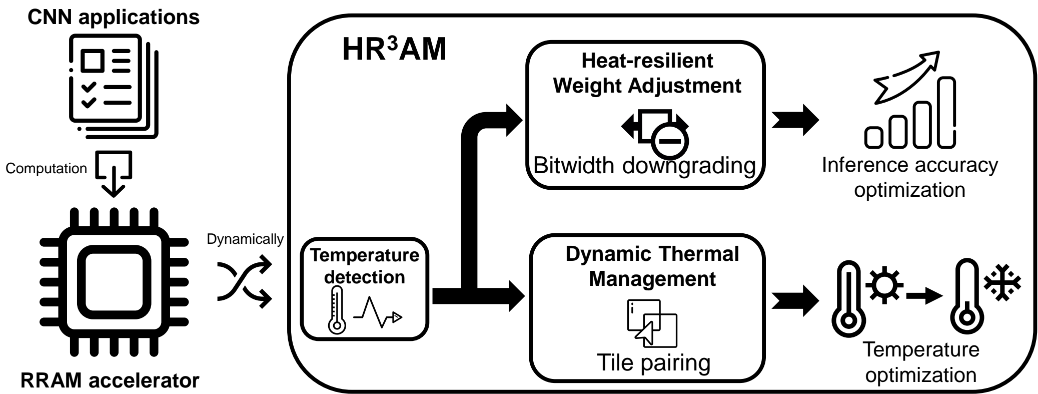


Figure 5.3: Overview of HR^{3}AM structure.

To handle thermal issue imposed inference accuracy loss, we propose HR^{3}AM, a heat resilient design for RRAM based CNN accelerators. Figure 5.3 shows an overview of HR^{3}AM. When running CNN applications, HR^{3}AM monitors the dynamic thermal distribution of the

RRAM chip. We adopt the temperature sensor design in commercial processors [195], which provides temperature detection of each crossbar unit. HR^3AM provides two heat resilient methods, heat-resilient weight adjustment and dynamic thermal management, to optimize inference accuracy and temperature distribution respectively. For heat-resilient weight adjustment, we introduce a mechanism, called *bitwidth downgrading*, to dynamically change the bitwidth of RRAM cells. Downgrading bitwidth of hot RRAM cells would trade the precision of influenced weights for mis-representation of weights caused by conductance range reduction. For dynamic thermal management, we propose the *tile pairing* to move a portion of operations in hot tiles to pre-allocated idle tiles. Hence, the power consumption of hot tiles would be reduced to release heat. The following of this section discusses details of these two methods.

5.2.2 Heat-resilient Weight Adjustment

Bitwidth Downgrading. As we discussed in Section 5.1.2, the major cause of accuracy decline is the conductance range drop when temperature rises. Our evaluation shows CNN applications have 0.9% inference accuracy loss on average for every 1K increase on temperature. To accommodate the shifting conductance range, we modify β and α of Equation 5.1. Based on the observation that G_{OFF} changes much less than G_{ON}, we can get $G_{OFF}^{new} \approx G_{OFF}^{old}$. When temperature rises, the new formula can be calculated with a new coefficient γ:

$$G_{new} = \gamma * (\alpha * W + \beta) \tag{5.2}$$

where $\gamma \approx \frac{G_{ON}^{new}}{G_{ON}^{old}}$. However, the RRAM conductance range, which is divided into multiple levels to represent all the bits states, are non-adjustable. The conductance shrinkage causes the weight value mapped to a high conductance to be misrepresented as a lower conductance. Instead of directly using these misrepresented values, we propose to downgrade the bitwidth such that converted weight can be properly mapped into available conductance range. Bitwidth downgrading acts as γ

in Equation 5.2. When both weight W and parameter β shift right N bits, the mechanism changes the conversion equation to: $G_{new} = 1/2^N * (\alpha * W + \beta)$, where N represents the number of shifted bits.

Our design also accommodates the multiplication result produced by adjusting weights to ensure output correctness. We directly shift N bits back on the result using existing shift-and-add units. The new formula for output voltage when the bitwidth downgrading effects is as follow: $V_O = V_I^T * G_{new} * R_S * 2^N = (V_I^T * R_S * G_{old}/2^N) * 2^N$. The disadvantage of this mechanism is the loss of weight accuracy. However, bitwidth downgrading uniformly and linearly changes weights. Therefore the ratios between different synapses' weights are unchanged such that calculations using bitwidth downgrading provide better accuracy than those using misrepresented values. As we show in Section 5.3, bitwidth downgrading causes much less inference accuracy loss than that of temperature induced conductance changes.

Hardware Support. Figure 5.4 presents the hardware design for bitwidth downgrading. We add a temperature register, a comparator, a *downgrade bit* for each crossbar array and a control circuit. The temperature register stores the most recent sensed temperature of the crossbar array. The comparator checks if the current temperature of the crossbar array exceeds the threshold. The control signal from comparator updates the downgrade bit to notify the crossbar to conduct bitwdith downgrading. The weight encoding hardware reprograms weights to the crossbar cells. We slightly modify weight encoding logic such that the weights shift N bits left before encoding when the downgrade bit is set. Therefore the encoded conductance is $\frac{1}{2^N}$ of the original. The adjustment on the output only requires minor modification on the shift-and-add unit. The control signal from the downgrade bit forces the shift-and-add unit to shift N bits right, such that the final result expands 2^N times and matches with the output of no bitwidth downgrading. The implementation cost consists of a temperature register, a comparator, a downgrade bit and its control logic, and modification to the encoding logic to support bit shifting for each crossbar array. These are either small size storage or simple logic circuits.

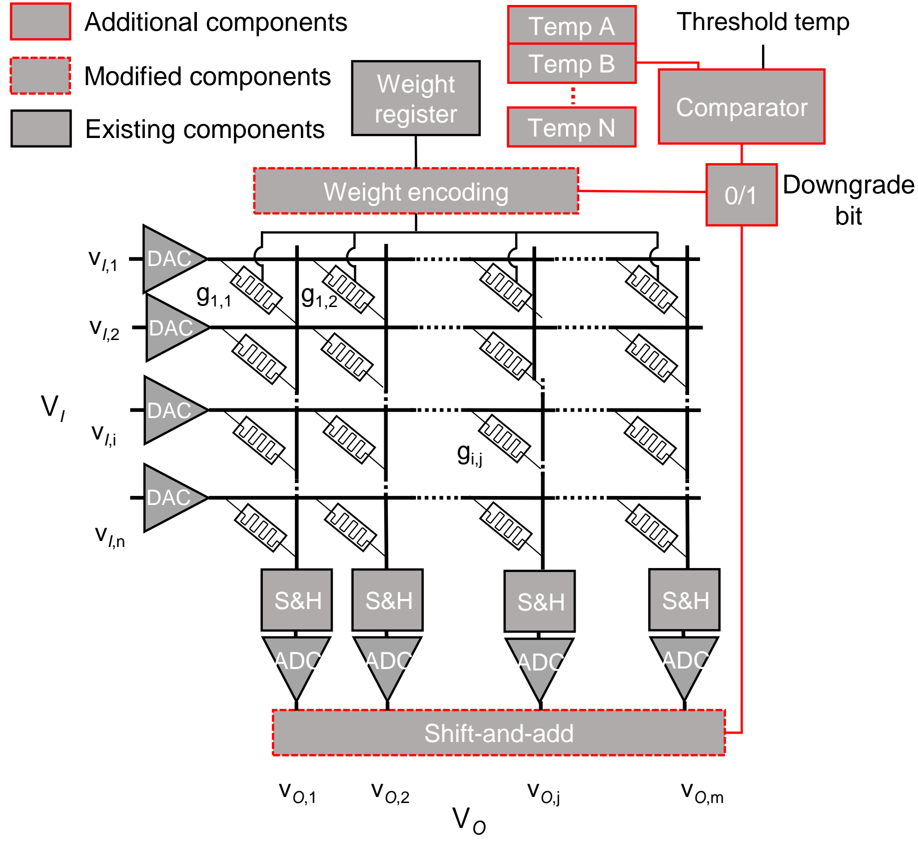


Figure 5.4: Bitwidth downgrading hardware in the crossbar array.

Bitwidth downgrading procedure. Figure 5.5 presents the flowchart of the bitwidth downgrading mechanism on a crossbar array. At step ❶, the comparator updates the downgrade bit based on the current temperature. Step ❷ checks if the status of the downgrade bit changes, if it is unchanged the crossbar array directly starts multiplication by going to step 3b, while a true state leads to step 3a (bitwidth downgrading/restoration). There are two cases in step 3a. When the downgrade bit makes $0 \rightarrow 1$ change, the crossbar array recomputes conductance with N bit shifted weights and encode them to RRAM cells. When the bit makes $1 \rightarrow 0$ change, the crossbar array conducts weight restoration: it loads original weights and encodes them to RRAM cells. Step 3b conducts matrix multiplication within the crossbar. Step ❹ checks the downgrade bit. A true state leads to step ❺, the crossbar array shifts N bits back on the output using the shift-and-add unit. The final result is forwarded to tile buffer.

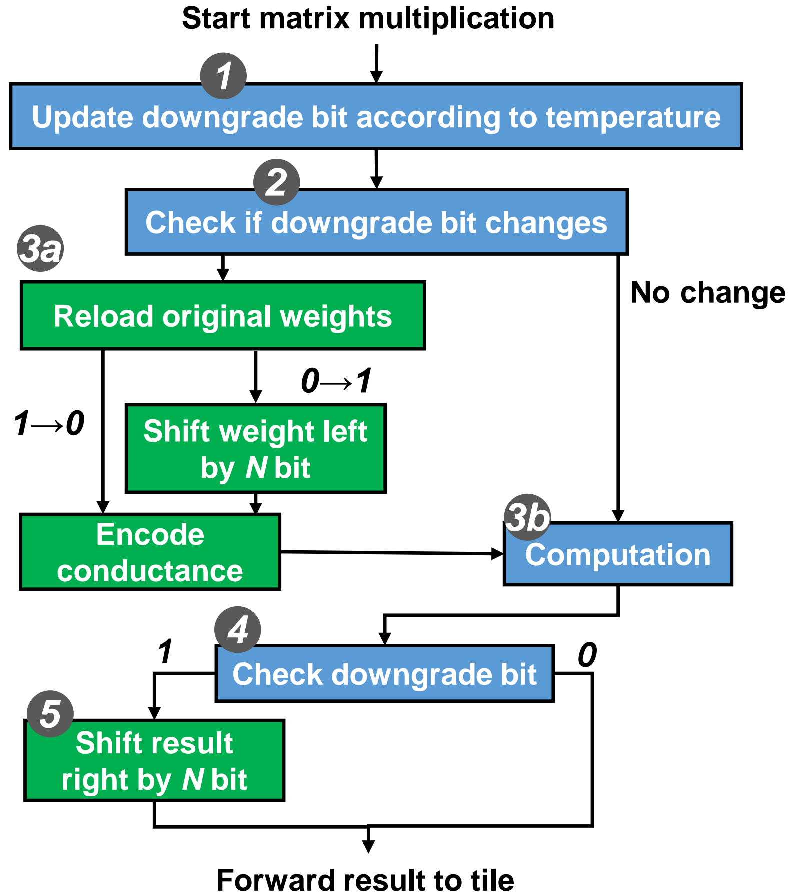


Figure 5.5: Bitwidth downgrading operation flowchart.

The bitwidth downgrading mechanism only effects inside the crossbar array. It requires neither the coordination between the crossbar array and the MAC, nor the coordination between bitwidth downgraded crossbar arrays. The downgraded weights are generated temporarily and avoid overwriting the original copy. We only shift one bit to match the shifted conductance in the worst case (400K) as we observed in Section 5.1.2. Therefore, we set N=1 across the entire design. We choose the temperature threshold of 330K because we observe a significantly increased accuracy decline rate after 330K on Figure 5.1.

5.2.3 Dynamic Thermal Management

Tile Pairing. As we demonstrate in Section 5.1.2, thermal distribution is not uniform on the RRAM chip. Temporally reducing power consumption on the hot spot tiles may improve the thermal status. However, conventional power saving techniques such as decreasing frequency requires complicated design and significantly decreases performance. Therefore, we propose a novel approach — tile pairing to match an overheated tile with an idle tile, while both paired tiles are working at low power mode. In the low power mode, every other row in one crossbar array is activated, such that only half of the cells on a crossbar array are functioning. Based on [74], a low power mode tile only consumes 61.8% of the power of a full power mode tile. We pair overheated units at the tile level. Because pairing at a finer granularity (MAC or crossbar array) leads to overhead for tracking paired units.

We identify an overheated tile and a cooled-down tile with collected temperature statistics. When 80% of the crossbar arrays within an unpaired tile reach the threshold, we identify the tile as overheated. When the percentage of the overheated crossbar arrays in a paired tile drops below 40%, we identify the tile as cooled-down. A cooled-down tile unpairs with its paired tile. The paired tile resumes as an idle tile.

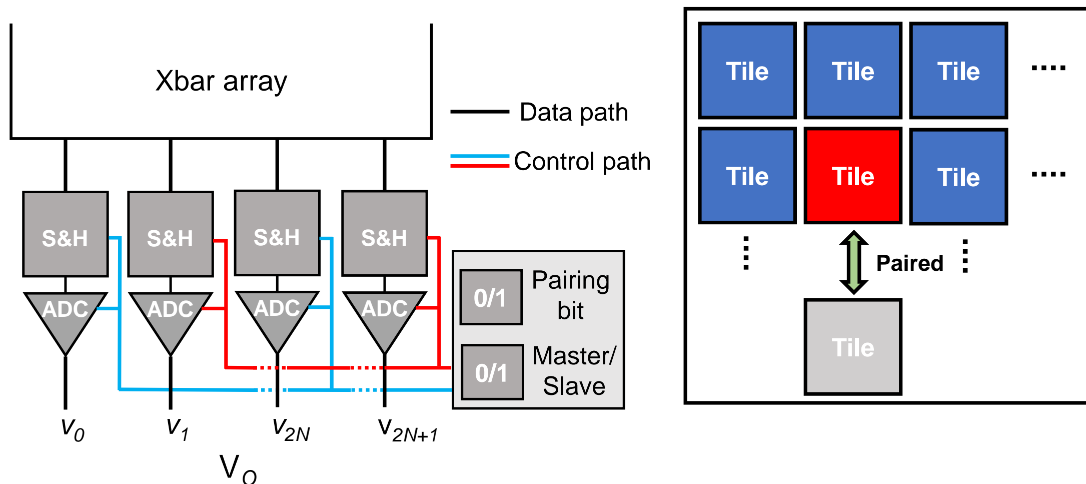


Figure 5.6: Tile pairing (a) mechanism and (b) control logic.

