UNIVERSITY OF CALIFORNIA

Los Angeles

Lightweight Opportunistic Memory Resilience

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Electrical and Computer Engineering

by

Irina Alam

2021

ABSTRACT OF THE DISSERTATION

Lightweight Opportunistic
Memory Resilience

by

Irina Alam
Doctor of Philosophy in Electrical and Computer Engineering
University of California, Los Angeles, 2021
Professor Puneet Gupta, Chair

The reliability of memory subsystems is worsening rapidly and needs to be considered as one of the primary design objectives when designing today's computer systems. From on-chip embedded memories in Internet-of-Things (IoT) devices and on-chip caches to off-chip main memories, they have become the limiting factor in the reliability of these computing systems. Today's applications demand large capacity of on-chip or off-chip memory or both. With aggressive technology scaling, coupled with the increase in the total area devoted to memory in a chip, memories are becoming particularly sensitive to manufacturing process variation, environmental operating conditions, and aging-induced wearout. However, the challenge with memory reliability is that the resiliency techniques need to be effective but with minimal overhead. Today's typical error correcting schemes do not take into consideration the data value that they are protecting and are purely based on positional errors. This increases their overheads and makes them too expensive, especially for on-chip memories. Also, the drive for denser off-chip main memories is worsening their reliability. But strengthening today's error correction techniques will result in non-negligible increase in overheads. Hence, this dissertation proposes *Lightweight Opportunistic Memory Resilience*. We exploit the following three aspects to make memories more reliable with low overheads: (1) Underlying memory fault models, (2) Data value behavior of commonly used applications, and (3) The architecture of the memory itself. We opportunistically exploit these three aspects to provide stronger protection against memory errors. We design novel error detecting and correcting codes and develop several other architectural fault tolerance techniques at minimal overheads compared to

the conventional reliability techniques used in today's memories.

In part 1 of this dissertation, we address the reliability concerns in lightweight on-chip caches or embedded memories like scratchpads in IoT devices. These memories are becoming larger in size, but needs to be low power. Using standard error correcting codes or traditional row/column sparing to recover from faults are too expensive for them. Here, we leverage the fact that manufacturing defects and aging-induced hard faults usually only affect only a few bits in a memory. These bits, however, inhibit how low of a voltage these chips can be operated at. Traditional software fails even when a small number of bits in a memory are faulty. For the first time, we provide two solutions, *FaultLink* and *SAME-Infer*, which help deal with these weak faulty cells in the memory by generating a custom-tailored fault-aware application binary image for each chip. Next, we designed *Software-Defined Error Localization Code (SDELC)* and *Parity++* as lightweight runtime error recovery techniques that leverage the insight that data values have locality in them and certain ranges of data values occur more frequently than others. Conventional ECC is too expensive for these lightweight memories. SDELC uses novel ultra-lightweight error-localizing codes to localize the error to a chunk in the data. It then heuristically recovers from the localized error by exploiting side information about the application's memory contents. Parity++ is a novel unequal message protection scheme that preferentially provides stronger error protection to certain "special messages". This protection scheme provides Single Error Detection (SED) for all messages and Single Error Correction (SEC) for a subset of special messages. Both these novel codes utilize data value behavior to provide single error correction at 2.5x-4x lower overhead than a conventional hamming single error correcting code.

In part 2 of this dissertation, we focus on off-chip main memory technologies. We primarily leverage the details of the memory architecture itself and their dominant fault mechanisms to effectively design reliability schemes. The need for larger main memory capacity in today's workstation or server environments is driving the use of non-volatile memories (NVM) or techniques to enable high density DRAMs. Due to aggressive scaling, the single-bit error rate in DRAMs is steadily increasing and DRAM manufacturers are adopting on-die error correction coding (ECC) schemes, along with within memory controller ECC, to correct single-bit errors in the memory. In *COMET* we have shown that today's standard on-die ECCs can lead to silent data corruption if not

designed correctly. We propose a collaborative on-die and in-controller error correction scheme that prevents double-bit error induced silent data corruption and corrects 99.9997% of these double-bit errors at absolutely no additional storage, latency, and area overheads. Not just DRAMs, reliability is a major concern in most of the emerging NVM technologies. In *Compression with Multi-ECC (CME)*, we propose a new opportunistic compression-based ECC protection scheme for magnetic memory-based main memories. CME compresses every memory line and uses the saved bits to add stronger protection. In some of these NVMs, error rates increase as we try to improve read/write latencies. In *PCM-Duplicate*, we propose an enhanced PCM architecture that reduces PCM read latency by more than 3x and makes it comparable to that of DRAM. We then use ECC to tolerate the additional errors that arise because of the proposed optimizations.

Overall, we have developed a complementary suite of novel methods for tolerating faults and correcting errors in different levels of the memory hierarchy. We exploit the memory architecture and fault mechanisms as well as the application data behavior to tune the proposed solutions to the particular memory characteristics; lightweight solutions for low-cost embedded memories and latency-critical on-chip caches while stronger protection for off-chip main memory subsystems. With memory reliability being a major bottleneck in today's systems, these novel solutions are expected to alleviate this problem, help cope with the unique outcomes of hardware variability in memory systems and provide improved reliability at minimal cost.

The dissertation of Irina Alam is approved.

Tony Nowatzki

Chih-Kong Ken Yang

Lara Dolecek

Puneet Gupta, Committee Chair

University of California, Los Angeles

2021

To my parents who inspire me,

and without whom none of this would have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation is the culmination of many years of hard work, the support and guidance of my advisor and the doctoral committee, and the love and encouragement of my family and friends. This journey of five years has not always been easy. There were times of joy and satisfaction, along with phases of struggles and disappointment. But overall this experience has been extremely fulfilling and I owe gratitude to a number of people without whom my doctoral studies would not have been possible.

I would first like to thank my doctoral committee: Profs. Puneet Gupta (chair and advisor), Lara Dolecek, Ken Yang, and Tony Nowatzki for their outstanding teaching and research guidance. Prof. Gupta has been a wonderful advisor and mentor. I am grateful to him for the constant support, patience, motivation, and guidance. This dissertation would not have been possible without his excellent help and foresight. He helped me in becoming a successful researcher. I would like to thank Prof. Dolecek for being an instrumental collaborator who introduced me to the fundamentals of coding theory. Most of the dissertation stands on the knowledge that I gained from my collaboration with her. I am grateful to Prof. Yang for his valuable inputs, especially in the PCM project. He highlighted key issues that were unknown to me and that significantly helped in improving the project. Prof. Nowatzki introduced me to interesting research topics in computer architecture through his graduate course. Discussions with him during the course and beyond have inspired me to broaden my research focus and look at new problems.

I would like to thank Dr. Mark Gottscho (now at Google) for being an excellent mentor and helping me complete my very first project (FaultLink and SDELC) as a Ph.D. student. He held my hand through my very crucial first year of research at UCLA that helped in forming and shaping my dissertation focus. I would like to thank my collaborator, Dr. Clayton Schoeny (now at Square) for all the discussions and technical help. We collaborated on most of the projects in the first part of my dissertation (FaultLink, SDELC, and Parity++). Without his novel code construction ideas, these projects would not have been possible.

I would then like to thank all my internship mentors. At Micron Technology, Inc. I worked under the guidance of Ameen Akel and Ken Curewitz. The knowledge and insights from this internship

immensely helped me with my research in memory architecture and resiliency. At Google LLC., I worked with Hema Hariharan and Mark Gottscho to build a framework for developing memory reliability solutions. This project helped me implement some of my research ideas and gave me the satisfaction of seeing my hard work become a part of Google's chip building infrastructure. At Apple, Inc. my manager Seung Lee and mentors Heonjae Ha, Xiaowen Han, and Andrew Chu introduced me to a completely new area of memory research (not including details because of confidentiality concerns). All three internships have been extremely rewarding in terms of the knowledge that I have gained and have contributed significantly towards my doctoral research. Thanks also to Dr. Greg Wright from Qualcomm Research for inviting me to present at Qualcomm Research (San Diego) in 2017. His feedback and comments on my work have added valuable perspectives.

I would also like to acknowledge the support of my labmates: Dr. Mark Gottscho, Dr. Yasmine Badr, Dr. Shaodi Wang, Dr. Wei-Che Wang, Saptadeep Pal, Yoojin Chae, Tianmu Li, Wojciech Romaszkan, Shurui Li, and Alexander Graening. I am thankful for their friendship and their inputs during technical discussions. Special thanks to all my friends in India, Singapore, and the US for their support. Thank you Yasmine Badr for helping me settle down and get past my first year at UCLA. I would also like to thank Steven Moran for motivating me to remain fit and reminding me to take occasional breaks to refresh and recharge. Finally, I will be forever grateful to Saptadeep Pal for being my biggest support in these five years, for always inspiring and motivating me, for critiquing my work and providing valuable inputs through lengthy discussions, and for proofreading all my papers.

Lastly and most importantly, I would like to thank my parents, Maksud Alam and Leena Alam, for their unconditional love, unwavering support, and encouragement. I can never find enough words to express my gratitude towards them for their sacrifices and for the key role that they played in shaping my future. None of my accomplishments would have been possible without them.

Copyrights and Re-use of Published Material

This dissertation contains significant material that has been previously published or is intended to be published in the future. Chapters 2 and 4 (FaultLink and SDELC) contain material that were published in [1, 4, 5]. Chapter 3 (SAME-Infer) includes material published in [6]. The fundamental concepts in Chapter 5 (Parity++) appeared in [4, 7, 8]. Chapters 6 (COMET) and 8 (PCM-Duplicate) are being prepared for publication. Chapter 7 (CME) includes material published in [9].

Some of the work in my PhD that was conducted in collaboration with other individuals (where I contributed, but did not lead) are not included in the body of this dissertation.

VITA

2014	B.Eng., Electrical and Electronic Engineering (Minor in Computing), Nanyang Technological University, Singapore
2014-2016	Product Engineer, Micron Semiconductor Asia Pte. Ltd., Singapore
2017, 2018	UCLA EE Department Fellowship
2018	M.S., Electrical and Computer Engineering, UCLA
2018	Outstanding Master's Thesis in Circuits and Embedded Systems, ECE, UCLA
2018	PhD Intern, Micron Technology Inc.
2019	Cadence Women in Technology Scholarship
2019	PhD Intern, Google LLC.
2020	PhD Intern, Apple Inc.

PUBLICATIONS

Saptadeep Pal, **Irina Alam**, Krutikesh Sahoo, Haris Suhail, Rakesh Kumar, Sudhakar Pamarti, Puneet Gupta, and Subramanian S. Iyer, "I/O Architecture, Substrate Design, and Bonding Process for a Heterogeneous Dielet-Assembly based Waferscale Processor", in *IEEE 71st Electronic Components and Technology Conference (ECTC)*, June 2021.

Irina Alam, Lara Dolecek, and Puneet Gupta, "Lightweight Software-Defined Error Correction for Memories", in *Dependable Embedded Systems*, Springer, Cham, 2021.

Irina Alam, and Puneet Gupta "SAME-Infer: Software Assisted Memory Resilience for Efficient Inference at the Edge", in *International Symposium on Memory Systems(MEMSYS)*, Washington D.C., USA, September 2020.