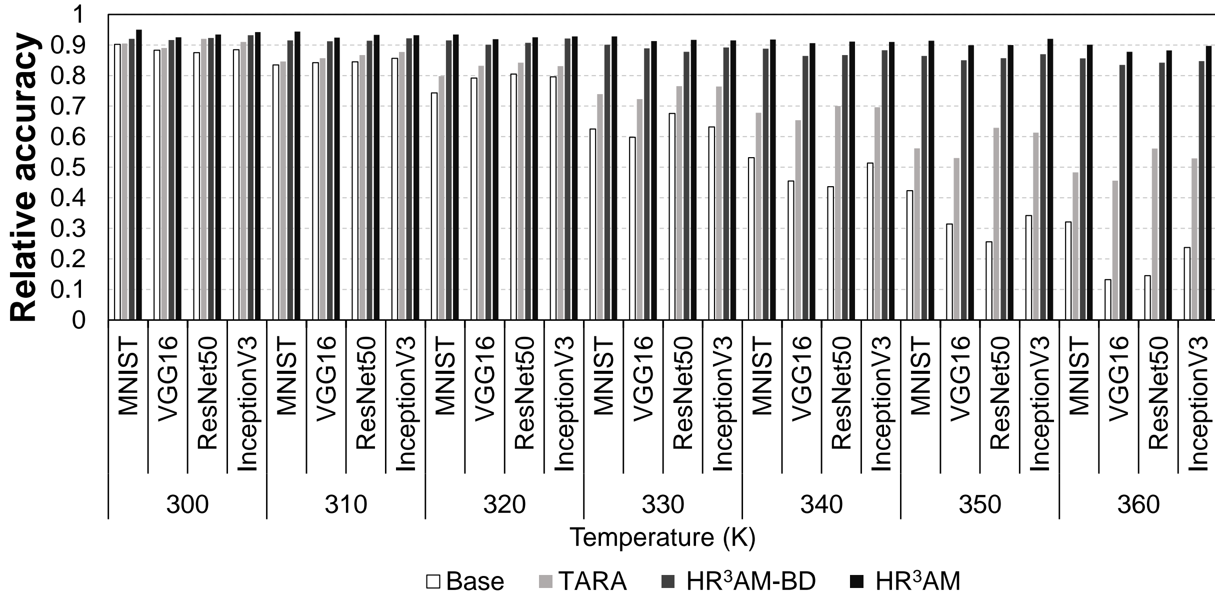


Figure 5.7: Inference accuracy of four neural network models.

Figure 5.6(a) demonstrates the paring mechanism. When an overheated tile pairs with an idle tile, the hot tile is the master tile and the idle tile is the slave tile. Every crossbar array in the master tile pairs with a crossbar array with same index in the slave tile. Both master and slave tiles use half of their cells to produce the result. Two crossbar arrays use the same weights G and same input V_I. The master tile array uses even-index columns and the master tile array uses odd-index columns. The master tile array produces output $V_O^m = \{v_0, v_2 ... v_{2N}\}$ and the slave tile array produces output $V_O^s = \{v_1, v_3 ... v_{2N+1}\}$. The final result is the union of the two outputs:$V_O = V_O^m \cap V_O^s$, which is constant with result generated from an unpaired crossbar.

To scatter a matrix multiplication to two tiles, we rearrange the data placement between master and slave tiles. During pairing, the controller transfers the weight matrix from weight memory of the master tile to the slave tile's. During multiplication, the master and the slave tile share the input data. The data stored in eDRAM vaults are transferred through routers. After calculation, master tile combines outputs of two tiles.

Hardware Support. As Figure 5.6(b) shows, we put additional control logic for each RRAM crossbar to support the low power mode. We add a *paring bit* and a *master/salve bit* to

indicate the status of the tile. The crossbar array acts as normal when the pairing bit is disabled. When the pairing bit is set, the hot tile is set as master with master/slave bit. The idle unit is set as the slave with the same bit. On the master tile, the control logic only enables $2N$ index columns, S&Hs and ADCs are disabled accordingly as well. The slave tile only enables $2N+1$ index columns.

State-of-the-art RRAM architecture [74] does not support dynamic scheduling of tiles. When the pipeline design allocates all the tiles to all the layers, certain tiles are idle during inference but are still bonded to a network layer. Therefore, we reserve several tiles as idle tiles and keep them from being assigned to any network layer. The reserved tiles potentially hurt chip performance due to loss of parallelism. Our performance overhead evaluation in Section 5.3.4 shows throughput decreases only 2.1% on average.

5.3 Evaluation

5.3.1 Methodology

Our RRAM chip setup is based on a state-of-the-art architecture [74]. Our chip consists of four layers of eDRAM and an RRAM layer. The chip operates at 1.2GHz. We build our own RRAM based neural network accelerator simulator. Our simulator models the RRAM conductance variation between 300K to 400K based on [185]. The simulator updates conductance according to the temperature per 100ms at the crossbar array granularity. Our simulator obtains chip temperatures dynamically from HotSpot [196]. The thermal characteristics of the HMC architecture is obtained from previous work [197]. Our thermal simulation uses a single RRAM cycle (100ns) as the time step as described in Section 5.2.1.

Our simulator coordinates with the Tensorflow framework [198] such that it evaluates any Tensorflow based model on RRAM crossbars. We evaluate our design with a small scale two-layer neural network for MNIST handwritten classification [199] and large scale neural

networks (VGG16, ResNet50, and InceptionV3) for ImageNet [200] classification. We use a set of 60000 cases for training and 10000 cases for inference in MNIST. For ImageNet classification networks, we use pre-trained weights for convenience. We use 10000 pictures for inference from Large Scale Visual Recognition Challenge [194]. We evaluate the inference accuracy with top-five prediction results. We compare our design with two other baselines:

- **No heat resilience (Base)** implements a plain RRAM chip without any heat resilient design.
- **Temperature-Aware Row Adjustment (TARA)** implements prior work [189] proposed scheme.

Our schemes include:

- **HR^{3}AM with Bitwidth Downgrading (HR^{3}AM-BD) only** implements proposed bitwidth downgrading.
- **HR^{3}AM** implements the both proposed bitwidth downgrading and tile pairing mechanisms.

5.3.2 Inference accuracy

Figure 5.7 shows the inference accuracy of four neural network models under various ambient temperatures. We scale the ambient temperature from room temperature 300K to 360K, where RRAM chip is significantly affected by the poor heat dissipation. We observe several findings from the results. First, our design is resistant to the heat-induced temperature increase. HR^{3}AM-BP and HR^{3}AM manage to respectively obtain at least 83.5% and 87.8% relative accuracy among all the cases. Second, HR^{3}AM has relatively stable accuracy. TARA's accuracy drops by 40.05% on average when the ambient temperature raises 60K. At the same time, the inference accuracy of HR^{3}AM-BP only drops 7.77%, and that of HR^{3}AM only drops 4.82%. Third, networks with larger scale and more layers obtain more benefits from HR^{3}AM. VGG16, ResNet50, and InceptionV3 have a larger improvement from other baselines to our schemes than MNIST. The thermal issue is worse on these networks. These networks have complex structures and process large input images, which lead to intensive data movement between tiles and eDRAM

and high power consumption. HR^{3}AM provides further accuracy and thermal optimization for these cases. Overall, we manage to improve inference accuracy by 4.8%–58% over the Base and 4.3%–41.8% over TARA.

5.3.3 Thermal status

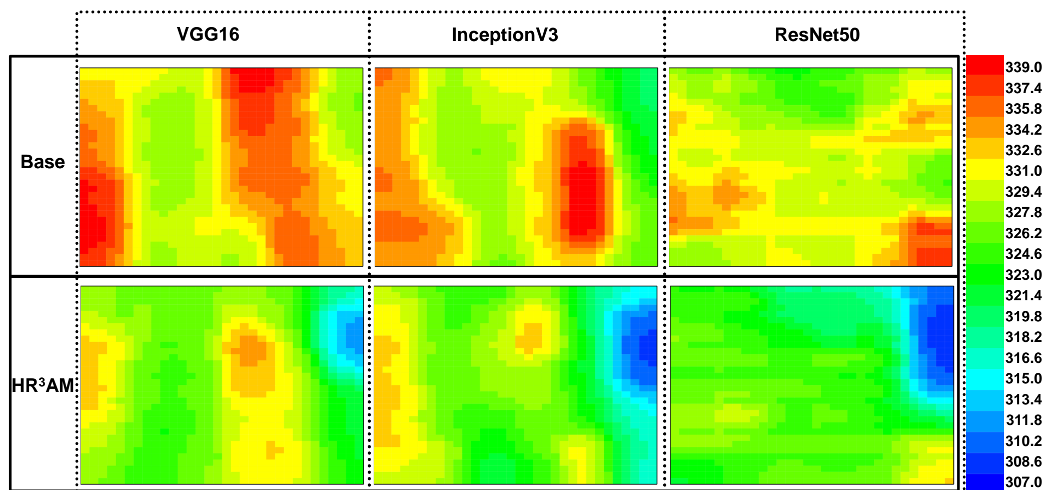


Figure 5.8: Comparison of thermal distributions of a RRAM chip when running three networks.

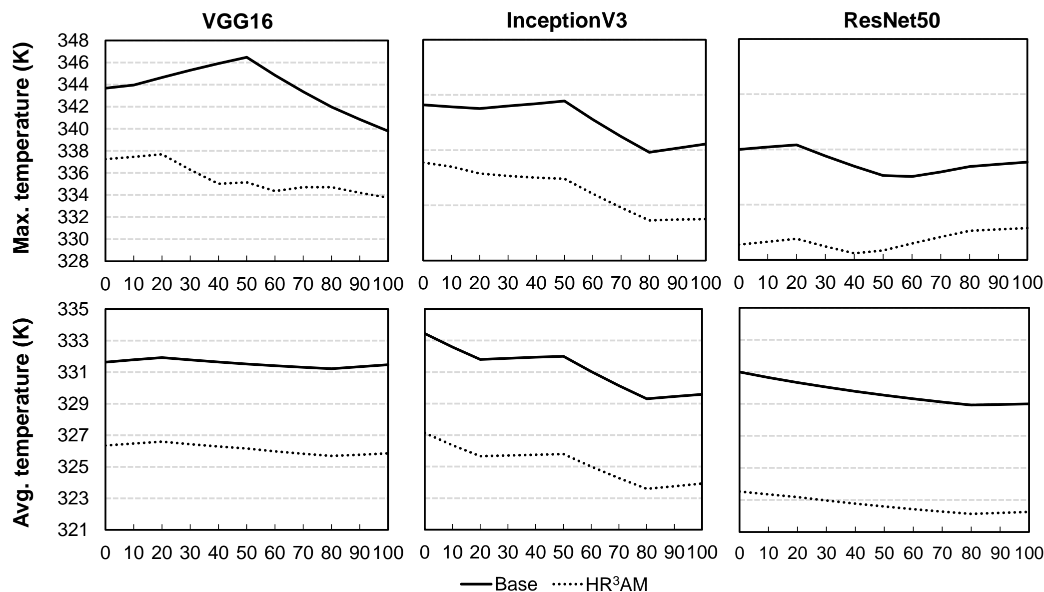


Figure 5.9: Comparison of maximum and average temperatures of RRAM chip when running three networks. The *x*-axis represents the temperature measurement time in milliseconds.

We present the results of thermal status with three network models: VGG16, ResNet50, and InceptionV3. Because TARA [189] does not optimize thermal status, we only compare our design with the Base which has no thermal optimization. We reserve 10% tiles as idle tiles for HR^{3}AM.

Figure 5.8 presents the comparison between HR^{3}AM and the Base on steady-state thermal distribution. We have three observations. First, our design significantly reduces the hot spot region. This indicates that our design manages to identify overheated tiles and mitigate heat for these tiles. Second, idle tiles help mitigate heat and stay at a relatively low temperature. However, these idle tiles form a cold spot. We can further optimize our design by wisely distributing idle tiles across the chip. Lastly, the released heat on hot tiles also benefits the inference accuracy. HR^{3}AM has a smaller percentage of crossbar arrays that triggers bitwidth downgrading than that of the Base, such that it brings a 2% improvement in inference accuracy compared to HR^{3}AM-BP.

To show the effectiveness of HR^{3}AM during chip operation, we measure the maximum and average temperature every 10ms across all the crossbar arrays of the chip. We measure a total of 100ms time during steady operational state. Figure 5.9 shows the maximum and average temperatures during this time. Our design limits the maximum temperature below 336K and average temperature below 327K. HR^{3}AM manages to contribute a 6.2K decrease on maximum temperature and a 6K decrease on average temperature.

5.3.4 Performance and energy

We adopt the performance and power model from [74] to estimate HR^{3}AM. We normalize the results of each network running on HR^{3}AM to the same network running on the Base. Figure 5.10(a) shows the 2.1% loss on normalized throughput. Both weight encoding in *bitwidth downgrading* and reserving idle tiles in tile pairing cause the performance downgrading. The weight encoding process typically delays the multiplication. However, it only happens when the downgrade bit changes. In tile pairing, HR^{3}AM reserves 10% of the tiles and therefore

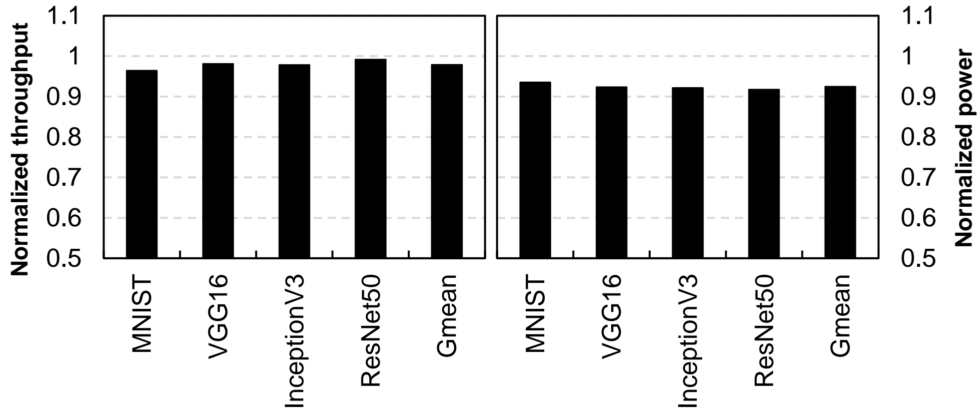


Figure 5.10: Normalized (a) throughput and (b) power consumption on HR^{3}AM.

loses computational parallelism. However, because these tiles work under the low power mode, Figure 5.10(b) shows a 7.7% average decrease on the normalized power.

5.4 Conclusion

In this work, we investigate how heat impacts inference accuracy of RRAM based accelerator. Our evaluation shows network inference accuracy drops significantly when the temperature increases. We propose HR^{3}AM, a heat resilient design for RRAM based neural network accelerators. HR^{3}AM consists two mechanisms: *bitwidth downgrading* and *tile pairing*. *Bitwidth downgrading* adjusts weights to the reduced RRAM conductance. Tile pairing matches hot spot RRAM tiles with idle tiles. Compared to state-of-the-art prior work, HR^{3}AM achieves an accuracy improvement by up to 41.8% on four real world applications of CNN models. Our thermal evaluation shows a 6.2K decrease on the maximum temperature and a 6K decrease on average temperature.

Acknowledgments

This chapter contains material from "HR^{3}AM: A Heat Resilient Design for RRAM-based Neuromorphic Computing.", by Xiao Liu, Minxuan Zhou, Tajana S. Rosing, and Jishen Zhao, which appears in the Proceedings of the 2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED 2019). The dissertation author is the primary investigator and first author of this paper.

Chapter 6

Mirage: A Highly Paralleled And Flexible RRAM Accelerator

The emergence of NVM provides an alternative for memory devices. One representative NVM – Resistive RAM (RRAM), or memristor, enables not only data storage ability, but also computation ability [201, 79]. This duality blurs the boundary between memory and computation and is also referred to as processing-in-memory (PIM). PIM architecture may achieve higher performance and better power efficiency, compared to the conventional storage-only memory architecture [202, 203, 204].

The RRAM-based PIM architecture has been intensively explored to accelerate deep neural network (DNN) applications in recent years [74, 75, 76]. Multiple RRAM cells form an array structure called "crossbar", which can conduct a multiply-accumulate operation in a single cycle. This Single Instruction, Multiple Data (SIMD) design allows parallel matrix multiplications and convolutions, which are the fundamental operations in various DNNs. For instance, calculations in fully connected layers and convolutional layers in convolutional neural networks (CNN) are essentially matrix multiplications and convolutions. Prior works demonstrate SIMD architecture can achieve better performance compared to CMOS-based application-specific

integrated circuits (ASIC) accelerators [74, 75, 76], and higher energy efficiency compared to GPU [73, 76].

However, existing RRAM-based DNN accelerators lack several essential features and compromise performance and flexibility. First, the data dependency resulting from shared data in the existing pipeline design degrades the performance of the accelerator. Based on our analysis of the pipeline that consists of both inter-layer and intra-layer parallelisms [76, 74], we discover that the existing designs spend up to 50% of inference time on idle cycles because of the shared data induced dependency. As we demonstrate in Section 6.1, data sharing between duplicate kernels intensifies the data dependencies between different layers, which leads to significant pipeline stalls during the execution. Our experimental results show that idle cycles caused by shared data can take up to 50% of the single inference latency for certain network models. Second, existing hardware assignments cannot fit with the pipeline design in the RRAM-based accelerator. The pipeline design requires a mapping between the hardware resource to different layers in a network. Each network needs to have a specific hardware mapping in order to execute on the accelerator. However, this requires an approach to customize the hardware assignment to the network layers automatically. On the one hand, the user-generated assignment could degrade the performance provided by hardware parallelism. On the other hand, the existing automatically generated assignments [78, 205, 206, 77] are not able to apply to the pipeline structure.

To address these issues, we introduce Mirage, an RRAM-based accelerator that provides 1) high parallelism by resolving the data dependency due to shared data in duplicate kernels, and 2) high flexibility by enabling automatic hardware assignment for general DNN applications. Mirage consists of two primary designs: Finer-grained Parallel RRAM Architecture (FPRA) and Auto Assignment (AA), to deal with issues raised by the shared data and hardware assignment, respectively. FPRA is based on the insight that the proper arrangement of the duplicate kernels and unified data buffering of input and output data can resolve the data dependency caused by shared data. FPRA introduces *kernel-batching* to group the computation of the duplicate kernels

of the same layer, and *data sharing aware memory* to uniformly manage the input and output data of duplicate kernels. AA utilizes the network parameters and accelerator hardware configurations to generate a hardware assignment automatically. AA follows two rules that we observed from the DNN models: 1) layers at the front need to have the same or higher level of parallelism of the layers deeper in the network[1], and 2) consecutive layers with the same level of parallelism suffer from the least amount of stall brought by data dependency. With the greedy strategy that follows these rules, AA searches for the best assignment for all the layers under the available hardware.

This dissertation makes the following contributions:

- We discover that shared data induces data dependency, and evaluate the corresponding inefficiency in the pipeline design of the RRAM-based NN accelerators.
- We show that the manual assignment of layers to the crossbars leads to underutilization of the hardware, while the existing automatic assignments are inefficient when the design is pipelined.
- We propose Mirage, a highly parallel and flexible RRAM-based NN accelerator. Mirage consists of two main features: Finer-grained Parallel RRAM Architecture (FPRA) and Auto Assignment (AA).
- We evaluate Mirage on eight image recognition DNN models. Our results show 2.0$\times$ speedup[2] and 2.1$\times$ throughput increase against the state-of-art baseline.

6.1 Background & Motivation

Existing parallelisms in RRAM-based accelerator. Similar to the CMOS-based NN accelerators [207, 208, 209, 210], RRAM-based accelerators possess *intra-layer* parallelism. In the intra-layer parallelism, the computations of all the parameters of a kernel are concurrent. The reason is that one crossbar array computes all the weights stored in RRAM cells concurrently

[1]For any layer in the network, "layers at the front" refers to the layers that are closer to the first layer, "layers deeper in the network" refers to the layers that are further from the first layer.

[2]Speedup represents the speed up of a single inference latency.

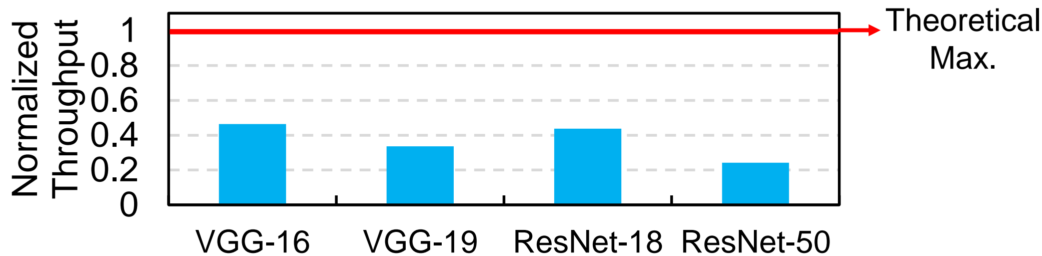


Figure 6.1: The throughput of four neural networks. Numbers are normalized to the theoretical maximum throughput of the accelerator.

over a single cycle. RRAM-based accelerators obtain intra-layer parallelism through mapping the kernel parameters to one or several crossbar arrays. RRAM-based accelerators can also duplicate kernel parameters to different groups of crossbar arrays to further improve parallelism, such that different parts of the input feature map are processed concurrently.

The pipeline design in the RRAM-based accelerators introduces *inter-layer parallelism*. RRAM-based accelerators can deploy more computational units within the same chip area because of the size advantage of the memristor. These additional computational units enable the RRAM-based accelerator to process more matrix multiplications in parallel. Therefore, the existing RRAM-based accelerators [74, 76] deploy all the network layers on an RRAM-based accelerator at the same time. Doing that requires the accelerator to allocate tiles for each layer, such that each layer becomes a pipeline stage of processing inference. A layer does not need to wait until the previous layer output feature map is entirely generated. This pipeline design allows each layer to start computation when the required input pixels are ready [74].

We estimate the theoretical maximum throughput of the RRAM-based accelerator (with the configuration in Table 6.1) to be 6.5×10^3 GOP/s (e.g., Giga 16-bit operations per second) for each tile. Figure 6.1 compares the theoretical throughput with the throughput of four RRAM-accelerated neural networks executing on the simulator. Results show a significant performance gap between theoretical throughput and simulated throughput. Based on our detailed analysis in the following section, we find that most of the performance gap is the data dependency related pipeline idleness.

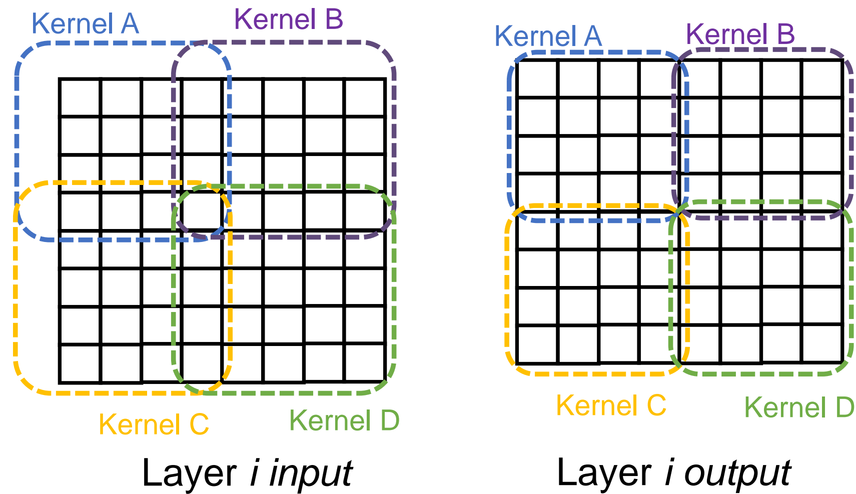


Figure 6.2: Shared data in a convolutional layer with four duplicate kernels.

6.1.1 Shared Data Induced Data Dependency

Shared data between duplicate kernels emerges when pipelining and kernel duplication are applied simultaneously. In kernel duplication, the feature map is evenly divided by the kernels. Different kernels compute different parts of the output feature map such that none of the computation is repeated. However, when the kernel size is larger than 1×1, parts of the input feature map overlap among the duplicate kernels. Figure 6.2 shows the data sharing of a convolutional layer with four duplicate 2×2-sized kernels. We may partition the input feature map differently. However, shared data still exists among duplicate kernels.

The shared data does not affect intra-layer parallelism by itself when all the input feature maps are prepared at the start of computation. The pipeline naturally has the data dependencies between consecutive layers. The shared data, however, further aggravates the data dependencies because the shared data blocks multiple kernels in the subsequent layers. Figure 6.3 illustrates an example of shared data induced data dependency in the first two convolutional layers in a DNN model. These two layers both use a 2×2-sized kernel and both kernels are duplicated twice. As the figure shows, all the output feature maps are divided into two halves. In each layer, two duplicate kernels share the middle part of the pixels of the input feature maps. During the

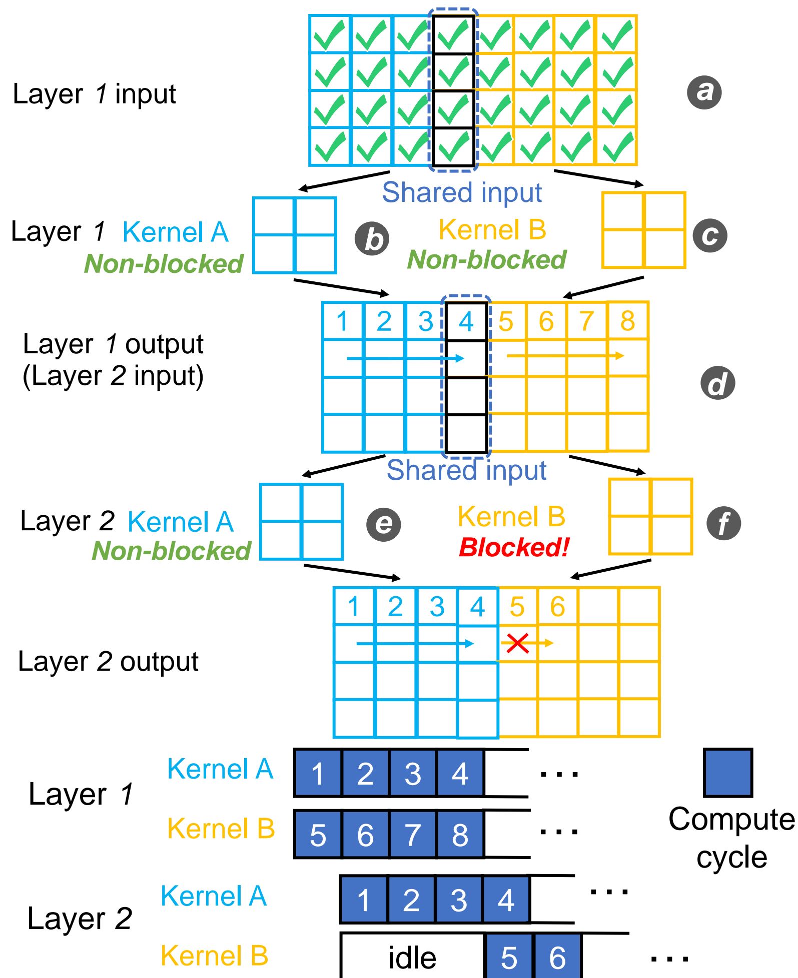


Figure 6.3: An example of data sharing in the first two layers of a DNN model.

computation, each kernel computes a single output pixel within a compute cycle. As starting at the top left corner of its input, the kernel moves from left to right to perform computations. The first layer has all its input feature maps (ⓐ) ready at the beginning. Therefore, none of the kernels (ⓑ, ⓒ) in layer 1 are blocked. In layer 2, kernel B (ⓕ) is blocked due to the missing input data (output pixel 4 in the layer 1) (ⓓ). Kernel B needs to wait until the output pixel 4 from layer 1 is computed. We may change the order of computation in layer 1 to force kernel A to compute the shared data first. This, however, blocks the computation of kernel A (ⓔ) in layer 2. With more duplicate kernels and deeper network layers, more shared data exists in the input feature map. This aggravates the data dependency problem, inducing more idleness of the duplicate kernel.

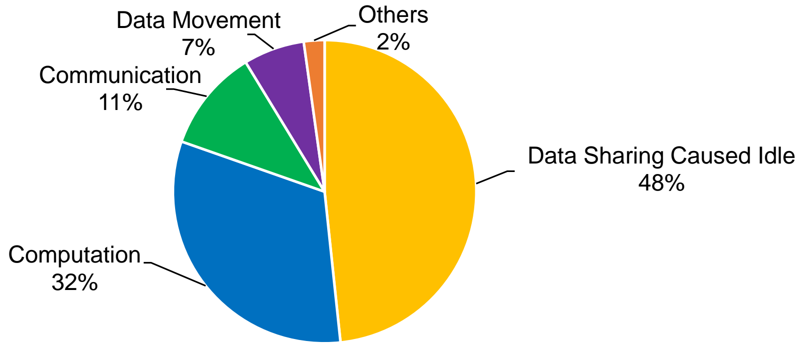


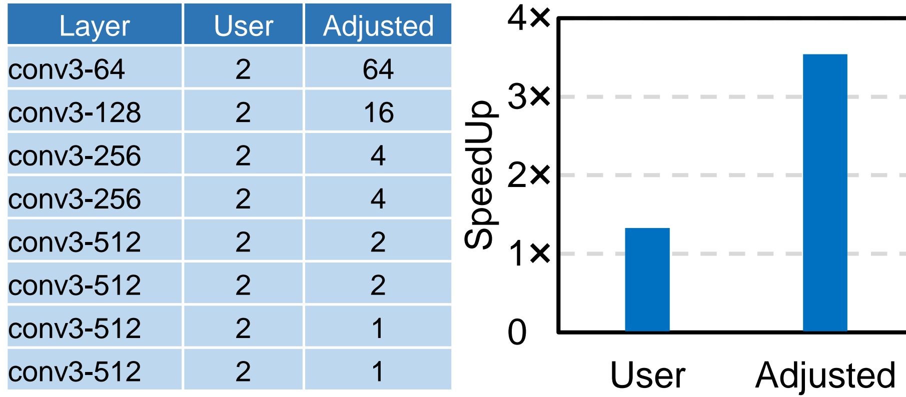
Figure 6.4: Single inference latency breakdown.

To quantify the inefficiency of the shared data induced data dependency, we simulate a pipeline-enabled RRAM-based accelerator (configuration details in Section 6.4) and break down the single inference latency, as shown in Figure 6.4. We break down the latency into five categories: computation time of crossbar arrays, idle time resulting from shared data related dependency, communication time (e.g., time for synchronization between tiles and MACs), data movement time (e.g., timing for transferring input and output data), and others.

Due to the pipeline, crossbar arrays belonging to different layers have significantly different cycles on each type of operation. We collect the cycles for each type of operation and calculate the breakdown for all the crossbar arrays of each layer. We calculate the overall breakdown with the geometric mean of each layer. We observe that the shared data induced idleness forms a significant portion, contributing to nearly 50% of the overall latency. This can be attributed to pending shared data blocking computations of multiple kernels. Therefore, arranging the sequence of computations on shared data becomes critical to the performance of pipeline-enabled RRAM accelerators.