Clayton Schoeny, Frederic Sala, Mark Gottscho, **Irina Alam**, Puneet Gupta, and Lara Dolecek, "Context-Aware Resiliency: Unequal Message Protection for Random-Access Memories", in *IEEE Transactions on Information Theory*, October 2019.

Irina Alam, Saptadeep Pal, and Puneet Gupta, "Compression with Multi-ECC: Enhanced Error Resiliency for Magnetic Memories", in *International Symposium on Memory Systems(MEMSYS)*, Washington D.C., USA, September 2019.

Clayton Schoeny, **Irina Alam**, Mark Gottscho, Puneet Gupta, and Lara Dolecek, "Error Correction and Detection for Computing Memories Using System Side Information", in *IEEE Information Theory Workshop (ITW)*, Guangzhou, China, November 2018.

Irina Alam, Clayton Schoeny, Lara Dolecek, and Puneet Gupta, "Parity++: Lightweight Error Correction for Last Level Caches", in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Luxembourg City, Luxembourg, June 2018.

Clayton Schoeny, Frederic Sala, Mark Gottscho, **Irina Alam**, Puneet Gupta, and Lara Dolecek, "Context-Aware Resiliency: Unequal Message Protection for Random-Access Memories", in *IEEE Information Theory Workshop (ITW)*, Kaohsiung, Taiwan, November 2017.

Mark Gottscho, **Irina Alam**, Clayton Schoeny, Lara Dolecek, and Puneet Gupta "Low-Cost Memory Fault Tolerance for IoT Devices", in *ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), published in ACM Transactions on Embedded Computing Systems (TECS)*, Seoul, South Korea, October 2017.

CHAPTER 1

Introduction

Memories are one of the key bottlenecks in the performance, reliability, and energy efficiency of most computing systems. As computing systems have scaled over the decades, the need for memory systems where large amounts of data can be stored and retrieved efficiently has also risen rapidly. To achieve this, memory and storage systems have been scaled for maximum information density. Moore's Law has been the primary driver behind the phenomenal advances in computing capability of the past several decades. However, with technology scaling having reached the nanoscale era, integrated circuits, especially computing memories, are becoming increasingly sensitive to process variations leading to reliability and yield concerns.

The biggest challenge with memory reliability is that the resiliency techniques are expected to be effective while incurring minimal overheads. Today's standard error detection and correction schemes do not take into consideration multiple factors, such as underlying actual fault models and error mechanisms, application data value behavior as well memory architecture and protocols. As a result, they are unable to provide maximum protection possible within the area, power, and performance overhead budgets. *Lightweight Opportunistic Memory Resilience* opportunistically exploits these aspects to provide a complementary suite of novel methods that tackle memory reliability at different levels of the memory hierarchy while incurring minimal cost.

To contextualize and motivate the research in this dissertation, a brief overview of memory reliability challenges is provided in Section 1.1. The problems faced in the reliable operation of on-chip memories are outlined in Section 1.2. Sections 1.3 and 1.4 discuss the challenges faced in today's off-chip main memory technologies (DRAM as well as alternate emerging non-volatile memory technologies). *Lightweight Opportunistic Memory Resilience* provides a complementary suite of novel methods that tackle memory reliability at different levels of the memory hierarchy,

and is described in Section 1.5 summarizes all the research projects covered in this dissertation that provide efficient resiliency solutions tailored to the memory type. The overall framework helps to cope with the reliability and performance challenges that computing memory faces today and tomorrow.

1.1 Memory Reliability is Becoming a Key Concern

Memories have become the limiting factor in the reliability of computing systems [10] because they are primarily designed to maximize bit storage density. On one hand, with technology scaling, the memory cell dimensions are reducing that is allowing memory manufacturers to pack in more memory cells per unit area. This makes memories particularly sensitive to manufacturing process variation, environmental operating conditions, and aging-induced wearout [11, 12]. On the other hand, system designers are trying to accommodate as many memory modules as possible to increase the total memory capacity and improve overall system performance. This is increasing the total memory array area and the likelihood of defects affecting the memory as well as the number of memory module components in the overall system that can fail during operation. The combined effect of these two phenomena is significantly worsening memory subsystem reliability.

In warehouse-scale computers, hard faults in memories manifest as correctable/uncorrectable errors. These errors have become expensive culprits that cause machine crashes, corrupted data, security vulnerabilities, service disruption, and costly repairs and hardware servicing [10, 13]. Google has observed 70000 failures in time (FIT)/Mb in commodity on-chip DRAM memory, with 8% of modules affected per year [10], while Facebook has found that 2.5% of their servers have experienced memory errors per month [14]. The Blue Waters supercomputer had 8.2% of the dual in-line memory modules (DIMMs) (modules that contain multiple RAM chips) encounter an error over the course of a 261-day study [15]. These trends are expected to continue rising.

The concern with memory reliability is not limited to high-performance systems. With IoT/embedded devices increasingly becoming part of critical infrastructure and being deployed in failure-intolerant modes (e.g., cars), the development of inexpensive fault tolerance schemes for them has become important [16]. Also, with sensing and data-processing being one of the most

important use cases for edge devices, these devices are seeing increasing use of large memories. SRAM-based scratchpad memories are often the choice of memory architecture used in IoT devices. As demand for higher memory density increases, memory cells are shrunk using advanced technology nodes which, in turn, makes the memory cells more susceptible to both soft and hard faults. The need for low-power and hence lower operating voltage exacerbates the error rates further.The weak cells in the memory limit the voltage and energy scaling in these memories. These trends indicate that memory failures are likewise going to be critical for emerging edge/IoT computing devices as well.

Techniques adopted to ensure reliability in today's systems incur performance, power, and area overheads. Error correcting codes (ECC) are typically used to recover from memory errors. However, they explicitly ignore data values stored in the memory and are purely based on positional errors. As a result, they add significant overheads. Ideally, we want to ensure reliable memory operation while incurring negligible overheads. Hence, the solution has to be tailored to the type of memory and its position in the memory hierarchy. For latency/power critical on-chip caches or embedded memories, tolerable overhead is much smaller than that for off-chip memories. Hence, conventional ECCs are too expensive for them. Also, systems running moderately fault-tolerant applications do not require strong protection techniques that reliability critical systems do. In workstation/server environments, need for larger amounts of memory is driving use of dense non-volatile memories or techniques to enable high density DRAMs. These memory technologies have significant reliability concerns and hence, require much stronger protection. Hence, in this dissertation, we take into account all these factors and provide a suite of novel memory resiliency techniques for different classes and types of memory technologies. We exploit data value behavior to design novel ECC solutions with much reduced overheads as compared to conventional codes. For embedded memories, our lightweight solutions provide protection against both hard faults discovered pre-deployment and unpredictable soft faults during runtime. For off-chip dense memories, we exploit the underlying fault models and memory architecture to improve reliability by providing stronger protection techniques with almost negligible increase in overheads.

1.2 Power/Performance Scaling and Fault Tolerance in Lightweight On-Chip SRAM-based Memories

Low power density is the key to achieving the vision of both exascale computing and the Internet of Things (IoT) [17]. To achieve that, systems need to adopt intelligent power-saving techniques. Memories, both on-chip and off-chip, consume a significant portion of system power. One way to reduce power consumption in on-chip SRAM-based memories is to reduce the supply voltage (VDD). However, as shown in Figure 1.1, scaling the VDD down leads to an exponential rise in hard faults in the memory cells [18]. Not only hard faults, but the memories also become more susceptible to radiation-induced soft faults at lower voltages, thus degrading yield at low voltage. Moreover, on-chip embedded memories or caches in high-performance computing systems are often the largest consumers of chip area. This further increases the likelihood of defects affecting memory rather than logic and process variations with respect to individual memory cells create a significant impact.

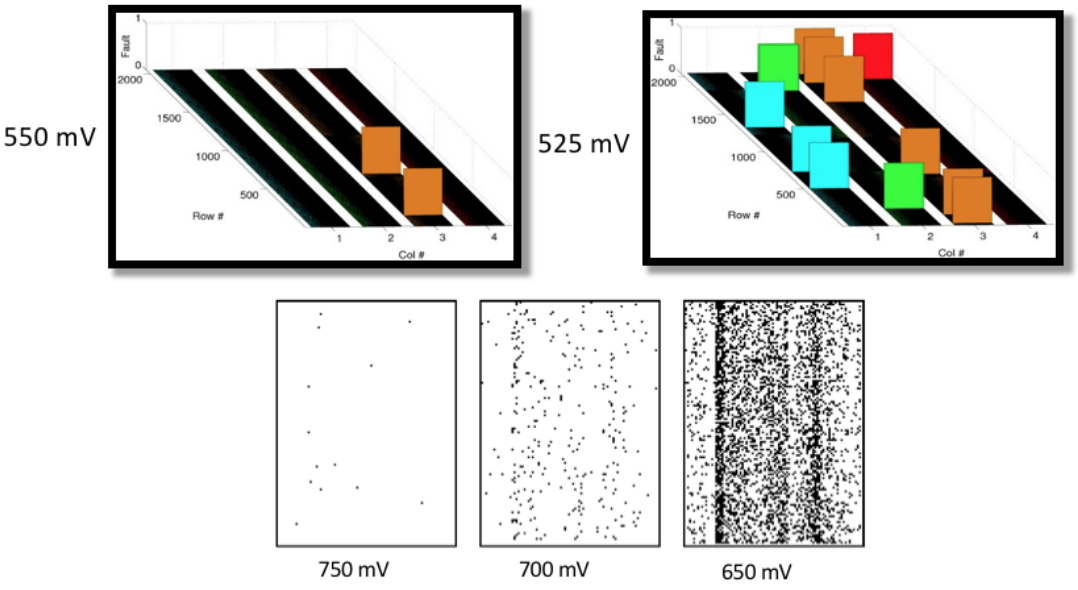


Figure 1.1: Faults in two SRAM based scratchpad memories at different voltages

To deal with on-chip memory errors due to manufacturing defects designers traditionally include spare rows and columns in the memory arrays [19] and employ large voltage guardbands [20]

to ensure reliable operation. Unfortunately, large guardbands limit the energy proportionality of memory. For unpredictable runtime bit flips, the widely used technique to guarantee the reliability of storage devices is using information redundancy in the form of Error Correcting Codes (ECC). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There are additional encoding (while writing data) and decoding (while reading data) procedures required as well. Thus, redundancy in ECC schemes not only incurs area overhead, the encoding and decoding mechanisms also incur additional overheads in terms of latency and energy. As a result, using strong ECC to correct errors in these latency/power critical lightweight memories often becomes overkill and has a non-negligible impact on overall system performance, power, and area. Thus, designing low overhead resiliency schemes to tolerate hard and soft faults in these memories is often a challenge. This dissertation proposes a two-step approach to improve the reliability of on-chip lightweight memories at minimal cost. In step one, we propose techniques to tolerate hard faults that are detected during deployment. In step two, we propose multiple correction mechanisms that allow recovery from unpredictable single-bit flips that occur during runtime. We exploit software behavior and characteristics to reduce the overall protection overheads.