6.1.2 Issues with Existing Hardware Assignments

The pipeline requires a dedicated assignment that maps each layer to different hardware [74, 76]. To properly execute a network, the accelerator relies on the users' experience



Layer	User	Adjusted
conv3-64	2	64
conv3-128	2	16
conv3-256	2	4
conv3-256	2	4
conv3-512	2	2
conv3-512	2	2
conv3-512	2	1
conv3-512	2	1

Figure 6.5: Comparison between user assignment and adjusted assignment on (a) the number of duplicate kernels in eight convolutional layers, and (b) inference speedup.

for the assignment. The user needs to estimate the required hardware resource of the network and assigns hardware accordingly. To maximize the accelerator's performance, the user also needs to be aware of the pipeline and kernel duplication to fully explore the parallelism of the accelerator. Prior works [78, 205] provide automatic hardware assignments of the network, but do not distinguish the difference between pipeline and non-pipeline designs. To fully utilize the performance of pipeline structure, the specific assignments for each layer still need to be coordinated by the users. Requiring these architectural experiences is impractical for the regular user, who expects to execute the network model on the accelerator, similar to GPUs or CPUs.

Poor user assignment can lead to underutilization of the hardware, as shown by the example in Figure 6.5. We configure the RRAM-based accelerator to execute on a VGG-11 [192] DNN model. We assume the hardware is sufficient for processing twice the amount of the network parameters. An inexperienced user may simply duplicate kernels of each layer twice to improve the parallelism. With the same hardware, an adjusted assignment reduces the kernel duplication in the last two layers to adopt more duplication in the layers at the beginning. This adjustment utilizes the hardware more efficiently. The front layers process larger feature maps, which benefits more from having more kernels. Even without shared data induced idleness, with same number of kernels, front layers take longer to process a feature map than the subsequent layers.

Figure 6.5 (a) compares the number of duplicate kernels of eight convolutional layers between the user assignment and the adjusted assignment. We simulate the performances of these two assignments and compare the speedup in Figure 6.5 (b). Since an normal user is oblivious of the intricacies of the data dependency induced in the pipeline and the required hardware resources for each layer, the user assignment results in a significantly slower design than our adjusted assignment.

Overall, pipeline designs in the RRAM-based accelerators suffer from shared data and hardware assignment induced inefficiency. Some existing RRAM-based accelerators attempt to provide flexibility and reconfigurability [78, 205] while ignoring the shared data and the pipeline architecture. Other existing accelerators attempt to address both issues by abandoning the pipeline design [206, 77]. However, these solutions underutilize the on-chip area of RRAM, which leads to lower throughput Mirage tries to resolve these issues while supporting the pipeline design of the accelerators.

6.2 Mirage

In this section, we introduce Mirage, an architectural design for the RRAM-based DNN accelerator. The design of Mirage aims to provide high parallelism and flexibility for various neural networks. As shown in Figure 6.6, the Mirage design has two major components: Finer-grained Parallel RRAM Architecture (FPRA), and Auto Assignment (AA). FPRA improves the parallelism of the RRAM architecture from two aspects: reducing shared data induced idleness in the crossbar arrays, and decreasing the amount of data transferred between tiles. AA improves the flexibility by automatically generating hardware assignment with high hardware utilization and parallelism.

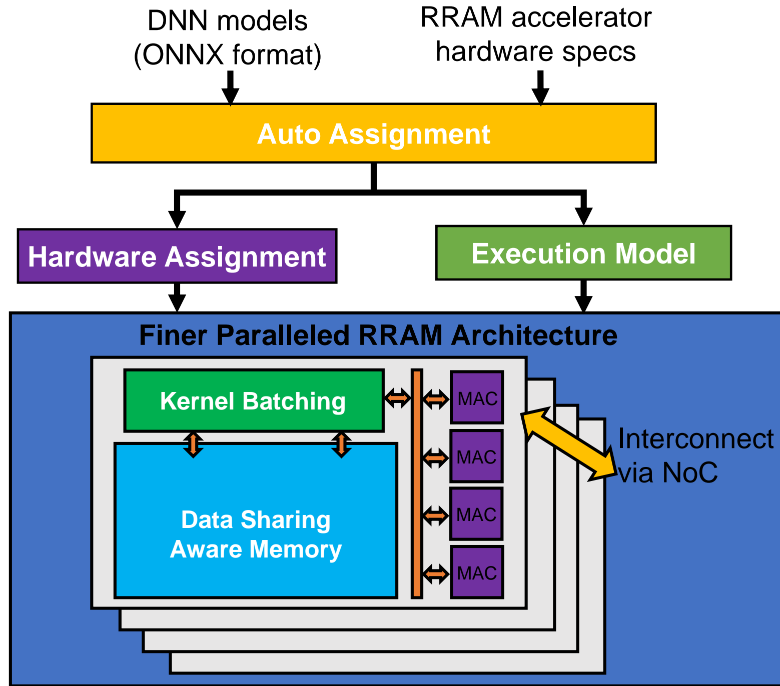


Figure 6.6: The overview of the Mirage architecture.

6.2.1 Finer-grained Parallel RRAM Architecture (FPRA)

To alleviate the shared data induced data dependencies (described in Section 6.1.1) in RRAM-based accelerators, we propose Finer-grained Parallel RRAM Architecture (FPRA). FPRA aims to achieve higher parallelism with finer-grained coordination of data between different kernels and different layers. As Figure 6.6 shows, FPRA is a tile-level design consisting of two mechanisms: *kernel batching* and *data sharing aware memory*. The kernel batching groups duplicate kernels and uniformly manages their data forwarding and computation. The data sharing aware memory stores shared input/output data between all batched kernels and manages data for all the kernels together to reduce unnecessary data duplication. We implement the kernel batching in the tile controller and data sharing aware memory in the eDRAM of each tile (details in Section 6.3.1).

Kernel batching. The shared data induced dependency exists because of the partitioned input feature map for each kernel (Section 6.1.1). We observe that enforcing all the kernels of the

same layer to simultaneously compute over continuous shared data, can eliminate the shared data induced dependency and only keep the data dependency between layers. We refer this mechanism as *kernel batching*. When all the layers in the network apply kernel batching, all the kernels belonging to the same layer behave uniformly - either pending for input or computing output.

Kernel batching requires several rules to guarantee the computations follow the network configuration without being repeated on different kernels. First, the kernel batching places all the kernels together with one stride size as the minimal gap. The adjacent two kernels have to be exactly one stride away. This makes sure the generated output data are continuous within each computation iteration, while none of them are overlapped. Second, the duplicate kernels shift in the same direction with the same gap between each computation. This makes sure the generated output data between computation iterations are continuous. Third, the duplicate kernels shifts may shift multiple strides between computations. Depending on how the kernels are batched and the direction kernels are moving, each kernel may take multiple strides to avoid recomputating the data.

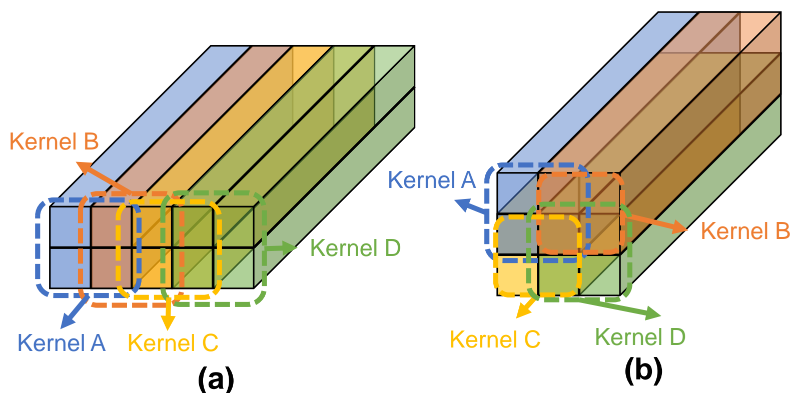


Figure 6.7: Two kernel batching types: (a) linear batching, and (b) square batching.

We propose two types of kernel batching: linear batching and square batching. The linear batching places all the kernels in a row (or a column). Figure 6.7 (a) shows an example of the linear batching in a convolutional layer. The kernel size is $2\times2\times64$, and the stride size is 1. The same kernel is duplicated into four replicas in a row with indices A, B, C, and D. The batched

kernels move in a vertical direction with a single stride gap. The linear batched kernels rely on rectangular-shaped input to generate a one-line output. The square batching places the same amount of kernels in consecutive rows and columns. Any two adjacent kernels still share only one stride gap. Figure 6.7 (b) shows an example of the square batching. In this example, we use the same kernel size and stride as the linear batching. The square batched kernels rely on the square-shaped input to generate a square-shaped output.

Two types of kernel batching fit with different neural network layers and different amounts of duplicate kernels. The squared batching fit with the convolutional layer connects to the pooling layer. Because pooling layer relies on square-shaped input, if the squared batched kernels produce exactly the same sized output data for every computation iteration, the idleness between the convolutional layer and the pooling layer is minimized. However, the square batching requires the kernel duplication number to be square, which may not be achievable due to the limitation of available hardware. Hence, the linear batching is more feasible with any number of duplicate kernels. Mirage uses linear batching and square batching simultaneously. The FPRA provides both types of batching while AA decides which type is used based on available hardware and topology of the neural network. We explore the performance impact of different strategies of batching type in Section 6.5.2.

Data sharing aware memory. The kernel batching helps reduce the shared data induced idleness in the pipeline. However, as the batched kernels share more data, the amount of input data used by all the kernels is significantly increased. For example, in a convolutional layer with 64 64×64-sized input feature maps and four kernel replicas, using linear batching can increase the amount of input data by $1.6\times$ compared to no kernel batching. Because the duplicate kernel is not aware of the shared data, the amount of data forwarding between layers can significantly increase. The increased data movement exhausts the bandwidth of NoC, and leads to limited performance increase (Section 6.5.1).

To solve the issue, we propose *data sharing aware memory* to efficiently forward data

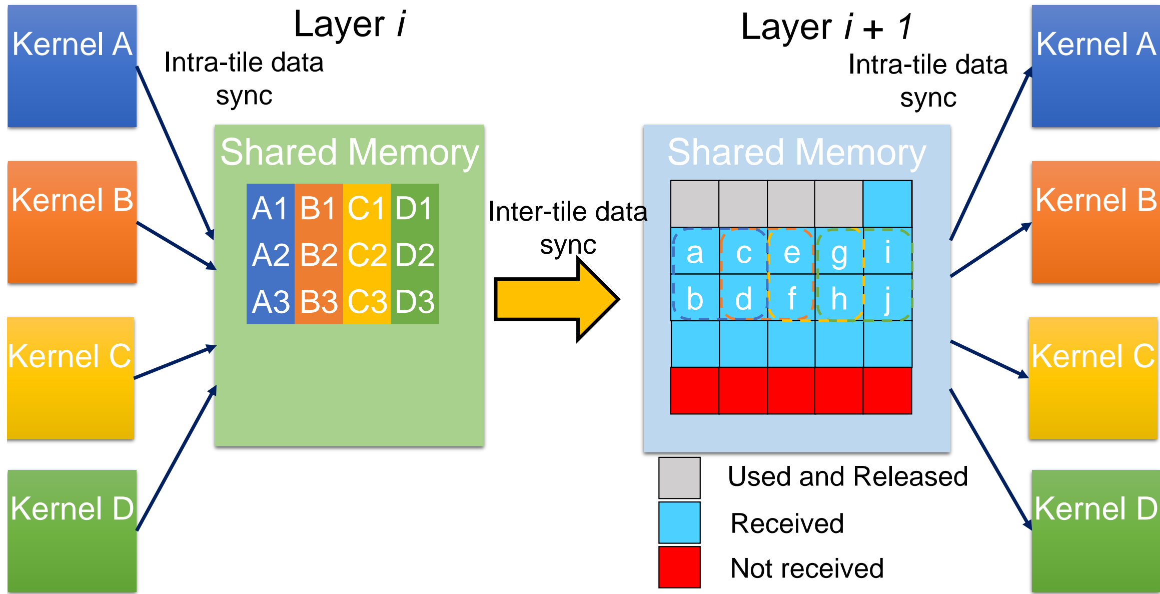


Figure 6.8: Data sharing aware memory in two layers of a DNN.

between layers, and dispatch data to duplicate kernels. The data sharing aware memory utilizes the memory space provided by the eDRAM of each tile (Section 6.3.1). The data sharing aware memory buffers a part of the input and output feature maps of the layer. As the Figure 6.8 shows, the feature maps data is transferred via inter-tile data synchronization through NoC. The output data of four linearly batched kernels in layer i stores in the data sharing aware memory, and transfer to the memory of layer $i+1$ together. The data sharing aware memory stores the input feature map for all the kernels as an entirety. The memory dispatches the input data to each kernel's crossbar arrays. The dispatch only happens inside the tile (e.g., intra-tile data synchronization), which costs no NoC bandwidth. This is because each kernel is loaded across multiple crossbar arrays, where crossbar arrays belong to different tiles. In this example, the data sharing aware memory dispatches data $a-d$ to the kernel A's crossbar arrays, $c-f$ to the kernel B's crossbar arrays, $e-h$ to the kernel C's crossbar arrays, and $g-j$ to the kernel D's crossbar arrays.

Data sharing aware memory also manages the execution of batched kernels. Since data sharing is invisible to duplicate kernels, the memory utilizes intra-tile data synchronization to

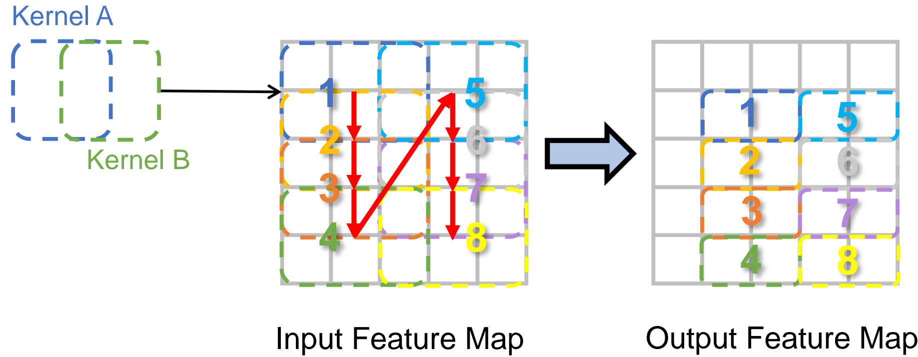


Figure 6.9: Batched window shifts in the input/output feature maps of a convolutional layer. The layer uses two batched $2 \times 2 \times 64$-sized kernels with one stride size. Two batched kernels use a shared input of a $2 \times 3 \times 64$-sized window. The numbers indicate the steps and the red arrow represents the direction.

start computation of each kernel. The data sharing aware memory keeps track of the kernel's location and progress, as the duplicate kernels metadata are stored inside the memory. This provides corresponding input data for each layer upon receiving these data. As batched kernels move with the same stride and direction, the data sharing aware memory uses the *batched window* to uniformly keep track of the input and output data. Figure 6.9 shows an example of the batched window shifting over the input and output shared memories. The movement of the batched window guarantees that none of the output is missed or overlapped. In each step of the batched window, the data sharing aware memory dispatches the data within the window to all the kernels. When the computation is complete, the output data is sent to the shared memory. When all the data within the output batched window is received, the tile forwards the data to the next layer. The data will be released when the computation and forwarding are finished.