1.3 Scalability Concerns in DRAM-based Main Memories

Memory reliability is a significant problem not just in on-chip memories, but also in off-chip main memory systems. Main memories serve a pivotal role, sitting in between the processor cores and the slow storage devices. With aggressive technology scaling, a large number of processor cores are being integrated in today's systems. As a result, there is an ever-increasing demand for main memory capacity in order to be able to exploit the processing power of these multicore and manycore systems and maintain the performance growth. DRAM is the primary main memory technology used in today's systems. However, DRAM scaling is, unfortunately, slowing down. Also, DRAM reliability is worsening with scaling. With increasing rate of scaling induced errors in DRAM [21–28], the traditional method of row/column sparing used by DRAM vendors to tolerate manufacturing faults [29] has started to incur large overheads. To improve yields and provide protection against single-bit failures in the DRAM array, memory manufacturers have

started incorporating on-die single error correction (SEC) coding (on-die ECC) [24, 26]. The ECC encoding/decoding happens within the DRAM chip. Only the actual data, post correction, is sent out of the DRAM, making on-die ECC transparent to the outside world. The on-die ECC, along with within memory controller Single-Error Correction Double Error Detection (SECDED) ECC, is used to improve DRAM reliability. However, in the case of multi-bit errors (especially double-bit errors), using the two disjoint ECC schemes in the data pipeline results in increased chances of silent data corruption. This is because, for double-bit errors, the on-die SEC ECC has a $> 45\%$ of miscorrecting it to a triple-bit error. The in-controller SECDED then has a $\sim 55\%$ chance of further miscorrecting the triple-bit error, leading to silent data corruption. This was previously not an issue with only in-controller ECC as the SECDED decoder in the controller could always detect the double-bit error and flag a detectable-but-uncorrectable error (DUE). Thus, on-die SEC improves DRAM yield by silently correcting SBEs, but significantly reduces memory reliability and increases chances of SDC in the case of double-bit errors. This dissertation proposes a collaborative technique that utilizes memory data transfer protocol to not only eliminate double-bit error-induced SDCs but also correct almost all the double-bit errors that occur within the memory array.

1.4 Reliability and Performance Challenges of Non-Volatile Main Memory Systems

Though DRAM is still the main memory workhorse, several application contexts need different properties from the main memory (higher density, non-volatility, higher performance, etc). Hence, it is becoming increasingly important to consider alternative technologies that can potentially avoid the problems faced by DRAM and enable new opportunities.

Several emerging non-volatile memory (NVM) technologies are now being considered as potential replacements for or enhancements to DRAM. Most of these new non-volatile technologies (Phase Change Memory[PCM], STT-RAM, Resistive RAM[ReRAM], etc.) promise better scaling, higher density, and reduced cost-per-bit [30]. However, they come with their own set of challenges. The biggest problem that these emerging technologies face is the high stochastic bit error rate. In fact, the reliability challenges of NVMs can offset the density and energy advantages that they offer.

Increase in demand for memory capacity requires aggressive scaling of area-per-bit of storage. At higher density, these non-volatile emerging memory technologies tend to be more susceptible to stochastic bit errors [31]. Due to the random nature of the bit errors, these memory technologies require strong in-field error-correcting code (ECC) [32]. Also, for most of these emerging NVM technologies, some states show higher error rates than the rest. As a result, the conventional ECC schemes used in DRAM-based memory need to be extended for providing multi-bit asymmetric protection to maintain acceptable limits of yield and performance of systems. However, this adds significant overhead not just in terms of storage but also power and performance.

The other concern with NVMs is the higher read and write latency of most of these memory technologies. For example, Phase Change Memory (PCM) is considered one of the most promising scalable DRAM alternatives. However, the read latency of PCM is more than 4x [33] higher than that of DRAM. This leads to significant overall system performance degradation. One way to deal with this problem is to use a hybrid main memory system where DRAM is used as a last-level cache for the slower NVM main memory like PCM. However, there are two problems with this approach. Firstly, the DRAM cache is transparent to the OS and hence, reduces the overall main memory capacity by a non-negligible amount. Secondly, using the DRAM as a transparent cache makes the overall main memory system volatile. Hence, lightweight main memory-based persistence cannot be achieved even though a non-volatile main memory technology is being used. In most cases, optimizations to improve read/write latencies result in an increase in read/write error rates.

Overall we see that there are significant performance and reliability challenges in adopting these alternate emerging non-volatile memory technologies. This dissertation proposes solutions to (1) provide strong protection against the stochastic read/write errors while minimizing the overheads, (2) improve performance by trading off some of the density benefits as we need to pack in stronger ECC to tolerate the increased error rates.

1.5 Dissertation Outline

This dissertation, *Lightweight Opportunistic Memory Resilience*, presents a complementary suite of novel methods for tolerating faults and correcting errors in different levels of memory hierarchy. The

solutions proposed are tuned to particular memory characteristics; lightweight solutions for low-cost embedded memories and latency critical on-chip caches (Part 1) while stronger protection for off-chip main memory subsystems (Part 2). The high-level vision is depicted in Figure 1.2. Unlike conventional error correction schemes, we exploit data value behavior, underlying memory fault model, and memory architecture to opportunistically provide stronger protection against memory errors through novel code constructions and fault tolerance techniques at minimal overhead.

The organization and key contributions of this dissertation are as follows:

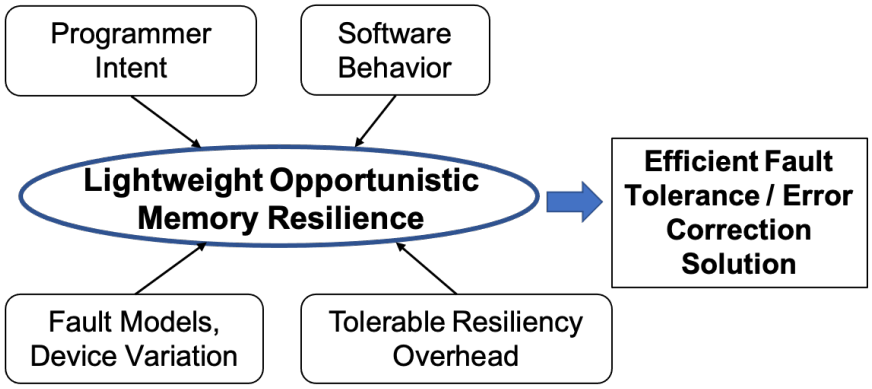


Figure 1.2: High level concept of Lightweight Opportunistic Memory Resilience

In Part 1, we provide a comprehensive set of solutions that cope with both hard and soft faults in lightweight on-chip memories at minimal cost.

- In Chapter 2, we develop FaultLink, a fault tolerance technique that tackles the problem of hard faults that appear at low voltages in software managed/scratchpad memories for embedded systems at the edge of the Internet-of-Things (IoT). It is a novel lazy link-time approach that extends the software construction toolchain with new fault-tolerance features for such memories. This approach builds an application binary that is custom-tailored for each individual chip based on the chip's memory fault map so that the faulty locations in the memory are never accessed when it is run at lower voltages.

- In Chapter 3, we extend FaultLink for approximation tolerant Deep Learning (DL) inference applications. We name the proposed technique SAME-Infer: providing the same inference accuracy in the presence of memory hard faults. Unlike FaultLink, SAME-Infer first deter-

mines the fault tolerance capabilities of each program section in the application. Based on this, it places the absolute critical program sections in non-faulty memory regions and the fault-tolerant sections in the partially faulty memory regions. This helps us to tolerant higher fault rates as compared to FaultLink and allows operation at even lower voltages.

- In Chapter 4, we propose Software-Defined Error Localization Code (SDELC), a hybrid hardware/software technique that allows the system to heuristically recover from unpredictable single-bit soft faults in instruction and data memories, which cannot be handled using FaultLink/SAME-Infer. SDELC first localizes the single-bit error to a particular chunk in the data and then heuristically recovers from it using side-channel information from the neighboring memory content.

- In Chapter 5, we propose Parity++, a novel unequal message protection scheme for embedded memories and last level caches that preferentially provides stronger error protection to certain "special messages". Like SDELC, Parity++ also helps to recover from single-bit flips that occur during runtime. Parity++ sits in between basic Single Error Detecting(SED) parity and a full single error correcting (SEC) Hamming code. The special messages get SEC protection while the non-special messages only have SED capability. This code requires only one extra parity bit over SED code and 4x lesser parity bits compared to a 64-bit SEC Hamming code. The critical portions of the application or the most frequently used instructions/data can be annotated as special messages to protect them from single-bit flips.

In Part 2, we provide solutions that improve the reliability of DRAM-based main memories and tackle challenges in adopting dense alternate non-volatile main memory technologies.

- In Chapter 6, we propose COMET: A Collaborative on-die and in-controller Memory ECC Technique that allows safer and stronger protection from DRAM errors. The proposed ECC construction techniques in COMET completely eliminate silent data corruption when a double-bit error occurs in the memory array. It also proposes a collaborative mechanism between the memory dies and the controller that allows correction of the majority (99.9997%) of the double-bit errors.

- In Chapter 7, we propose Compression with Multi-ECC (CME), a technique that opportunistically provides enhanced error resiliency for Magnetic Memories. We first compress each memory line using a compression scheme. Depending on how effective the compression was, the ECC scheme for that particular memory line is determined. The higher the compression, the stronger is the protection. This technique is designed such that it is compatible with most standard memory protocols.

- Chapter 8 presents an optimized PCM architecture that trades off some of the PCM density advantages over DRAM to bring down the PCM read latency. The proposed PCM-Duplicate mechanism provides a 2x capacity advantage (at the same cost) over DRAM while having almost DRAM-like read latency. We propose two possible memory system organizations to efficiently use PCM-Duplicate.

The different solutions discussed throughout this dissertation could be generalized to other systems and memory technologies. Overall the proposed suite of novel complementary techniques improves reliability across a broad domain of computing systems – from embedded edge devices in the IoT to high-performance computing systems – with significant energy efficiency and performance improvements that are critically needed in the nanoscale era. Instead of providing uniform protection against all errors, this dissertation exploits the behavior of the actual data it is protecting, understands the error patterns and fault models to identify the most dominant error mechanisms and then proposes a solution that is stronger and cheaper than the conventional techniques used in today's systems.

CHAPTER 2

FaultLink: Low Cost Fault Tolerance for IoT Devices

IoT devices need reliable hardware at low cost and low energy. One way to reduce energy consumption is by scaling down the on-chip memory supply voltage. However, this results in an exponential increase in the hard fault rate in these SRAM-based embedded scratchpad memories. It is challenging to efficiently cope with these faults. To address this problem, in this chapter, we propose *FaultLink*. FaultLink avoids hard faults found during testing by generating a custom-tailored application binary image for each individual chip. During software deployment-time, FaultLink optimally packs small sections of program code and data into fault-free segments of the memory address space and generates a custom linker script for a lazy-linking procedure. Our FaultLink approach improves min-VDD at which the scratchpad memories can be run by up to 440 mV as compared to the nominal VDD. This dramatically reduces energy consumption. Besides, FaultLink also helps to protect against aging-induced hard faults in memories and provides significant cost savings by removing the need for hardware replacement when memory faults occur post-deployment. It also helps to improve manufacturing yield and cost as memories with faults detected during testing no longer need to be discarded.

Collaborators:

- Dr. Mark Gottscho, UCLA/Google
- Dr. Clayton Schoeny, UCLA/Square
- Prof. Lara Dolecek, UCLA
- Prof. Puneet Gupta, UCLA

2.1 Introduction

For embedded systems at the edge of the Internet-of-Things (IoT), hardware design is driven by the need for the lowest possible cost and energy consumption, which are both strongly affected by on-chip memories [34]. Memories consume significant chip area and are particularly susceptible to parameter variations and defects resulting from the manufacturing process [35]. Meanwhile, much of an embedded system's energy is consumed by on-chip SRAM memory, particularly during sleep mode. The embedded systems community has thus increasingly turned to software-managed on-chip memories – also known as *scratchpad memories* (SPMs) [36] – due to their 40% lower energy as well as latency and area benefits compared to hardware-managed caches [37].

It is challenging to simultaneously achieve low energy, high reliability, and low cost for embedded memory. For example, an effective way to reduce on-chip SRAM power is to reduce the supply voltage [38]. However, this causes cell hard fault rates to rise exponentially [18] and increases susceptibility to radiation-induced soft faults, thus degrading yield at low voltage and increasing cost. Thus, designers traditionally include spare rows and columns in the memory arrays [19] to deal with manufacturing defects and employ large voltage guardbands [20] to ensure reliable operation. Unfortunately, large guardbands limit the energy proportionality of memory, thus reducing battery life for duty-cycled embedded systems [39], a critical consideration for the IoT. Although many low-voltage solutions have been proposed for caches, fewer have addressed this problem for scratchpads and embedded main memory.

Our goal in this work is to improve embedded software-managed memory reliability at minimal cost. We propose *FaultLink* that helps to guard applications against known hard faults. The key idea of this work is to automatically customize an application binary to individually accommodate each chip's unique hard fault map with no disruptions to source code. The contributions of this chapter are the following.

- We present FaultLink, a novel lazy link-time approach that extends the software construction toolchain with new fault-tolerance features for software-managed/scratchpad memories. FaultLink relies on hard fault maps for each software-controlled physical memory region that may be generated during manufacturing test or periodically during run-time using built-in-

self-test (BIST).

- We detail an algorithm for FaultLink that automatically produces custom hard fault-aware linker scripts for each individual chip. We first compile the embedded program using specific flags to carve up the typical monolithic sections, e.g., `.text`, `.data`, stack, heap, etc. FaultLink then attempts to optimally pack program sections into memory segments that correspond to contiguous regions of non-faulty addresses.

By experimenting with both real and simulated test chips, we find that with no hardware changes, FaultLink enables applications to run correctly on embedded memories using a min-VDD that can be lowered by up to 440 mV. *Our FaultLink approach could thus enable more reliable IoT devices while significantly reducing cost and run-time energy.*

This chapter is organized as follows. Background material that is necessary to understand our contributions is presented in Section 2.2. We then describe FaultLink in detail in Section 2.3 and evaluate in Section 2.4. We list some of the related works in Section 2.5. We then discuss other considerations and opportunities for future work in Section 2.6 and conclude the chapter in Section 2.7.

2.2 Background

We present the essential background on scratchpad memory, the nature of SRAM faults and sections and segments used by software construction linkers needed to understand our contributions.

2.2.1 Scratchpad Memories (SPMs)

Scratchpad memories (SPMs) are small on-chip memories that, like caches, can help speed up memory accesses that exhibit spatial and temporal locality. Unlike caches, which are hardware-managed and are thus transparent in the address space, data placement in scratchpads must be orchestrated by software. This requires additional effort from the application programmer, who must – with the help of tools like the compiler and linker – explicitly partition data into physical memory regions that are distinct in the address space. Despite the programming difficulty, SPMs

can be more efficient than caches. Banakar et al. showed that SPMs have on average 33% lower area requirements and can reduce energy by 40% compared to equivalently-sized caches [37]. In energy and cost-conscious embedded systems, SPMs are increasingly being used for this reason and because they provide more predictable performance. In this work, FaultLink is used to improve the reliability/min-VDD of SPMs/software-managed main memory.

2.2.2 Program Sections and Memory Segments

The Executable and Linkable Format (ELF) is ubiquitous on Unix-based systems for representing compiled object files, static and dynamic shared libraries, as well as program executable images in a portable manner [40]. ELF files contain a header that specifies the Instruction Set Architecture(ISA), Application Binary Interface(ABI), a list of program sections and memory segments, and various other metadata.

- A *section* is a contiguous chunk of bytes with an assigned name: sections can contain instructions, data, or even debug information. For instance, the well-known `.text` section typically contains all executable instructions in a program, while the `.data` section contains initialized global variables.

- A *segment* represents a contiguous region of the memory address space (i.e., ROM, instruction memory, data memory, etc.). When a final output binary is produced, the linker maps sections to segments. Each section may be mapped to at most one segment; each segment can contain one or more non-overlapping sections.

The toolchain generally takes a section-centric view of a program, while at run-time the segment-centric view represents the address space layout. Manipulating the mapping between program sections and segments is the core focus of FaultLink.

2.2.3 Tolerating SRAM Faults

There are several types of SRAM faults. In this chapter, we define *hard faults* to include all recurring and/or predictable failure modes that can be characterized via testing at fabrication time or in the

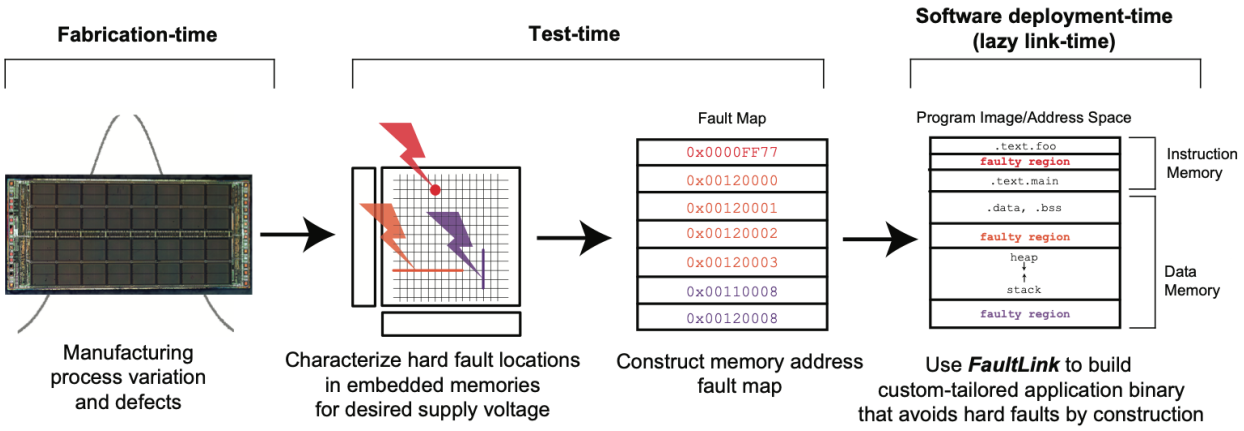


Figure 2.1: Our high-level approach to tolerating hard faults in on-chip scratchpad memories.

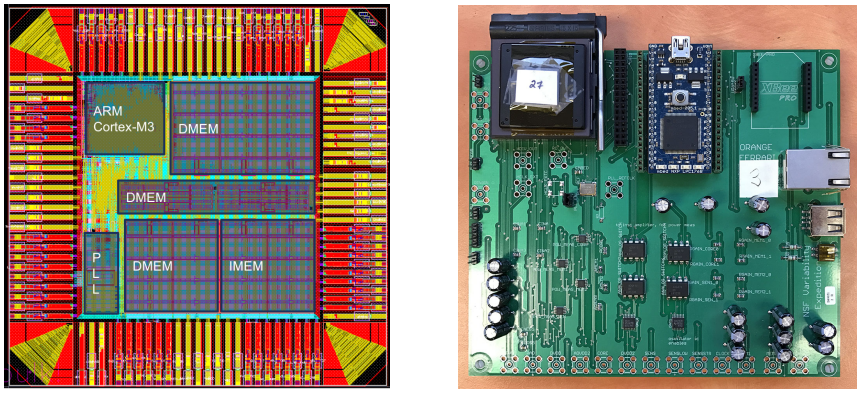
field. These include manufacturing defects, weak cells at low voltage, and in-field device/circuit aging and wearout mechanisms [41]. A common solution to hard faults is to characterize memory, generate a *fault map*, and then deploy it in a micro-architectural mechanism to hide the effects of hard faults.

We define *soft faults* to be unpredictable *single-event upsets* (SEUs) that do not generally reoccur at the same memory location and hence cannot be fault-mapped. The most well-known and common type of soft fault is the radiation-induced bit flip in memory [42]. Soft faults, if detected and corrected by an *error-correcting code* (ECC), are harmless to the system.

2.3 FaultLink

The high-level concept of FaultLink is illustrated in Figure 2.1. At fabrication time, process variation and defects may result in hard faults in embedded memories. During test-time, these are characterized and maintained in a per-chip fault map that is stored in a database for later. When the system developer later deploys the application software onto the devices, FaultLink is used to customize the binary for each individual chip in a way that avoids its unique hard fault locations.

Conventional software construction toolchains assume that there is a contiguous memory address space in which they can place program code and data. For embedded targets, the address space is often partitioned into a region for instructions and a region for data. On a chip containing hard faults,



(a) Chip floorplan **(b)** Board

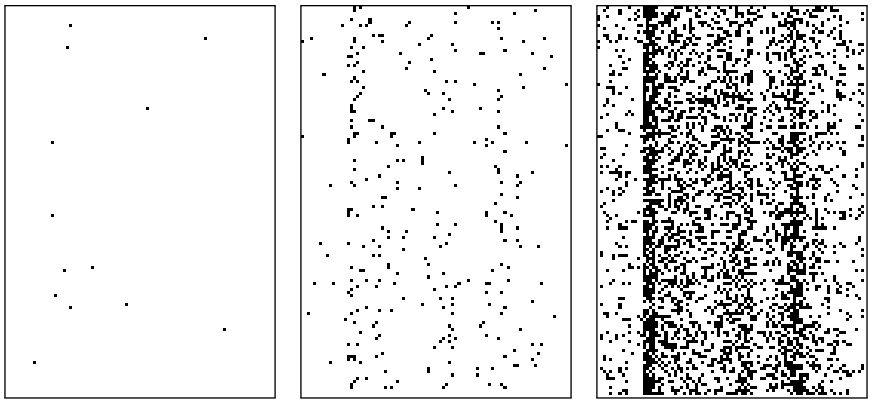
Figure 2.2: Test chip and board used to collect hard fault maps for FaultLink.

however, the specified address space can contain faulty locations. With a conventional compilation flow, a program could fetch, read, and/or write from these faulty locations, making the system unreliable.