Data sharing aware memory benefits from kernel batching design as well. The batched kernels always generate continuous output data, and consume input data continuously for compute. The data sharing aware memory only needs to buffer the received data that are needed for current and future computations. Once the computation completes, the input data no longer required can be released. We evaluate that the size of eDRAM (e.g., 64KB in Table 6.1) is efficient for the data sharing aware memory.

6.2.2 Auto Assignment

Auto Assignment (AA) is a method to automate the allocation of hardware resource to neural networks for the RRAM-based accelerator pipeline. AA engages at the configuration stage before the network is loaded to the accelerator. As shown in Figure 6.10, AA takes in the accelerator configurations and the network hyper parameters, to generate the hardware assignment as well as an execution model that specifies the order of execution and data forwarding path.

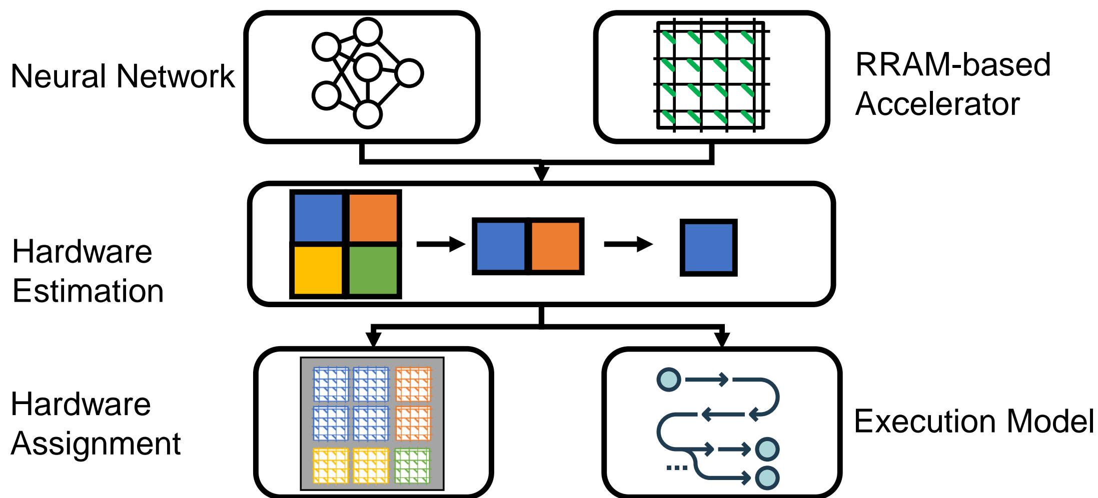


Figure 6.10: The work flow of the auto assignment.

AA consists of three steps. First, AA extracts key information from the accelerator configurations and network hyperparameters, including the crossbar array size, the number of crossbar arrays, the topology of the network, computation parameters of each layer (e.g., kernel size and strides), etc.

Second, AA estimates the hardware resources (e.g., number of crossbar arrays) required for each layer based on the network information. Using a greedy strategy-based Algorithm 1, AA decides the intra-layer parallelism (e.g., number of duplicate kernels) for each layer.

Lastly, AA generates the hardware assignment and execution model based on the estimated hardware resources. AA assigns each layer of the network to the specific hardware (e.g., indices of tiles, MAC, and crossbar array) and produces an execution model for each layer's hardware,

which includes number of kernels, kernel sizes and locations, type of kernel batching, and adjacent layers' hardware indices. The execution model specifies the order of data to be processed by assigned hardware and data forwarding between the hardware. The hardware assignment and execution model are loaded to the accelerator during preconfiguration.

Searching intra-layer parallelism. To find the best inter- and intra-layer parallelisms for the network, the AA first generates a baseline which only has a pipeline for the inter-layer parallelism with no kernel duplication. The baseline is the minimal requirement for hardware resource to execute the network. The extra hardware resource provided beyond the baseline can duplicate kernels in certain layers. We use two rules to assign the number of kernels for each layer:

- *Rule 1:* Layers at the front should have **more or at least the same** numbers of kernels of the layers deeper in the network.
- *Rule 2:* The difference in the number of kernels between two consecutive layers should be **minimum**.

`Rule 1` is based on the observation that layers deeper in the network always rely on the output from front layers. While in most DNNs, the feature map size on the x and y dimensions decreases as the network propagates. When the kernel moves in x and y dimensions, it takes less or the same computation iterations for the layers deeper in the network. Hence, having more kernels in the layers deeper in the network does not increase the parallelism of the layer. `Rule 2` is based on the observation that two consecutive layers with the same amount of kernels have the least amount of idleness with an exception for the layers connected with the pooling layer. Because the pooling operation reduces the amount of data, the layer before pooling needs to have more kernels to produce data.

Algorithm 1 and its flow chart (Figure 6.11) show how AA searches the best intra-layer parallelism for all network layers, where ϕ represents the number of duplicate kernels in each layer and R represents the hardware resource. The algorithm first generates a baseline (❶) and

Algorithm 1 Searching intra-layer parallelism.

Input: NN, available HW resource (R_{avl})
Output: $\phi[N]$

1: $\phi[:]=1$ {Setup the baseline.}
2: $\phi' = \phi$
3: **repeat**
4: $\quad \phi = \phi'$
5: $\quad L$ = Generate Candidate lyrs. (ϕ', NN) {L are the candidate lyrs indexes.}
6: $\quad$ **while** ($R_{req} > R_{avl}$) and (L is not empty) **do**
7: $\quad\quad \phi' = \phi$
8: $\quad\quad l$ = Pick one lyr(L)
9: $\quad\quad \phi'[l] = \phi'[l] \times 2$
10: $\quad\quad R_{req}$ = Estimate(ϕ') {Estimate baseline required HW.}
11: $\quad$ **end while**
12: **until** L is empty
13: **return** $\phi[N]$

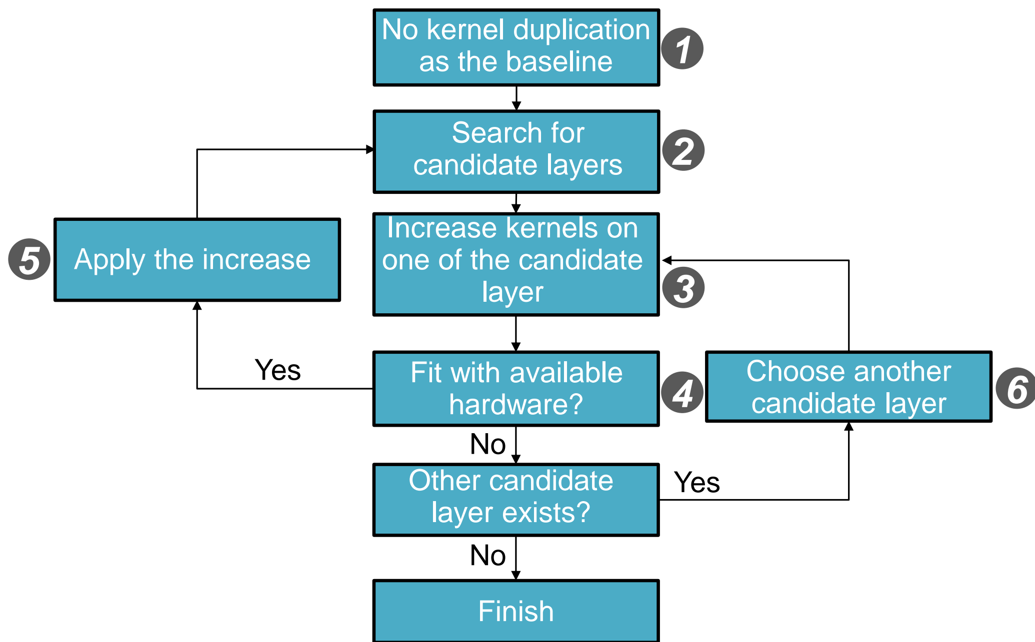


Figure 6.11: The flow chart of searching algorithm.

then adjusts its intra-layer parallelism. The algorithm selects candidate layers for increasing its number of duplicate kernels (❷). The requirement for the candidate layers is that increasing numbers of duplicate kernels on these layers still follows the above two rules. The candidate layers include the first layer and the layers with fewer kernels than the previous layer. The

algorithm evaluates two layers connected by the pooling layer in a different way. The subsequent layer becomes the candidate when the front layer has more than $k\times$ of kernels than the subsequent layer, where k stands for size of pooling filter. The algorithm then selects one layer from the candidate to increase its number of kernels (❸). The algorithm uses a greedy strategy that always picks the layers deeper in the network first. Because layers deeper in the network with a lower level of intra-parallelism are the bottleneck of the pipeline, increasing kernels on these layers provides higher parallelism than the layers in front. The algorithm evaluates the target number of hardware resources for such increase (❹). If the required duplicate kernels fit with the hardware, the algorithm applies the update and starts the next iteration (❺). If not, the algorithm chooses other candidate layers and re-evaluates the required hardware (❻). The algorithm terminates when none of the candidate layers can be used to increase parallelism.

6.3 Mirage Implementation

6.3.1 The Implementation of FPRA

FPRA Tile Architecture. To support kernel batching and memory sharing, we design the tile-level architecture for FPRA. Figure 6.12 shows the tile architecture of FPRA. This architecture design consists of I/O buffer, shared memory, tile controller, and MACs.

The I/O buffer handles the data transfer between tiles. The buffer connects to the router to handle communications with other tiles. The buffer uses two circular FIFO queues to send data and receive data separately.

The shared memory stores input/output feature maps and kernel-related metadata. The buffered input/output feature maps are stored separately and shared among all the MAC units. Kernel-related metadata contains hardware assignment data and the execution model. Unlike feature maps data, metadata are written into the memory during the accelerator's pre-configuration, and cannot be modified during the execution. Metadata is used by the tile controller to perform

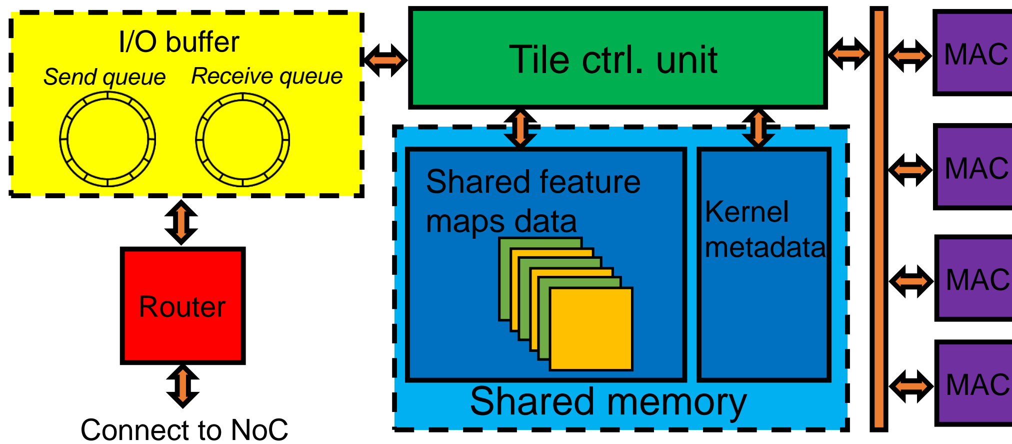


Figure 6.12: Tile architecture.

kernel batching and manage data sharing aware memory.

The tile controller manages the data forwarding between the rest of the components. The preconfigured metadata provides the controller information on how MACs are assigned to different kernels and how data is shared between MACs. As mentioned in Section 6.2.1, the controller stores only part of the feature map to save memory space. The tile controller uses the *intra-tile data synchronization* to not only transfer the data between shared memory and MACs, but also manage the execution of MACs in the correct order.

Intra-tile data synchronization. As shown on Figure 6.13, the intra-tile data synchronization consists of the following steps. At the beginning, MACs remain idle when the input data within the batched window is still pending (❶). When all the data in the batched window is received, the tile controller notifies all the MACs, and forwards the corresponding data to the MAC belonging to each kernel (❷). The MACs start computation upon receiving their input data (❸). When computation finishes, the output data is forwarded to the shared memory (❹). Once the output data within the output batched window is received, the tile controller sends data to the other tiles (e.g., next layer's tiles) through inter-tile data synchronization. To start the next iteration of computation, the controller also shifts the batched windows for both output and input (❺). The tile is reset to pending, and the input memory re-evaluates the accessibility of data

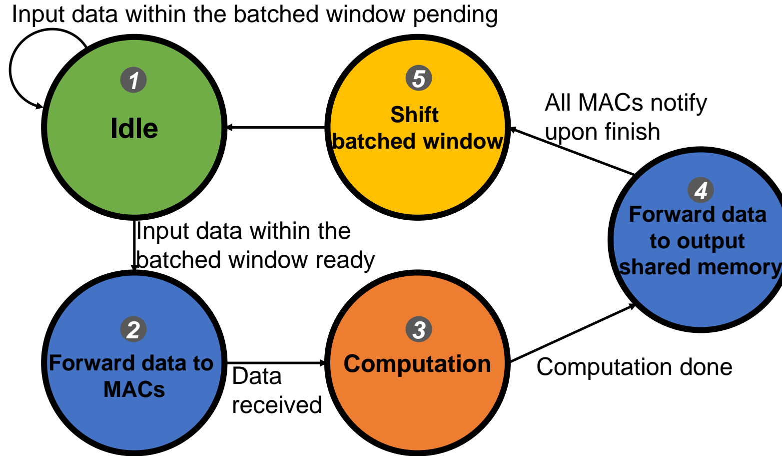


Figure 6.13: State diagram of the intra-tile data synchronization.

within the batched window. All MACs resume to the idle state and wait for the input data (❶).

6.3.2 Auto Assignment Tool Implementation

We implement AA as the auto assignment software tool deployed during the preconfiguration step. The tool implements the three major functionalities described in Section 6.2.2.

The first functionality is the analysis of the neural network based on the user's description on the structure of the neural network in a standard NN format ONNX [211]. ONNX enables neural network models implemented with PyTorch [212] or Tensorflow [198] to be analyzed by the AA tool without any transformation. For the accelerator specification, the current implementation directly uses the simulator hardware configuration. The AA tool estimates the required hardware based on the amount of computation in the neural networks. The estimation calculates the required number of MACs with the size for each layer's kernel, the size of each crossbar array, and the number of crossbar arrays in each MAC.

We implement the parallelism searching algorithm as described in Section 6.2.2. The time complexity of the algorithm is $O(N^2 log(K))$, where N is the number of network layers, and K is the amount of hardware resource (i.e., number of tiles). With the common network sizes and hardware specifications, our AA tool executes within a few seconds.

Lastly, the AA tool generates the hardware assignment and the execution model based on the estimated hardware resources. The tool assigns the hardware (e.g., tile and MACs) to each layer. The generated assignment is a list of entries consisting of the index of the hardware component, its corresponding network layer index, the number of duplicate kernels, and the index of each kernel. The execution model for each tile consists of the indices of layers for inter-layer data synchronization, each MACs' data for intra-tile data synchronization, and data sharing of duplicate kernels. Our tool generates the assignment and execution model in the YAML format files. The simulator imports these files to configure the accelerator.

6.4 Experimental Setup

Mirage simulation. We build our in-house cycle-accurate simulator for the RRAM-based accelerator in Python. Our simulator implements the entire architecture of the RRAM-based accelerator, and simulates computational and data movement operations in the level of crossbar arrays. The simulator models the pipeline between layers, such that hardware belonging to each layer has its own status (e.g., cycles and power) and uses message passing for data movements and communication. We adopt and modify the high bandwidth memory (HBM) model in Ramulator [213] to simulate the eDRAM. For the operations in the crossbar, we include the cycle parameters of matrix multiplication, ADC and S+A based on [74, 214, 215, 216]. The latency of basic operations are listed in Table 6.1.