FaultLink is a modification to the traditional embedded software toolchain to make it memory "fault-aware." At chip test-time, or periodically in the field using built-in-self-test (BIST), the software-managed memories are characterized to identify memory addresses that contain hard faults.

At software deployment time – i.e., when the application is actually programmed onto a particular device – FaultLink customizes the application binary image to work correctly on that particular chip given the fault map as an input. FaultLink does this by linking the program to guarantee that no hard-faulty address is ever read or written at runtime. Note that FaultLink is not heuristic and therefore does not induce errors.

We motivate FaultLink with fault mapping experiments on real test chips, describe the overall FaultLink toolchain flow, and present the details of the *Section-Packing* problem that FaultLink solves.



(a) 750 mV **(b)** 700 mV **(c)** 650 mV

Figure 2.3: Measured voltage-induced hard fault maps of the 176 KB data memory for one test chip. Black pixels represent faulty byte locations.

2.3.1 Test Chip Experiments

To motivate FaultLink, we characterized the voltage scaling-induced fault maps for eight microcontroller test chips. Each chip contains a single ARM Cortex-M3 core, 176 KB of on-chip data memory, 64 KB of instruction memory. They were fabricated in a 45nm SOI technology with dual-Vth libraries [43–45]; the chip floorplan and test board are shown in Figure 2.2. The locations of voltage-induced SRAM hard faults in the data memory for one chip are shown in Figure 2.3 as black dots. Its byte-level fault address map appears as follows:

```
0x200057D6
0x200086B4
...
0x2002142F
0x200247A9.
```

Without further action, this chip would be useless at low voltage for running embedded applications; either the min-VDD would be increased, compromising energy, or the chip would be discarded entirely. We now describe how the FaultLink toolchain leverages the fault map to produce workable programs in the presence of potentially many hard faults.

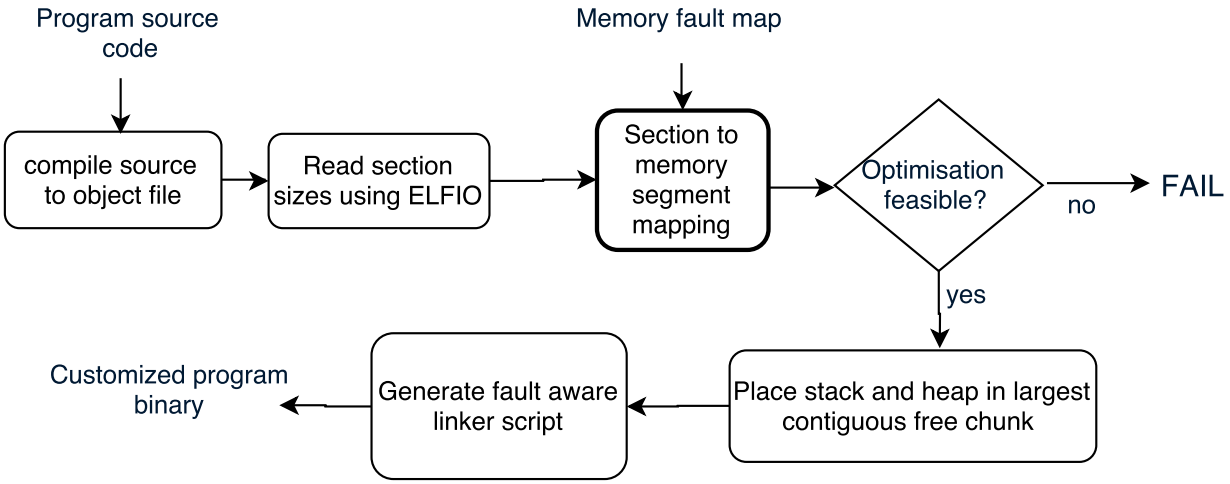


Figure 2.4: FaultLink procedure: given program source code and a memory fault map, produce a per-chip custom binary executable that will work in presence of known hard fault locations in the SPMs.

2.3.2 Toolchain

FaultLink utilizes the standard GNU tools for C/C++ without modification. The overall procedure is depicted in Figure 2.4. The programmer compiles code into object files but does not proceed to link them. The code must be compiled using GCC's `-ffunction-sections` and `-fdata-sections` flags, which instruct GCC to place each subroutine and global variable into their own named sections in the ELF object files. Our FaultLink tool then uses the ELFIO C++ library [46] to parse the object files and extract section names, sizes, etc. FaultLink then produces a customized binary for the given chip by solving the Section-Packing problem.

2.3.3 Fault-Aware Section-Packing

Section-Packing is a variant of the NP-complete Multiple Knapsacks problem. We formulate it as an optimization problem and derive an analytical approximation for the probability that a program's sections can be successfully packed into a memory containing hard faults.

2.3.3.1 Problem Formulation

Given a disjoint set of contiguous program sections M and a set of disjoint hard fault-free contiguous memory segments N, we wish to pack each program section into exactly one memory segment such that no sections overlap or are left unpacked. If we find a solution, we output the $M \rightarrow N$ mapping; otherwise, we cannot pack the sections (the program cannot accommodate that chip's fault map). An illustration of the Section-Packing problem is shown in Figure 2.5, with the program sections on the top and fault-free memory regions on the bottom.

Let m_i be the size of program section i in bytes and n_j be the size of memory segment j, y_j be 1 if segment j contains at least one section, otherwise let it be 0, and z_{ij} be 1 if section i is mapped to segment j, otherwise let it be 0. Then the optimization problem is formulated as an integer linear program (ILP) as follows:

$$\textbf{Minimize: } \sum_{j \in N} y_j$$

Subject to:

$$\sum_{i \in M} m_i \cdot z_{ij} \leq n_j \cdot y_j \; \forall j \in N$$

$$\sum_{j \in N} z_{ij} = 1 \; \forall i \in M$$

$$z_{ij} = 0 \text{ or } 1 \; \forall i \in M; j \in N$$

$$y_j = 0 \text{ or } 1 \; \forall j \in N.$$

We solve this ILP problem using CPLEX [47]. We use an objective that minimizes the number of packed segments because the solution naturally avoids memory regions that have higher fault densities. The constraints ensure that every program section gets packed in the non-faulty segments of the memory and the total size of all the sections packed in one non-faulty segment is no more than the size of that particular segment. (Note that other objectives will produce equally-valid section-packing solutions in terms of correctness; the important fault-avoidance constraints are fixed.) To pack any benchmark onto any fault map that we evaluated, CPLEX required no more than 14 seconds in the worst case; if a solution cannot be found or if there are few faults, typically

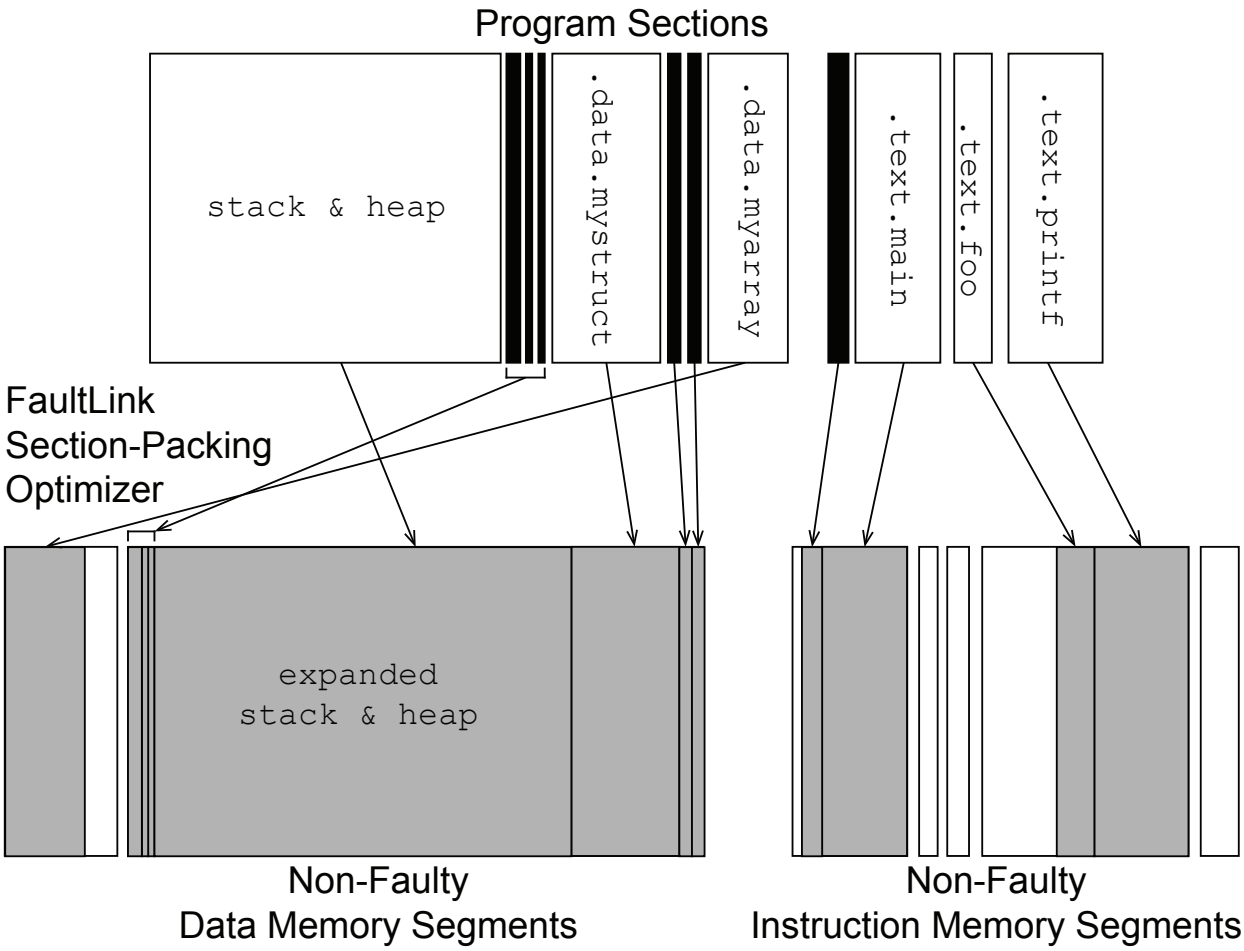


Figure 2.5: FaultLink attempts to pack contiguous program sections into contiguous disjoint segments of non-faulty memory. Gray memory segments are occupied by mapped sections, while white segment areas are free space. The depicted gaps between some of the gray/white boxes indicate faulty memory regions that are not available for section-packing.

FaultLink will complete much quicker. If a faster solution is needed, a greedy ILP relaxation can be used.

2.3.3.2 Analytical Section-Packing Estimation

We observe that the size of the maximum contiguous program section often comprises a significant portion of the overall program size, and that most FaultLink section-packing failures occur when the largest program section is larger than all non-faulty memory segments.