To utilize existing DNN models from popular frameworks such as TensorFlow [198] and PyTorch [212], the simulator uses the inference trace generated by the Glow compiler [217]. The Glow compiler is able to process the ONNX [211] format file specified by the DNN models. We modify the compiler to generates a trace of the network at the inference stage. The trace includes the number of computation operations, input and output data sizes, the sequence of executions.

Baselines. We include two other baselines for our evaluation: pipeline enabled RRAM-

based accelerator without kernel duplication (referred to as **w/o kernel duplication** in Section 6.5), and pipeline enabled RRAM-based accelerator with kernel duplication. The implementation of these two baselines are based on the architecture described in [74, 76]. For the kernel duplication-enabled baseline, because the primitive design does not specify the hardware assignment for each network, the baseline uses AA generated hardware assignment for each DNN and runs the corresponding DNN benchmark with the hardware (referred to as **w/ kernel duplication opt**).

Benchmarks. We select eight representative DNN models for ImageNet classification [200]. These networks include GoogleNet [125], MobileNet [218], ShuffleNet [219], InceptionV3 [220], ResNet [193], and VGG [192]. We use the same topologies and hyperparameters provided in the original publication. As Mirage and baselines only accelerate the inference operation, we use the pre-trained weights in the torchvision package [221] for the experiments. Because Mirage avoids any modification on the network models and the input data, results show no compromise over the inference accuracies (both top-one and top-five) of the network.

Hardware configuration and power model. We configure the RRAM-based accelerator using parameters in Table 6.1. The NoC uses mesh topology and minimal adaptive routing. For the scalability evaluation, we scale the chip area by increasing the number of tiles and all related NoC components. To minimize the communication cost, we lay out tiles to form a rectangle with the minimal length to width ratio.

We use the statistics from [184] to model the power consumption of crossbar array. The power and area of the tile controller are validated with System Verilog through Synopsys Design Compiler and Synopsys 32nm PDK [222]. The ADC power and area are based on the the SAR ADC model specified in [223, 224]. The DAC power and area are based on [225]. We use CACTI tool [226] to model the rest of the on-chip components including eDRAM and buses under 32nm. We compare our models to the configurations from [73] and [74] to ensure the consistency. The simulator computes the dynamic power based on the executed operations and the energy

Table 6.1: Hardware configuration of the Mirage.

Component	Parameter	Spec.	Power
Controller	Frequency	1GHz	0.3mW
Tile	Number	168	312mW
Shared eDRAM	Size Bus-width T_{read}/T_{write}	64KB 256b 12ns	17.66mW
I/O buffer	Size	256 entries 2 queues	5.3mW
MAC	No. per tile	12	24mW
Crossbar array	No. per MAC Size Bit-width $T_{mul}/T_{ADC}/T_{S+A}$	8 128×128 2 1 cycle	2.4mW
Off-chip memory DDR4	Size Read Bandwidth Write Bandwidth	16 GB 61 GB/s 47 GB/s	

consumption for each operation. The static power is considered for data storage components including eDRAM and buffers.

6.5 Evaluations

This section shows the evaluation results of Mirage in terms of performance (Section 6.5.1), flexibility (Section 6.5.2), the power efficiency (Section 6.5.3).

6.5.1 Performance

Inference latency speedup. Figure 6.14 shows the speedup over the baseline w/o kernel duplication of four configurations. Besides the Mirage configuration, we add the Mirage configuration without data sharing aware memory (referred to as **Mirage w/o data sharing aware memory**). We make three observations from the result. First, Mirage has the highest speedup among four configurations. Mirage has a 3.5× speedup against the w/o kernel duplication and a

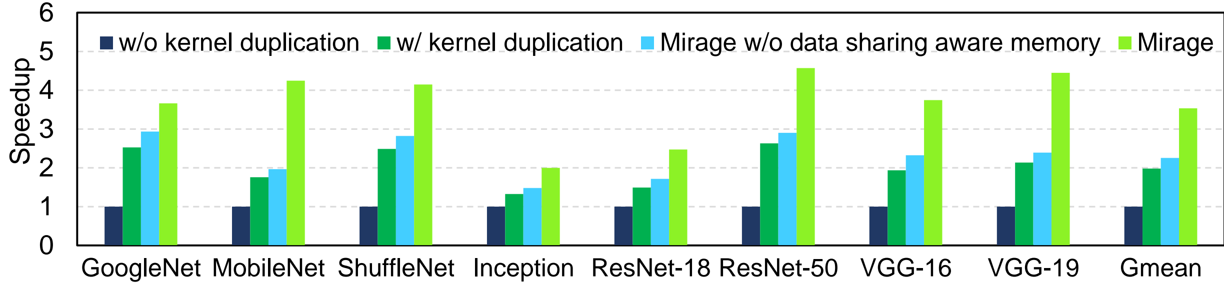


Figure 6.14: Speedup of single inference latency with four RRAM-based accelerator configurations.

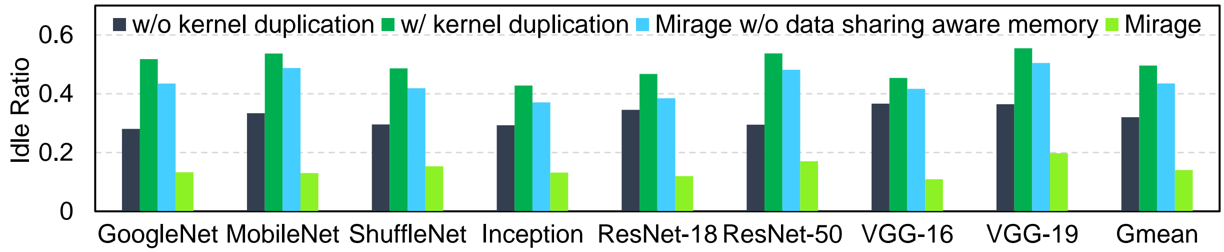


Figure 6.15: Idle ratio of four configurations.

2.0× speedup against w/ kernel duplication on average. Second, Mirage obtains a higher speedup on the networks with more layers. For instance, compared to the w/ kernel duplication, Mirage has a 2.1× speedup on the VGG-19, and a 1.9× speedup on the VGG-16. We observe the same trend on the ResNet that the speedup is higher on ResNet-50 than on ResNet-18. This indicates Mirage brings more benefit with deeper networks, as they tend to have a more severe data dependency. Third, data sharing aware memory is essential to the Mirage. Without data sharing aware memory, Mirage only has 1.1× speedup to the w/ kernel duplication. As Figure 6.15 shows, the idleness of batched kernels' hardware does not show a significant reduction.

Execution idle ratio. Figure 6.15 shows the shared data induced idle ratio of two Mirage configurations and two baselines. Because the idle cycle of crossbar arrays differ significantly between layers, the calculation using the summation of idle cycles of all the crossbar arrays diminishes the idle time on the layers with minimal parallelism. Hence, we calculate the accelerator idle ratio using the *geometric mean of each layer's idle ratio.* The idle ratio of each layer is the ratio of the idle cycles to its own execution cycles. We make two more observations from

the result. First, Mirage achieves the lowest idle ratio among configurations. Mirage is able decrease the shared data induced idleness by 3.5× compared to w/ kernel duplication. Second, Mirage w/o data sharing aware memory suffers from high idle ratio. We discover that the kernel batching increases the traffic to throttle the NoC bandwidth. This leads to both high idle ratio on Figure 6.15 and low speedup on Figure 6.14.

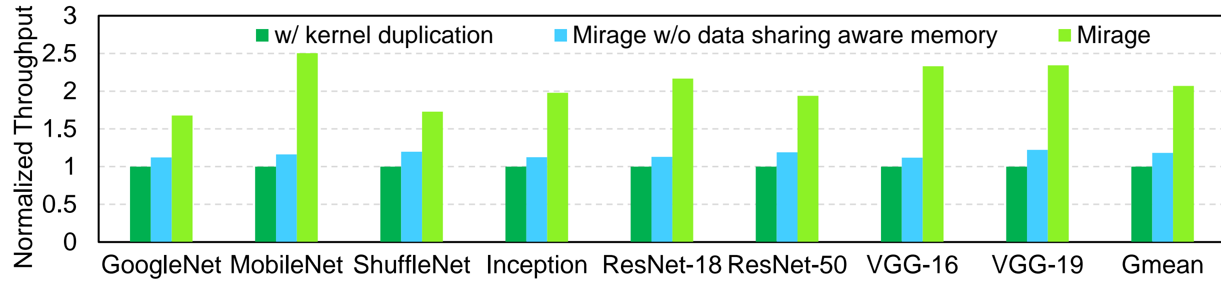


Figure 6.16: Comparison of throughputs of three configurations.

Throughput. Figure 6.16 shows the throughput of three configurations measured in giga 16-bit operations per second (GOP/s). We normalize the result to the w/ kernel duplication baseline. The result is generated with a 16-chip setup. We exclude the w/o kernel duplication baseline, due to its underutilization of the hardware resource and significantly lower throughput. We make two observations from the result. First, Mirage manages to achieve 2.1× increase compared to w/ kernel duplication baseline. Second, data sharing aware memory is essential to system throughput. Due to the throttled bandwidth, the Mirage w/o data sharing aware memory is unable to achieve notable improvement in throughput.

6.5.2 Flexibility

AA-generated hardware assignments. To show the flexibility of Mirage, we list the number of duplicate kernels along with the hardware utilization ratio in Table 6.2 for four different VGG configurations. We configure the accelerator to have 16 chips for processing the network. The hardware utilization ratio is the number of assigned crossbar arrays to the number of overall

Table 6.2: Number of duplicate kernels for each layer, and hardware utilization in four VGG configurations.

Model	VGG-11	VGG-13	VGG-16	VGG-19
conv3-64	4096	2048	2048	2048
conv3-64		2048	2048	2048
conv3-128	512	512	512	512
conv3-128		256	256	256
conv3-256	128	64	64	64
conv3-256	64	32	32	32
conv3-256			32	16
conv3-256				16
conv3-512	16	8	4	2
conv3-512	8	8	2	2
conv3-512			2	1
conv3-512				1
conv3-512	2	2	1	1
conv3-512	1	2	1	1
conv3-512			1	1
conv3-512				1
Utilization	99.98%	99.18%	99.83%	99.81%

crossbar arrays. Based on the result, we make two observations. First, AA manages to adjust hardware to different configurations of the same DNN model. With the same accelerator and same type of neural network, the generated assignment can be different even with only changes on a few layers. Adopting these changes can be a troublesome and error-prone task for the users. The AA-generated assignment follows the two rules and achieves the best performance. Second, the assignment achieves high hardware utilization for all the configurations. When a particular layers' desired hardware is unavailable, AA searches other candidate layers for the best second choice. This minimizes the unused hardware on an accelerator when the network size varies. The hardware utilization of Mirage is at 98.75% on average across all the benchmarked models.

Scaling the chip size. Figure 6.17 shows a sensitivity study of the chip size (from 1× to the 32×) for Mirage and the w/ kernel duplication. Both speedup and throughput are normalized to the w/ kernel duplication at 1× chip size. We make two observations. First, Mirage shows better scalability for both speedup and throughput compared to the baseline with kernel duplication and provides higher saturation point at a larger chip size. Second, Mirage has better scalability in

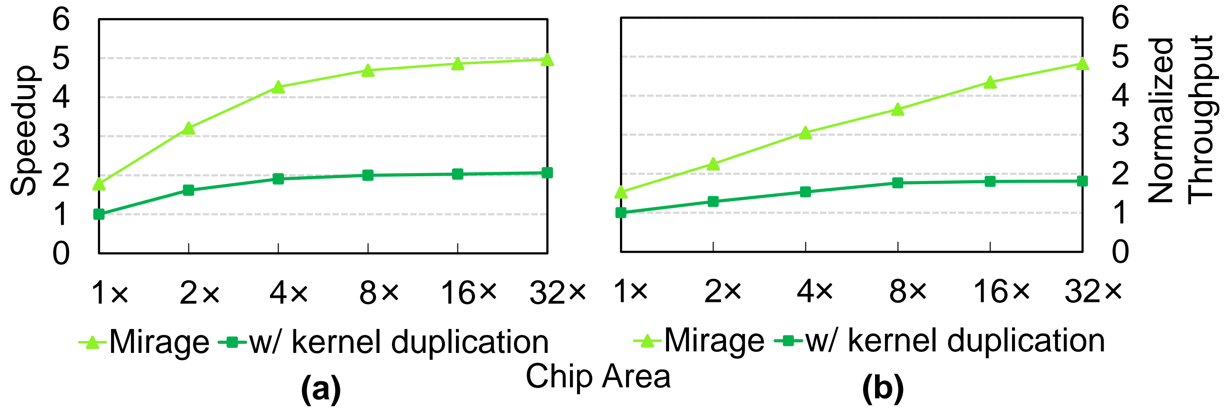


Figure 6.17: Comparison of Mirage and w/ kernel duplication over (a) speedup, and (b) throughput when the chip size scales from 1× to 32×.

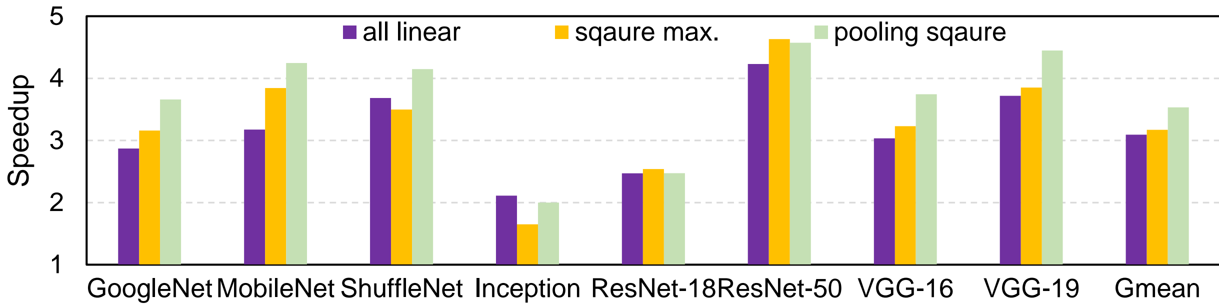


Figure 6.18: Speedup of three batching strategies with Mirage.

throughput than in speedup. The inter-layer parallelism benefits from FPRA to minimize the gap between difference inferences.

Two types of kernel batching. Figure 6.18 shows the comparison of three strategies for using two types of kernel batching. The all linear strategy utilizes linear batching on all the layers. The square max strategy attempts to use square batching wherever possible. This strategy batches the kernel to the maximum possible square, and places squares next to each other. For example, for 2048 kernels, it batches them to two 32×32 square batches, and then places them side by side. The pooling square strategy utilizes square batching on the layer before the pooling layer whenever possible. This strategy first batches the kernels to the size of pooling filter, and then linearly places batches side by side. For instance, for 32 kernels on a layer before 2×2 max-pooling, it batches them into eight 2×2-sized batches and linearly places them in a row.

We observe that pooling square strategy achieves the best performance on average among three. Despite the other two strategies showing a minor advantage in three models (e.g., Inception, ResNet-18 and ResNet-50), the pooling square strategy has the highest speedup among most workloads. Hence, Mirage uses the pooling square strategy by default. We only explored these three strategies due to space limitation. Other strategies may further increase the performance of Mirage.