Therefore, we estimate the FaultLink success rate based on the probability distribution of the longest consecutive sequences of coin flips as provided by Schilling [48]. Let L_k be a random variable representing the length of the largest run of heads in k independent flips of a biased coin (with p as the probability of heads). The following equation is an approximation for the limiting behavior of L_k, i.e., the probability that longest run of heads is less than x and assuming $k(1-p) \gg 1$ [48]:

$$P(L_k < x) \approx e^{-p^{(x-\log_{p^{-1}}(k(1-p)))}}. \tag{2.1}$$

We apply Schilling's above formula to estimate the behavior of FaultLink. Let b be the i.i.d. bit-error-rate and s be the probability of no errors occurring in a 32-bit word, i.e., $s = (1-b)^{32}$. Let `size` be the memory size in bytes and $m_{\max}$ be the size in bytes of the largest contiguous program section. Using Equation 2.1, we plug in $p = s$, $k = \texttt{size}/4$, and $x = m_{\max}/4$. Then, we can approximate the probability of there *not* being a memory segment that is large enough to store the largest program section:

$$P\left(L_{\texttt{size}/4} < \frac{m_{\max}}{4}\right) \approx e^{-s^{\left(\frac{m_{\max}}{4} - \log_{s^{-1}}\left(\frac{\texttt{size}}{4}(1-s)\right)\right)}}. \tag{2.2}$$

This formula will be used in the evaluation to estimate FaultLink yield and min-VDD.

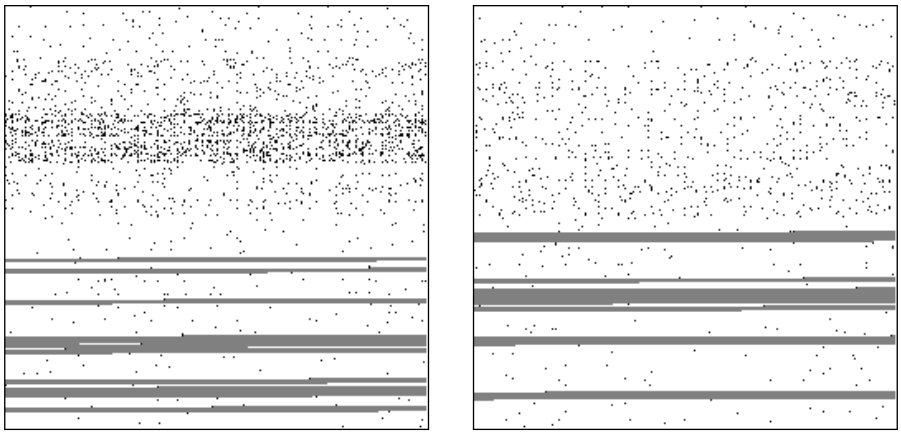
2.4 Evaluation

We evaluate FaultLink in terms of its ability to proactively avoid hard faults in software-managed memories.

We first demonstrate how applications can run on real test chips at low voltage with many hard faults in on-chip memory using FaultLink, and then evaluate the yield benefits at low voltage for a synthetic population of chips.

2.4.1 Voltage Reduction on Real Test Chips

We first apply FaultLink to a set of small embedded benchmarks that we build and run on eight of our microcontroller-class 45nm *"real test chips."* Each chip has 64 KB of instruction memory and



(a) Chip 1 **(b)** Chip 2

Figure 2.6: Result from applying FaultLink to the `sha` benchmark for two real test chips' 64 KB instruction memory at 650 mV.

176 KB of data memory. The five benchmarks are `blowfish` and `sha` from the mibench suite [49] as well as `dhrystone`, `matmulti` and `whetstone`. We characterized the hard voltage-induced fault maps of each test chip's SPMs in 50 mV increments from 1 V (nominal VDD) down to 600 mV using March-SS tests [50] and applied FaultLink to each benchmark for each chip individually at every voltage. Note that the standard C library provided with the ARM toolchain uses split function sections, i.e., it does not have a monolithic `.text` section. For each FaultLink-produced binary that could be successfully packed, we ran them to completion on the real test chips. The FaultLink binaries were also run to completion on a simulator to verify that no hard fault locations are ever accessed.

FaultLink-packed instruction SPM images of the `sha` benchmark for two chips are shown in Figure 2.6 with a runtime VDD of 650 mV. There were about 1000 hard-faulty byte locations in each SPM (shown as black dots). Gray regions represent `sha`'s program sections that were mapped into non-faulty segments (white areas).

We observe that FaultLink produced a unique binary for each chip. Unlike a conventional binary, the program code is not contiguous in either chip because the placements vary depending on the actual fault locations. In all eight test chips, we noticed that lower addresses in the first instruction SPM bank are much more likely to be faulty at low voltage, as seen in Figure 2.6. This could be

caused either by the design of the chip's power grid, which might have induced a voltage imbalance between the two banks, or by within-die/within-wafer process variation. Chip 1 (Figure 2.6a) also appears to have a cluster of weak rows in the first instruction bank. Because FaultLink chooses a solution with the sections packed into as few segments as possible, we find that the mapping for both chips prefers to use the second bank, which tends to have larger segments.

We achieved an average min-VDD of 700 mV for the real test chips. This is a reduction of 125 mV compared to the average non-faulty min-VDD of 825 mV, and 300 mV lower than the official technology specification of 1 V. FaultLink did not require more than 14 seconds on our machine to optimally section-pack any program for any chip at any voltage.

2.4.2 Yield at Min-VDD for Synthetic Test Chips

To better understand the min-VDD and yield benefits of FaultLink using a wider set of benchmarks and chip instances, we created a series of randomly-generated *synthetic fault maps*. For instruction and data SPM capacities of 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, and 4 MB, we synthesized 100 fault maps for each in 10 mV increments for a total of 600 *"synthetic test chips."* We used detailed Monte Carlo simulation of SRAM bit-cell noise margins in the corresponding 45 nm technology. Six more benchmarks were added from the AxBench approximate computing C/C++ suite [51] that are too big to fit on the real test chips: `blackscholes`, `fft`, `inversek2j`, `jmeint`, `jpeg`, and `sobel`. These AxBench benchmarks were compiled for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [52] and privileged specification v1.7 using the official tools. This is because unlike the standard C library for our ARM toolchain, the library included with the RISC-V toolchain has a monolithic `.text` section. This allows us to consider the impact of the library sections on min-VDD.

The expected min-VDD for 99% chip yield across 100 synthetic chip instances for seven memory capacities is shown in Figure 2.7. The vertical bars represent our analytical estimates calculated using Equation 2.2. The red line represents the empirical worst case out of 100 synthetic test chips, while the blue line is the lowest non-faulty voltage in the worst case of the 100 chips. Finally, the green line represents the nominal VDD of 1 V.

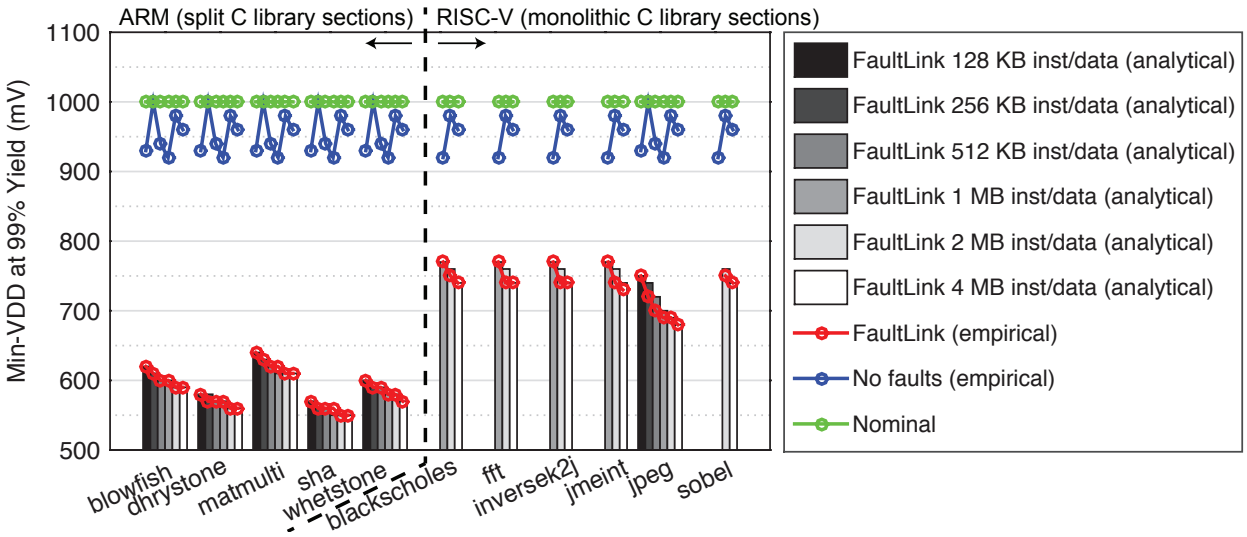


Figure 2.7: Achievable min-VDD for FaultLink at 99% yield. Bars represent the analytical lower bound from Equation 2.2 and circles represent our actual results using Monte Carlo simulation for 100 synthetic fault maps.

FaultLink reduces min-VDD for the synthetic test chips at 99% yield by up to 450 mV with respect to the nominal 1 V and between 370 mV and 430 mV with respect to the lowest non-faulty voltage. All but `jpeg` from the AxBench suite were too large to fit in the smaller SPM sizes (hence the "missing" bars and points). When the memory size is over-provisioned for the smaller programs, min-VDD decreases moderately because the segment size distribution does not have a strong dependence on the total memory size.

The voltage-scaling limits are nearly always determined by the length of the longest program section, which must be packed into a contiguous fault-free memory segment. This is strongly indicated by the close agreement between the empirical min-VDDs and the analytical estimates, the latter of which had assumed the longest program section is the cause of section-packing failure.

To examine this further, the program section size distribution for each benchmark is depicted in Figure 2.8. The name of the largest section is shown in the legend for each benchmark.