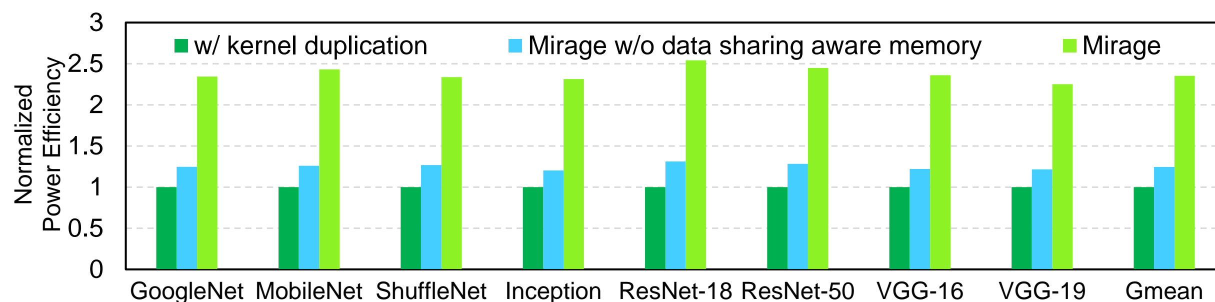


Figure 6.19: Power efficiency of three configurations.

6.5.3 Power Efficiency

Figure 6.19 shows the power efficiency result of Mirage and baseline. The power efficiency is measured by giga 16-bit operations per second per watt (GOPS/W). We normalized the results of two Mirage configurations to the w/ kernel duplication opt baseline. We make two observations from the result. First, the Mirage without data sharing aware memory configuration only has a 24% increase compared to the baseline on the power efficiency. The kernel batching brings more data movements between tiles. The heavy memory traffic delays the communication between tiles. Second, the Mirage design manages to achieve 2.4× power efficiency of the baseline. With kernel batching and data sharing aware memory, FPRA is able to reduce idleness during the inference, which leads to more operations with a single unit of power.

6.6 Conclusion

We propose Mirage, a flexible and highly parallel RRAM-based accelerator design. Mirage consists of Finer-grained Parallel RRAM Architecture (FPRA) and Auto Assignment (AA). FPRA applies the *kernel batching* and *data sharing aware memory* to reduce the data sharing induced idle in the pipeline. FPRA's kernel batching positions the kernel closer to each other to reduce the data sharing induced data dependencies between kernels, while FPRA's data sharing aware memory efficiently manages data forwarding between layers and duplication kernels. AA enables the execution of various sized networks on the RRAM-based accelerator pipeline design. AA takes the topology and parameters of the DNN model and specifications of the accelerator as input. It automatically generates a hardware assignment and execution model for the accelerator. The assignment achieves high utilization of the hardware and provides high performance. The evaluation shows Mirage achieves 2.0× speedup improvement and 2.1× throughput increase compared to the state-of-the-art baseline. Mirage also achieves high power efficiency and good scalability with chip size.

Acknowledgments

This chapter contains material from "Mirage: A Highly Paralleled And Flexible RRAM Accelerator.", by Xiao Liu, Minxuan Zhou, Rachata Ausavarungnirun, Sean Eilert, Ameen Akel, Tajana S. Rosing, Vijaykrishnan Narayanan, and Jishen Zhao, which is submitted for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 7

Conclusion

7.1 Summary of the Thesis

This thesis discusses the approaches to adopting NVM in computer architecture for modern applications. We propose customized designs for the data-intensive and neural network applications. For the data intensive applications, we first provide a characteristic study against the persistent memory applications. Our study provides a comprehensive analysis of the performance and hardware behavior of persistent memory applications. We also propose an error-resilient design for the data intensive application using NVM-based main memory. We redesign the cache writeback and NVM wear-leveling and coordinate them with the existing resilience schemes to significantly improve the reliability of the memory hierarchy. We introduce thermal aware chip management for the neural network applications to prevent the NVM-based NN accelerator from accuracy loss. We redesign the data forwarding in the pipeline and enable automated hardware assignment of the NVM-based NN accelerator to mitigate the data dependency caused underutilization and lack of flexibility. We conduct evaluations to prove each of the proposed designs in the dissertation to be efficient, reliable, and scalable for carrying the correlated modern application.

7.2 Future Works

The Binary Star design can be applied to shared memory in a distributed system. When the shared memory receives updates from RDMA access, the NVM-based shared memory may treat them in the same way as consistent cache writeback. The node that maintains the NVM-based shared memory holds a consistent checkpoint. If errors happen within the memory of any other node, it can recover with the data from the NVM-based shared memory.

The idea of coordinating reliability schemes among NVM and other devices can further extend to the block devices. The reliability schemes among NVM and SSD may use a unified logic-to-physical address translation to handle the wear-leveling of both devices together. Such design needs to coordinate different block sizes and utilizes different endurance cycles of two devices. The design would extend the reliability of the storage devices and further enhance the entire memory hierarchy.

Aside from thermal-induced error, the NVM-based NN accelerator also needs prevention against other sources of errors. The other sources of errors in NVM include row hammer [227], diffusion of oxygen vacancies [228], devices yeild [229], and etc. These errors could potentially harm the inference accuracy as the thermal issue. The mechanisms in the HR^{3}AM may be applicable as a countermeasure for other sources of errors as well.

Moreover, the NVM-related errors may cause different outcomes on other neural network applications. Applications such as image generation based on generative adversarial network (GAN) and robotics control based on reinforcement learning do not use accuracy for evaluation. We can quantify the effect of NVM-related errors on these applications with different metrics. We may propose customized countermeasure based on sources of the errors and characteristics of the applications in the future.

Bibliography

[1] S. Cha, S. O, H. Shin, S. Hwang, K. Park, S. J. Jang, J. S. Choi, G. Y. Jin, Y. H. Son, H. Cho, J. H. Ahn, and N. S. Kim, "Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices," in *HPCA*, 2017.

[2] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *ASPLOS*, 2015.

[3] S. Swami, P. M. Palangappa, and K. Mohanram, "ECS: Error-correcting strings for lifetime improvements in nonvolatile memories," *ACM TACO*, 2017.

[4] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing on-die error detection information for strong memory reliability," in *ISCA*, 2016.

[5] D. Agrawal, A. El Abbadi, S. Antony, and S. Das, "Data management challenges in cloud computing infrastructures," in *Databases in Networked Information Systems*, S. Kikuchi, S. Sachdeva, and S. Bhalla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–10.

[6] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017. [Online]. Available: https://books.google.com/books?id=zFheDgAAQBAJ

[7] A. Chen, "A review of emerging non-volatile memory (NVM) technologies and applications," *Solid-State Electronics*, vol. 125, pp. 25–38, 2016, extended papers selected from ESSDERC 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0038110116300867

[8] Intel and Micron, "Intel and Micron produce breakthrough memory technology," 2015, http://newsroom.intel.com/community/intel_newsroom/.

[9] Intel, "Intel architecture instruction set extensions programming reference," 2015, https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[10] Intel, "Persistent memory file system," https://github.com/linux-pmfs/pmfs.

[11] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 323–338.

[12] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: Association for Computing Machinery, 2017, p. 460–477.

[13] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and rdma-capable networks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 221–234.

[14] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and availability via client-local NVM in a distributed file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1011–1027.

[15] S. Scargall, *Introducing the Persistent Memory Development Kit*. Berkeley, CA: Apress, 2020, pp. 63–72.

[16] C. C. Chou, J. Jung, A. L. N. Reddy, P. V. Gratz, and D. Voigt, "vnvml: An efficient user space library for virtualizing and sharing non-volatile memories," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, 2019, pp. 103–115.

[17] T. Shull, J. Huang, and J. Torrellas, "Autopersist: An easy-to-use java nvm framework based on reachability," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 316–332.

[18] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 897–912.

[19] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 258–270.

[20] R. Elkhouly, M. Alshboul, A. Hayashi, Y. Solihin, and K. Kimura, "Compiler-support for critical data persistence in NVM," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019.

[21] S. Yu and P. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[22] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *International Symposium on Computer Architecture*, 2009, pp. 2–13.

[23] J. Liang, R. G. D. Jeyasingh, H. Chen, and H. . P. Wong, "A 1.4μa reset current phase change memory cell with integrated carbon nanotube electrodes for cross-point memory application," in *2011 Symposium on VLSI Technology - Digest of Technical Papers*, 2011, pp. 100–101.

[24] C. Villa, D. Mills, G. Barkley, H. Giduturi, S. Schippers, and D. Vimercati, "A 45nm 1Gb 1.8V phase-change memory," in *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2010, pp. 270–271.

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

[26] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong, "A 20nm 1.8 v 8Gb PRAM with 40MB/s program bandwidth," in *IEEE International Solid - State Circuits Conference - (ISSCC)*, 2012.

[27] K. Bourzac, "Has intel created a universal memory technology? [news]," *IEEE Spectrum*, vol. 54, no. 5, pp. 9–10, May 2017.

[28] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 50–61.

[29] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2013, pp. 256–267.

[30] S. P. Park, S. Gupta, N. Mojumder, A. Raghunathan, and K. Roy, "Future cache design using STT MRAMs for improved energy efficiency: Devices, circuits and architecture," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 492–497. [Online]. Available: https://doi.org/10.1145/2228360.2228447

[31] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, 2013, pp. 421–432.

[32] P. Chi, S. Li, S. H. Kang, and Y. Xie, "Architecture design with STT-RAM: Opportunities and challenges," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 109–114.

[33] L. Xia, T. Tang, W. Huangfu, M. Cheng, X. Yin, B. Li, Y. Wang, and H. Yang, "Switched by input: Power efficient structure for RRAM-Based convolutional neural network," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE Press, 2016, p. 1–6.

[34] I. Zhang, A. Szekeres, D. V. Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy, "Customizable and extensible deployment for mobile/cloud applications," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zhang

[35] C. Philip Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information Sciences*, vol. 275, pp. 314–347, 2014.

[36] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. Netto, and R. Buyya, "Big data computing and clouds: Trends and future directions," *Journal of Parallel and Distributed Computing*, vol. 79-80, pp. 3–15, 2015, special Issue on Scalable Systems for Big Data Management and Analytics.

[37] O. Mutlu, *Main Memory Scaling: Challenges and Solution Directions*. New York, NY: Springer New York, 2015, pp. 127–153.

[38] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. Hung Byers, *Big data: The next frontier for innovation, competition, and productivity*. McKinsey Global Institute, 2011.

[39] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi, "Big data and its technical challenges," *Communications of the ACM*, vol. 57, no. 7, pp. 86–94, 2014.

[40] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory positional effects in DRAM and SRAM faults," in *SC*, 2013.

[41] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *SC*, 2012.

[42] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *ASPLOS*, 2017.

[43] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," vol. 41, no. 1, 2016, pp. 6–12.

[44] J. Axboe, "Flexible I/O tester," 2016, https://github.com/axboe/fio.

[45] T. Bray, "Bonnie64," https://code.google.com/archive/p/bonnie-64/.

[46] Oracle, "MySQL," 2019, https://www.mysql.com/.

[47] M. Hirabayashi, "Tokyo cabinet: a modern implementation of DBM," 2010. [Online]. Available: http://fallabs.com/tokyocabinet/

[48] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[49] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016.

[50] JEDEC, "JEDEC publishes DDR4 NVDIMM-P bus protocol standard," 2021, https://www.jedec.org/news/pressreleases/jedec-publishes-ddr4-nvdimm-p-bus-protocol-standard.

[51] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019.

[52] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182.

[53] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 496–508.

[54] M. Awasthi, M. Shevgoor, K. Sudan, B. Rajendran, R. Balasubramonian, and V. Srinivasan, "Efficient scrub mechanisms for error-prone emerging memories," in *HPCA*, 2012.

[55] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system," *ISCA*, 2013.

[56] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *ISCA*, 2010.

[57] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "NVM Duet: Unified working memory and persistent store architecture," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 455–470.

[58] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient Hardware-based Undo+Redo Logging for Persistent Memory Systems," in *HPCA*, 2018.

[59] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465.

[60] Intel, "Enabling Persistent Memory in the Storage Performance Development Kit (SPDK)," 2019, https://software.intel.com/content/www/us/en/develop/articles/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html.

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

[62] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. New York, NY, USA: Association for Computing Machinery, 2014, p. 433–452.

[63] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 427–442. [Online]. Available: https://doi.org/10.1145/2872362.2872410

[64] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 133–146.

[65] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an rdma-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 773–785. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[66] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 494–508.

[67] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014, pp. 15:1–15:15.

[68] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[69] Y. Zhang and S. Swanson, “A study of application performance with non-volatile main memory,” in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 2015, pp. 1–10.

[70] M. Patel, J. S. Kim, H. Hassan, and O. Mutlu, “Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices,” in *DSN*, 2019.

[71] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, “The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study,” in *SIGMETRICS*, 2014.

[72] M. K. Qureshi, D. Kim, S. Khan, P. J. Nair, and O. Mutlu, “AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems,” in *DSN*, 2015.

[73] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, “PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2019, pp. 715–731.

[74] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 14–26.

[75] 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 *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 27–39.

[76] L. Song, X. Qian, H. Li, and Y. Chen, “PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 541–552.

[77] T. Chou, W. Tang, J. Botimer, and Z. Zhang, “CASCADE: Connecting RRAMs to Extend Analog Dataflow In An End-To-End In-Memory Processing Paradigm,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 114–125.

[78] Y. Ji, Y. Zhang, X. Xie, S. Li, P. Wang, X. Hu, Y. Zhang, and Y. Xie, “FPSA: A full system stack solution for reconfigurable ReRAM-Based NN accelerator architecture,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 733–747.

[79] S. Yu, “Neuro-inspired computing with emerging nonvolatile memorys,” *Proceedings of the IEEE*, vol. 106, no. 2, pp. 260–285, 2018.

[80] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, “Training and operation of an integrated neuromorphic network based on metal-oxide memristors,” *Nature*, vol. 521, no. 7550, pp. 61–64, 2015.

[81] G. W. Burr, R. M. Shelby, S. Sidler, C. di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti, B. N. Kurdi, and H. Hwang, “Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element,” *IEEE Transactions on Electron Devices*, vol. 62, no. 11, pp. 3498–3507, 2015.

[82] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec 2017.

[83] H. Li, H. Y. Chen, Z. Chen, B. Chen, R. Liu, G. Qiu, P. Huang, F. Zhang, Zizhen Jiang, B. Gao, L. Liu, X. Liu, S. Yu, H. . S. P. Wong, and J. Kang, “Write disturb analyses on half-selected cells of cross-point RRAM arrays,” in *2014 IEEE International Reliability Physics Symposium*, 2014, pp. MY.3.1–MY.3.4.

[84] S. Li, W. Chen, Y. Luo, J. Hu, P. Gao, J. Ye, K. Kang, H. Chen, E. Li, and W. Y. Yin, “Fully coupled multiphysics simulation of crosstalk effect in bipolar resistive random access memory,” *IEEE Transactions on Electron Devices*, vol. 64, no. 9, pp. 3647–3653, 2017.

[85] P. Mehra and S. Fineberg, “Fast and flexible persistence: the magic potion for fault-tolerance, scalability and performance in online data stores,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2004, pp. 206–218.

[86] R. Patel, X. Guo, Q. Guo, E. Ipek, and E. G. Friedman, “Reducing switching latency and energy in STT-MRAM caches with field-assisted writing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 129–138, 2016.

[87] V. Sousa, “Phase change materials engineering for RESET current reduction,” in *Proceedings of the Memory Workshop*, 2012.

[88] C. Cagli, “Characterization and modelling of electrode impact in HfO2-based RRAM,” in *Proceedings of the Memory Workshop*, 2012.

[89] T. C. Bressoud, T. Clark, and T. Kan, “The design and use of persistent memory on the DNCP hardware fault-tolerant platform,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2001, pp. 487–492.

[90] M. Wilcox, “Add support for NV-DIMMs to ext4,” https://lwn.net/Articles/613384/.

[91] Intel, “Intel Optane DC Persistent Memory,” 2019, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[93] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[94] Intel, "A collection of linux persistent memory programming examples," https://github.com/pmem/linux-examples.

[95] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011, pp. 2–2.

[96] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A case for efficient hardware/software cooperative management of storage and memory," in *Proceedings of the 5th Workshop on Energy-Efficient Design (WEED), held in conjunction with the International Symposium on Computer Architecture (ISCA-40)*, 2013.

[97] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, no. 1, pp. 143–143, Jan. 2010.

[98] W. Zhao, E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle, "Macro-model of spin-transfer torque based magnetic tunnel junction device for hybrid magnetic-CMOS design," in *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, Sept 2006, pp. 40–43.

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

[100] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 167–181.

[101] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 5–5.

[102] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014, pp. 14:1–14:14.

[103] X. Wu and A. Reddy, "SCMFS: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 39.

[104] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*. New York, NY, USA: ACM, 2016, pp. 12:1–12:16.

[105] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, 2015, p. 707–722.

[106] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi, "SLM-DB: Single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, 2019, pp. 191–205.

[107] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and high-performance memory control for persistent memory systems," in *Proceedings of the 47th International Symposium on Microarchitecture (MICRO-47)*, 2014.

[108] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *MICRO*, 2015.

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

[110] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the International Symposium on Computer Architecture*, 2014, pp. 1–12.

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

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

[113] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Phase change memory architecture and the quest for scalability," *Commun. ACM*, vol. 53, no. 7, pp. 99–106, Jul. 2010.

[114] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *International Conference on Computer Design*, 2012, pp. 1–8.

[115] D. Sengupta, Q. Wang, H. Volos, L. Cherkasova, J. Li, G. Magalhaes, and K. Schwan, "A framework for emulating non-volatile memory systemswith different performance characteristics," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 317–320.

[116] A. C. De Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[117] D. Heger, J. Jacobs, and S. Rao, "Flexible file system benchmark," https://sourceforge.net/projects/ffsb/.

[118] Linux man-pages project, "open(2) – linux programmer's manual," 2016, http://man7.org/linux/man-pages/man2/open.2.html.

[119] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: Association for Computing Machinery, 2009, p. 14–23.

[120] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2005, pp. 190–200.

[121] The Apache Software Foundation, "Spark," 2014, http://spark.incubator.apache.org/.

[122] VoltDB, "VoltDB: Smart data fast," 2014, http://voltdb.com/.

[123] J. Barr, "EC2 in-memory processing update: Instances with 4 to 16 TB of memory and scale-out SAP HANA to 34 TB," 2017.

[124] SAP, "SAP HANA: an in-memory, column-oriented, relational database management system," 2014, http://www.saphana.com/.

[125] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[126] O. Mutlu, "Memory scaling: A systems architecture perspective," in *IMW*, 2013.

[127] T. Oh, H. Chung, J. Park, K. Lee, S. Oh, S. Doo, H. Kim, C. Lee, H. Kim, J. Lee, J. Lee, K. Ha, Y. Choi, Y. Cho, Y. Bae, T. Jang, C. Park, K. Park, S. Jang, and J. S. Choi, "A 3.2 Gbps/pin 8 Gbit 1.0 V LPDDR4 SDRAM With Integrated ECC Engine for Sub-1 V DRAM Core Operation," *IEEE Journal of Solid-State Circuits*, 2015.

[128] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *DSN*, 2014.

[129] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *DSN*, 2015.

[130] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," in *SIGMETRICS*, 2009.

[131] "Data-centric innovation spotlight series: Big memory breakthrough for your biggest data challenges," 2019, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[132] Intel, "Intel launches Optane DIMMs up to 512GB: Apache Pass is here!" 2018, https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here.

[133] Intel, "An intro to MCDRAM (high bandwidth memory) on Knights Landing," 2016, https://software.intel.com/en-us/blogs/2016/01/20/an-intro-to-mcdram-high-bandwidth-memory-on-knights-landing.

[134] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombie memory: Extending memory lifetime by reviving dead blocks," in *ISCA*, 2013.

[135] Intel, "PMDK: Persistent memory development kit," 2019, https://github.com/pmem/pmdk/.

[136] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *ASPLOS*, 2017.

[137] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[138] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE TCAD*, 2019.

[139] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," in *ASPLOS*, 2012.

[140] D. S. Yaney, C. Y. Lu, R. A. Kohler, M. J. Kelly, and J. T. Nelson, "A meta-stable leakage phenomenon in DRAM charge storage - Variable hold time," in *IEDM*, 1987.

[141] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms," in *ISCA*, 2013.

[142] M. Patel, J. S. Kim, and O. Mutlu, "The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions," in *ISCA*, 2017.

[143] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *ISCA*, 2013.

[144] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi, "Co-architecting controllers and DRAM to enhance DRAM process scaling," in *The Memory Forum*, 2014.

[145] U. Kang, H. Chung, S. Heo, D. Park, H. Lee, J. H. Kim, S. Ahn, S. Cha, J. Ahn, D. Kwon, J. Lee, H. Joo, W. Kim, D. H. Jang, N. S. Kim, J. Choi, T. Chung, J. Yoo, J. S. Choi, C. Kim, and Y. Jun, "8 Gb 3-D DDR3 DRAM Using Through-Silicon-Via Technology," *IEEE Journal of Solid-State Circuits*, 2010.

[146] J. Sim, G. H. Loh, V. Sridharan, and M. O'Connor, "Resilient die-stacked DRAM caches," in *ISCA*, 2013.

[147] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently protecting stacked memory from TSV and large granularity failures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–24, 2016.

[148] H.-M. Chen, C.-J. Wu, T. Mudge, and C. Chakrabarti, "RATT-ECC: Rate Adaptive Two-Tiered Error Correction Codes for Reliable 3D Die-Stacked Memory," *ACM TACO*, 2016.

[149] G. Mappouras, A. Vahid, R. Calderbank, D. R. Hower, and D. J. Sorin, "Jenga: Efficient Fault Tolerance for Stacked DRAM," in *ICCD*, 2017.

[150] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *ISPASS*, 2013.

[151] Micron, "3D XPoint Technology: Breakthrough Nonvolatile Memory Technology," 2016, https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[152] K. Deierling, "Persistent memory over fabric: let it out of the box!" in *Open Server Summit*, 2016.

[153] K. Suzuki and S. Swanson, "The Non-Volatile Memory Technology Database (NVMDB)," 2015, http://nvmdb.ucsd.edu.

[154] N. Christiansen, "Storage class memory support in the Windows OS," in *SNIA NVM Summit*, 2016.

[155] C. Diaconu, "Microsoft SQL Hekaton – Towards Large Scale Use of PM for In-memory Databases," in *SNIA NVM Summit*, 2016.

[156] J. Moyer, "Persistent memory in Linux," in *SNIA NVM Summit*, 2016.

[157] K. Deierling, "Persistent memory over fabric," in *SNIA NVM Summit*, 2016.

[158] Z. Zhang, W. Xiao, N. Park, and D. J. Lilja, "Memory module-level testing and error behaviors for phase change memory," in *ICCD*, 2012.

[159] W. C. Chien, H. Y. Cheng, M. BrightSky, A. Ray, C. W. Yeh, W. Kim, R. Bruce, Y. Zhu, H. Y. Ho, H. L. Lung, and C. Lam, "Reliability study of a 128Mb phase change memory chip implemented with doped Ga-Sb-Ge with extraordinary thermal stability," in *IEDM*, 2016.

[160] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "FREE-p: Protecting non-volatile memory against both hard and soft errors," in *HPCA*, 2011.

[161] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 14–23.

[162] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, "Reliability-aware data placement for heterogeneous memory architecture," in *HPCA*, 2018.

[163] S. Kim, "Area-efficient error protection for caches," in *DATE*, 2006.

[164] D. H. Yoon and M. Erez, "Memory mapped ECC: low-cost error protection for last level caches," in *ISCA*, 2009.

[165] D. H. Yoon and M. Erez, "Flexible cache error protection using an ECC FIFO," in *SC*, 2009.

[166] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *MICRO*, 2016.

[167] P. Koopman and T. Chakravarty, "Cyclic redundancy code (CRC) polynomial selection for embedded networks," in *DSN*, 2004.

[168] 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 *HPCA*, 2010.

[169] Intel, "Over-provisioning nand-based intel SSDs for better endurance," 2018, https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/over-provisioning-nand-based-ssds-better-endurance-whitepaper.pdf.

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

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

[172] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” in *ISCA*, 2000.

[173] W. K. Zuravleff and T. Robinson, “Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order,” 1997.

[174] Micron, “Hybrid memory cube specification 2.1,” 2014, http://www.hybridmemorycube.org/.

[175] P. J. Nair, D. A. Roberts, and M. K. Qureshi, “FaultSim: A fast, configurable memory-reliability simulator for conventional and 3D-Stacked systems,” *ACM TACO*, 2015.

[176] H. Jeon, G. H. Loh, and M. Annavaram, “Efficient RAS support for die-stacked DRAM,” in *ITC*, 2014.

[177] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[178] B. Fitzpatrick, “Distributed caching with memcached,” *Linux journal*, 2004.

[179] T. Council, “tpc-c benchmark, revision 5.11,” 2010.

[180] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *SoCC*, 2010.

[181] A. Kopytov, “Sysbench manual,” *MySQL AB*, 2012.

[182] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *PACT*, 2008.

[183] D. Roberts and P. Nair, “FAULTSIM: A fast, configurable memory-resilience simulator,” in *The Memory Forum: In conjunction with ISCA*, 2014.

[184] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams, “Dot-product engine for neuromorphic computing: Programming 1t1m crossbar to accelerate matrix-vector multiplication,” in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[185] C. Walczyk, D. Walczyk, T. Schroeder, T. Bertaud, M. Sowinska, M. Lukosius, M. Fraschke, D. Wolansky, B. Tillack, E. Miranda, and C. Wenger, “Impact of temperature on the resistive switching behavior of embedded HfO_2-based rram devices,” *IEEE TEC*, vol. 58, no. 9, pp. 3124–3131, Sep. 2011.

[186] B. Liu, H. Li, Y. Chen, X. Li, Q. Wu, and T. Huang, "Vortex: Variation-aware Training for Memristor X-bar," in *DAC*. New York, NY, USA: ACM, 2015, pp. 15:1–15:6.

[187] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *DAC*, June 2017, pp. 1–6.

[188] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, "Accelerator-friendly Neural-network Training: Learning Variations and Defects in RRAM Crossbar," in *DATE*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2017, pp. 19–24.

[189] M. V. Beigi and G. Memik, "Thermal-aware Optimizations of reRAM-based Neuromorphic Computing Systems," in *DAC*. New York, NY, USA: ACM, 2018, pp. 39:1–39:6.

[190] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, p. 075201, jan 2012.

[191] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory," in *ACM/IEEE ISCA*, June 2016, pp. 380–392.

[192] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[193] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[194] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[195] E. Rotem, J. Hermerding, A. Cohen, and H. Cain, "Temperature measurement in the Intel (R) CoreTM Duo Processor," *arXiv preprint arXiv:0709.1861*, 2007.

[196] Wei Huang, S. Ghosh, S. Velusamy, K. Sankaranarayanan, K. Skadron, and M. R. Stan, "HotSpot: A compact thermal modeling methodology for early-stage VLSI design," *IEEE TVLSI*, vol. 14, no. 5, pp. 501–513, 2006.

[197] A. Agrawal, J. Torrellas, and S. Idgunji, "Xylem: enhancing vertical thermal conduction in 3D processor-memory stacks," in *MICRO 2017*. ACM, 2017, pp. 546–559.

[198] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[199] Y. LeCun, “The MNIST database of handwritten digits,” *http://yann.lecun.com/exdb/mnist/*, 1998.

[200] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *IEEE CVPR*, vol. 00, June 2018, pp. 248–255.

[201] D. Ielmini and H.-S. P. Wong, “In-memory computing with resistive switching devices,” *Nature Electronics*, vol. 1, no. 6, pp. 333–343, 2018.

[202] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 336–348.

[203] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, “Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories,” in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC ’16. New York, NY, USA: Association for Computing Machinery, 2016.

[204] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, “Google workloads for consumer devices: Mitigating data movement bottlenecks,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: Association for Computing Machinery, 2018, p. 316–331.

[205] Y. Ji, Y. Zhang, W. Chen, and Y. Xie, “Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 448–460.

[206] X. Qiao, X. Cao, H. Yang, L. Song, and H. Li, “Atomlayer: A universal ReRAM-Based CNN accelerator with atomic layer computation,” in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC ’18. New York, NY, USA: Association for Computing Machinery, 2018.

[207] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a

tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. New York, NY, USA: ACM, 2017, pp. 1–12.

[208] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan 2017.

[209] 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)*. New York, NY, USA: ACM, 2014, pp. 269–284.

[210] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622.

[211] ONNX Project, "Open neural network exchange," 2019, https://onnx.ai/.

[212] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, 2019, pp. 8024–8035.

[213] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[214] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," *SIGPLAN Not.*, vol. 53, no. 2, p. 1–14, Mar. 2018.

[215] L. Xia, B. Li, T. Tang, P. Gu, P. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "MNSIM: Simulation platform for memristor-based neuromorphic computing system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1009–1022, 2018.

[216] P. Chen, X. Peng, and S. Yu, "Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 12, pp. 3067–3080, 2018.

[217] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph lowering compiler techniques for neural networks," 2018.

[218] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[219] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design," in *The European Conference on Computer Vision (ECCV)*, September 2018.

[220] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[221] Torch Contributors, "torchvision – pytorch master documentation," 2019, https://pytorch.org/docs/stable/torchvision/index.html#torchvision.

[222] Synopsys, "Teaching resources for ic design," 2020, https://www.synopsys.com/community/university-program/teaching-resources.html.

[223] B. Murmann, "ADC performance survey 1997-2011," 2011, http://www.stanford.edu/˜murmann/adcsurvey.html.

[224] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Braendli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici, "A 3.1mW 8b 1.2GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32nm digital SOI CMOS," in *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, Feb 2013, pp. 468–469.

[225] M. Saberi, R. Lotfi, K. Mafinezhad, and W. A. Serdijn, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation adcs," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 8, pp. 1736–1748, Aug 2011.

[226] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[227] M. N. I. Khan and S. Ghosh, "Analysis of row hammer attack on STT-RAM," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 75–82.

[228] A. M. S. Tosson, S. Yu, M. H. Anis, and L. Wei, "Analysis of RRAM reliability soft-errors on the performance of RRAM-Based neuromorphic systems," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2017, pp. 62–67.

[229] C. Chen, H. Shih, C. Wu, C. Lin, P. Chiu, S. Sheu, and F. T. Chen, "RRAM defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 180–190, 2015.