We observe all distributions have long tails, i.e., most sections are very small but there are a few sections that are much larger than the rest. We confirm that the largest section for each benchmark – labeled in the figure legend – is nearly always the cause of failure for the FaultLink

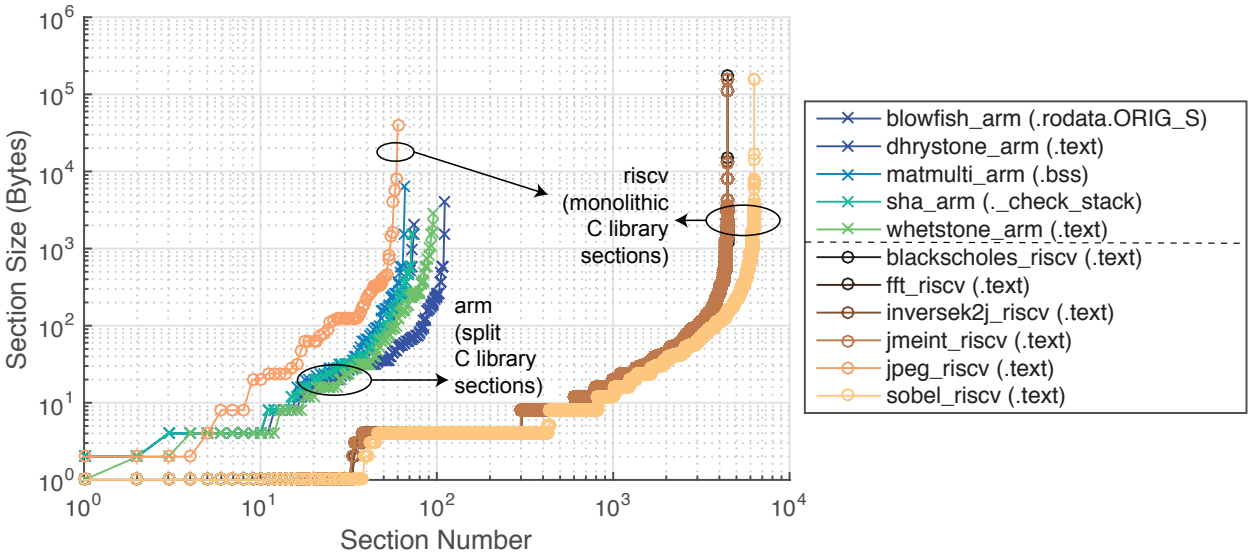


Figure 2.8: Distribution of program section sizes. Packing the largest section into a non-faulty contiguous memory segment is the most difficult constraint for FaultLink to satisfy and limits min-VDD.

section-packing algorithm at low voltage when many faults arise. Recall that the smaller ARM-compiled benchmarks have split C library function sections, while the AxBench suite that was compiled for RISC-V has a C library with a monolithic `.text` section; we observe that the latter RISC-V benchmarks have significantly longer section-size tails than the former benchmarks. This is why the AxBench suite does not achieve the lowest min-VDDs in Figure 2.7. Notice that program size is not a major factor: `jpeg` for RISC-V is similar in size to the ARM benchmarks, but it still does not match their min-VDDs. If the RISC-V standard library had used split function sections, the AxBench min-VDDs would be significantly lower. For instance, `jpeg` compiled on RISC-V achieves a min-VDD of 750mV for 128 KB memory, while on ARM (not depicted) it achieves a min-VDD of 660mV.

FaultLink does not require any hardware changes; thus, energy-efficiency (voltage reduction) and cost (yield at given VDD) for IoT devices can be considerably improved.

2.5 Related Work

We summarize and differentiate our work from related work on fault-tolerant caches and scratchpads.

2.5.1 Fault-Tolerant Caches

There is an abundance of prior work on fault-tolerant and/or low-voltage caches. Examples include PADded Cache [53], Gated-VDD [54], Process-Tolerant Cache [55], Variation-Aware Caches [56], Bit Fix/Word Disable [57], ZerehCache [58], Archipelago [59], FFT-Cache [60], VS-ECC [61], Correctable Parity Protected Cache (CPPC) [62], FLAIR [63], Macho [64], DPCS [65], DARCA [66], and others (see related surveys by Mittal [11, 67]). These fault-tolerant cache techniques tolerate hard faults/save energy by sacrificing capacity or remapping physical data locations. This affects the software-visible memory address space and hence they cannot be readily applied to SPMs.

Although they are cache-specific, some of the above techniques can be roughly compared with FaultLink in terms of min-VDD. For instance, DPCS [65] achieves a similar min-VDD to FaultLink of around 600 mV, while FLAIR [63] achieves a lower min-VDD (485 mV). We emphasize that the above techniques cannot be applied to SPMs and are therefore not a valid comparison.

2.5.2 Fault-Tolerant Scratchpads

The community has proposed various methods for tolerating variability and faults in SPMs that relate closely to this work. Traditional fault avoidance techniques like dynamic bit-steering [68] and strong ECC codes are too costly for small embedded memories. Meanwhile, spare rows and columns cannot scale to handle many faults that arise from deep voltage scaling.

E-RoC [69] is a SPM fault-tolerance scheme that aims to dynamically allocate scratchpad space to different applications on a multi-core embedded SoC using a virtual memory approach. However, it requires extensive hardware and run-time support. Several works [70–73] propose to use OS-based virtual memory to directly manage memory variations and/or hard faults, but they are not feasible in low-cost IoT devices that lack support for virtual memory; nor do they guarantee avoidance of

known hard faults at software deployment time. Others have proposed to add small fault-tolerant buffers that assist SPM checkpoint/restore [74], re-compute corrupted data upon detection [75], build radiation-tolerant SPMs using hybrid non-volatile memory [76] and duplicate data storage to guard against soft errors [77]; these are each orthogonal to this work.

2.6 Discussion

We highlight several considerations and beneficial use cases for FaultLink and outline directions for future work.

2.6.1 Memory Reliability Binning

FaultLink could bring significant cost savings to both IoT manufacturers and IoT application developers throughout the lifetime of the devices. Manufacturers could sell chips with hard defects in their on-chip memories to customers instead of completely discarding them, which increases yield. Customers could run their applications on commodity devices with or without hard defects at lower-than-advertised supply voltages to achieve energy savings. Fault maps for each chip at typical min-VDDs are small (bytes to KBs) and could be stored in a cloud database or using on-board flash. Several previous works have proposed heterogeneous reliability for approximate applications to reduce cost [78–81].

2.6.2 Coping with Aging and Wearout using FaultLink

Because IoT devices may have long lifetimes, aging becomes a concern for the reliability of the device. Although explicit memory wearout patterns cannot be predicted in advance, fault maps could be periodically sampled using BIST and uploaded to the cloud. Because IoT devices by definition already require network connectivity for their basic functionality and to support remote software updates and patching of security vulnerabilities, it is not disruptive to add remote FaultLink support to adapt to aging patterns. Because running FaultLink remotely takes just a few seconds, customers would not be affected any worse than the downtime already imposed by routine software

updates and the impact on battery life would be minimum.

2.6.3 Directions for Future Work

The FaultLink approach can be further improved upon. One could extend FaultLink to accommodate hard faults within packed sections to reduce min-VDD and increase reliability. For FaultLink with instruction memory, one approach could be to insert unconditional jump instructions to split up basic blocks, similar to a recent cache-based approach [82]. For FaultLink with data memory, one could use smaller split stacks [83] and design a fault-aware `malloc()`.

2.7 Conclusion

We proposed FaultLink to improve memory resiliency for IoT devices in the presence of hard faults. FaultLink tailors a given program binary to each individual embedded memory chip on which it is deployed. This improves both device yield by avoiding manufacturing defects and saves runtime energy by accounting for variation-induced parametric failures at low supply voltage. FaultLink does not require any hardware changes; thus, energy-efficiency (voltage reduction) and cost (yield at given VDD) for IoT devices can be considerably improved.

CHAPTER 3

SAME-Infer: Software Assisted Memory Resilience for Efficient Inference at the Edge

In the previous chapter, we presented FaultLink that helps to tolerate hard faults at low voltages in SRAM-based scratchpad memories typically found in edge devices. Deep learning neural network applications constitute a significant fraction of the workloads that are run today on these low cost embedded devices. Despite the inherent resilience of most of these deep learning applications, inference accuracy degrades significantly at high fault rates. In this chapter, we propose SAME-Infer, a **s**oftware **a**ssisted **m**emory resilience technique for **e**fficient **infer**ence at the edge. It is a fault-aware linking methodology for software-managed embedded memories to efficiently map the critical code/layers onto the non-faulty segments of the memory and the non-critical fault tolerant sections in the faulty or error-prone memory segments. This is done in a way such that memory hard faults can be tolerated and voltage be lowered without degrading the accuracy (**SAME infer**ence accuracy at lower voltage/higher error rate). Unlike FaultLink, SAME-Infer exploits the approximation-tolerant nature of DNNs and efficiently uses faulty regions of the memory as well, thereby delivering much higher energy reduction/fault tolerance (more than 100mV min-VDD gain) as compared to FaultLink. Our evaluation on 10 real microcontroller class chips shows that more than 175mV reduction in voltage can be achieved without any loss in accuracy for a variety of neural networks. SAME-Infer can also be considered as an efficient fault tolerance/in-field repair technique as it tolerates on average 25x (upto 350x) increase in bit error rate with minimal impact on inference accuracy.

Collaborators:

- Prof. Puneet Gupta, UCLA

3.1 Introduction

The demand for deploying deep learning neural network (DNN) algorithms in edge and mobile devices is increasing. These applications are extremely compute intensive. Since the edge devices are often energy constrained, it is critical to enhance the energy efficiency of DNN inference on such devices. Further, these edge devices are deployed in increasingly harsh environments resulting in worsened hardware failure rates [84] but still require continued reliable operation. To make matters worse, typical fault tolerance techniques (sparing, system-level fault tolerance, error correcting codes) are usually unaffordable due to cost or energy reasons in these contexts. Furthermore, many of the faults are permanent or semi-permanent and possibly wearout related. As a result, in-field repair/replace, though needed, is very difficult in many environments.

As mentioned in Chapter 2, cost and energy consumption in embedded systems at the edge of the Internet-of-Things (IoT) are both strongly affected by on-chip memories [34]. To make these memories efficient, these devices typically use software-managed on-chip memories – also known as *scratchpad memories* (SPMs) [36] – due to their 40% lower energy as well as latency and area benefits compared to hardware-managed caches [37]. One way to further reduce the energy consumption of these on-chip memories is by reducing the supply voltage. However, doing so leads to an exponential rise in the memory cell hard fault rate. Running applications on a voltage scaled device with faulty memory leads to erroneous behaviour of the application. To enable voltage scaling, we proposed FaultLink in Chapter 2.

On the other hand, DNN algorithms are known to be approximation friendly and fault resilient [85]. Previous works have shown that if a few elements in the weight matrix or inputs are erroneous, the final inference accuracy remains unchanged. These errors often do not get propagated to the output or the perturbation these errors cause is negligible enough such that the final classification is not affected. A recent body of work has focused on exploiting this characteristic of Deep Learning (DL) networks by reducing precision of computation through quantization as it does not affect accuracy but significantly reduces storage requirement [86, 87]. Therefore, DNN algorithms have an inherent capability of masking errors due to memory faults if the errors occur in non-critical locations.

In this chapter we extend FaultLink and propose SAME-Infer, a lazy link-time fault-aware memory mapping approach for Deep Learning networks. SAME-Infer extends the software construction toolchain (compiler and linker), for software managed memories, to intelligently map the critical regions of the network in non-faulty memory segments and non-critical fault tolerant regions in the faulty segments of the memory. Contributions of this chapter include:

- We study the impact of memory faults on inference accuracy by approximating the error tolerance of each layer's weight and activation values. We extend this methodology to approximate the error tolerance of every weight kernel (per layer, per filter). Further, we use this methodology to bin the data sections of a DNN program based on varying degrees of criticality.
- We develop SAME-Infer approach to relink compiled program based on memory fault map and the criticality measure of the data partitions.
- We develop analytical models for predicting probability of successful relinking given the program and the expected bit error rate (BER).
- We evaluate the SAME-Infer approach on ten real microcontroller class chips running 8-bit and binarized CNNs and MLPs on well-known MNIST, Google Speech command and CIFAR10 datasets. We measure the achieved energy reduction and fault tolerance. The results show opportunities of memory voltage reduction by up to 175mV and 350x improvement in BER tolerance.
- We also show approaches which try to make trained networks generally robust (e.g., [88]) do not work as well as SAME-Infer.

Thus, SAME-Infer provides a methodology to tolerate (and repair) increased hard fault rate in systems with scratchpad based memories while maintaining the **same infer**ence accuracy for Deep-Learning applications. The increased BER tolerance not only helps to lower supply voltage and save energy, it also helps to tolerate aging induced faults. When in-field memory faults happen, it provides a simple software patch solution alternative to difficult in-field repair and/or expensive

hardware replacement solutions. SAME-Infer also provides significant cost saving since device manufacturers can now save chips with fault-prone or faulty cells rather than discarding them.

3.2 Background

In this section we briefly present the essential background beyond FaultLink that is required to understand our contributions in SAME-Infer.

3.2.1 SRAM Faults

SRAM faults can be primarily characterized as either soft or hard faults. Soft faults manifest at runtime due to radiation induced high energy particle strikes, value disturbance due to cell leakage etc. Error Correcting Codes (ECC) is a typical approach to deal with soft faults. *Hard faults*, on the other hand, include all recurring and/or predictable failure modes that can be characterized via testing at fabrication time or in the field. These include: manufacturing defects, weak cells at low voltage, and in-field device/circuit aging and wearout mechanisms [41]. Using ECC for low voltage induced hard faults will require a very strong protection scheme with high overheads, making them impractical in the context of low cost platforms. A common solution to hard faults is to characterize the memory, generate a *fault map*, and then deploy it in a system-level mechanism (e.g., page retirement in systems which support virtual memory) to hide the effects of hard faults. However, most IoT devices are bare metal and do not have support for operating system and virtual memory framework due to limited memory capacity and energy budget. Simple solutions used traditionally by designers to increase reliability are including spare rows and columns [19] in the memory arrays and employing large voltage guardbands [20]. Unfortunately, as the voltage is scaled and the fault rate rises exponentially, sparing soon becomes insufficient. Also, large voltage guardbands limit the energy proportionality of memory, thus reducing battery life for duty-cycled embedded systems [39], a critical consideration for the IoT. Although there are several past works that propose approaches for reliable operation in low voltage SRAM caches [57, 61], they cannot be used in the context of scratchpads and embedded main memory and often incur impractical overheads for low cost devices.

3.2.2 Fault Resilient DL networks

Most deep learning neural networks are known to be moderately fault resilient because of the abundant redundancy present in these networks. However, the resilience of a DL network depends on the type of data (such as inputs vs. weights), data values, data-types (32-bit float vs. 8-bit integer), layer type/position in the network (such as input layer vs. hidden layer, convolution layer vs. fully connected layer), etc [85]. Inherent resilience and redundancy in neural networks have also been leveraged to reduce precision of computation [86, 87] or for compression [89]. A recent work [88] focuses on exploiting the fault resilient characteristic of these networks for performance improvements and energy savings in DRAM. However, it requires the network to be retrained on the target approximate DRAM system. Such an approach is often infeasible for low cost, compute/memory starved edge devices. They also proposed offloading the retraining on a separate system using a random bit error rate (BER). However, hard faults in memory (especially at lower voltage in SRAMs) are often correlated and hence, modeling the bit errors as a uniform random distribution is not very accurate (as we show later in this chapter).

3.3 SAME-Infer Methodology

Software construction toolchains, by default, consider the memory address space to be contiguous and place the program code and data accordingly. However, with hard faults in the memory, the contiguous placement of data and code can result in the intersection of program sections with faults, making the system and program execution unreliable. SAME-Infer extends the default toolchain so that it is fault-aware and has the ability to incorporate the fault map while placing instructions and data into the memory with faults. Thus, at software deployment time, SAME-Infer, with the help of the modified toolchain, prepares a customized binary for each chip such that all critical sections of the program are placed in non-faulty contiguous memory segments and the non-critical sections in memory segments containing faults.

Figure 3.1 shows the complete SAME-Infer flow. In order to be able to place program sections into different memory segments efficiently, the monolithic program sections such as *.text*

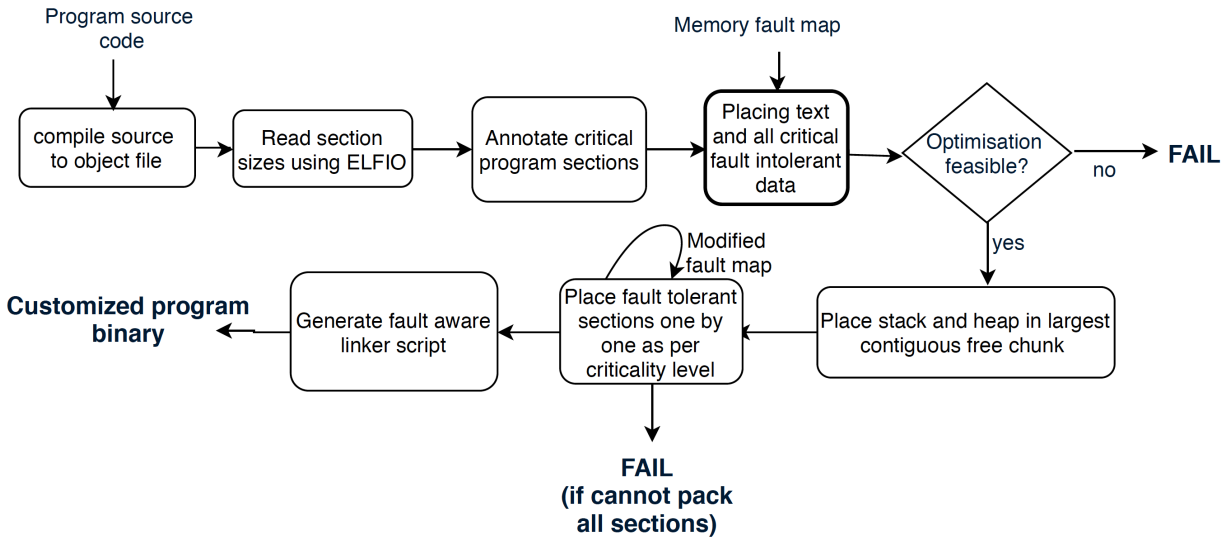


Figure 3.1: SAME-Infer procedure: given source code of a DL network and a memory fault map, produce a per-chip custom binary executable that will work in presence of known hard fault locations in the SPMs.

and *.data* need to be split up on a per-function and/or a per-variable basis so that each smaller section can be mapped to a particular memory segment by the fault-aware linker. In order to do that, the programmer initially needs to compile the code using special compiler flags for GCC (`-ffunction-sections` and `-fdata-sections`) so that the compiler can place each subroutine and global variable into their own named sections in the ELF object file. After compiling the code the programmer does not link the object files. In the next step, the object file is parsed using standard ELFIO C++ library [46] to record the name of each program section and its size.

Once the program sections and their sizes are recorded, the sections are then annotated with their criticality level (i.e., how many least significant bit (LSB) errors can be tolerated). The section packing algorithm (described later in Section 3.3.2) then iteratively maps sections to segments starting with most critical sections first (each criticality level gets its own memory fault map). If the tool provides a feasible solution, a part of the customized linker is generated based on that solution for the critical program sections. The stack and heap are placed in the largest remaining non-faulty contiguous memory segment.

3.3.1 Fault Impact Analysis

The non-critical sections of the network (in this chapter we considered weights and activation data as non-critical sections) are fault resilient, but upto a certain degree. Naively placing these non-critical sections in the memory can dramatically impact accuracy. As a result it is important to measure the effect of bit errors in each of these non-critical weights and activation data on overall accuracy. In the ideal scenario, it is required to search for the highest tolerable bit error rate (maximum error) of each weight and activation data that would still yield an acceptable inference accuracy. However, this search space is exponential, given the total number of weights and activations in a reasonably sized DNN. In order to keep the granularity of sections reasonable, we did layer-wise sensitivity analysis of the weights and activations to understand the impact of each layer's weights and outputs on the overall accuracy.

The approach to calculate inference error sensitivity to bit errors leverages the quantization approach proposed in [90]. We essentially approximate bit errors in the k least significant bits by reducing the precision by k bits. For example, in our 8-bit 2 layer MLP network (weights and activations have 8-bit fixed point precision), the most sensitive weights (layer-1 weights) can be quantized down to 5 bits without loss in accuracy. We interpret this as layer-1 weights can have upto three bit errors in the least significant 3 bits. The fault map for layer-1 weights, thus, will contain all memory addresses where there is an error in the most significant 5 bits.

3.3.2 Packing Critical and Non-Critical Sections

We solve the section to segment mapping problem iteratively. In a 8-bit network, we allow 5 criticality levels: from 0 to 4 LSB errors. For every criticality level, the corresponding program sections are identified from the sensitivity analyses above. For every chip, the fault map is obtained from a software/BIST memory testing routine. The fault map is different for every criticality level depending on how many LSB errors can be tolerated. Section packing is then solved in criticality order (most critical first) i.e., the section packing algorithm is run 5 times (in the 8 bit case) sequentially each time with a different fault map and available memory segments. A sample packing solution is shown in Figure 3.2.

The section-packing problem itself, is a variant of the Multiple Knapsacks problem [91] which we solve using an ILP[1] with multiple criticality levels. The objective is to minimize the number of packed memory segments so that there are large enough chunks of memory in between the packed segments to accommodate the program sections for the remaining criticality levels. This objective also helps to naturally provide a solution that avoids memory regions with higher fault densities. The placement algorithm ensures that no weights or activation data of a particular layer intersects with faulty bytes that have errors in the intolerable more significant bit positions. The algorithm also ensures that these non-critical sections do not overlap with the already placed critical program sections. Once this mapping is done, the linker generates the customized binary ready to be deployed on that particular chip.

3.3.3 Breaking up monolithic weight sections into smaller kernels

We observe that SAME-Infer fails when the packing of the largest section fails. A lot of times the largest section turns out to be a data section corresponding to a particular layer's weight. One way to relieve this would be to do a one-time simple modification of the source code where the weight data sections are broken down into smaller sections. A simple way to do this would be to break up the convolution layer weights on a per-filter basis. This splitting induces no code space overhead as the same functions can be used, the only additional step would be to concatenate the final output. The layer-wise sensitivity analysis of the weights in [90] can be modified to compute weight quantization noise gain on a per layer per filter basis as shown below:

$$E_{W,l,k} = \mathbb{E}\left[\sum_{\substack{i=1 \\ i \neq Y_{fl}}}^{M} \frac{\sum_{h \in W_{l,k}} \left| \frac{\partial (Z_i - Z_{Y_{fl}})}{\partial w_h} \right|^2}{24 \left| Z_i - Z_{Y_{fl}} \right|^2}\right] \tag{3.1}$$

Here, M is the number of classes, $\{Z_i\}_{i=1}^{M}$ are the soft outputs, $Z_{Y_{fl}}$ is the floating point output, and $\{\{W_{l,k}\}_{l=1}^{L}\}_{k=1}^{N_C}$ are the per layer (L is the total number of layers), per filter (N_c is the number of filters in layer l) weights.

[1]Number of sections/segments is small enough that the ILP runtime was always less than 20 seconds in our experiments.

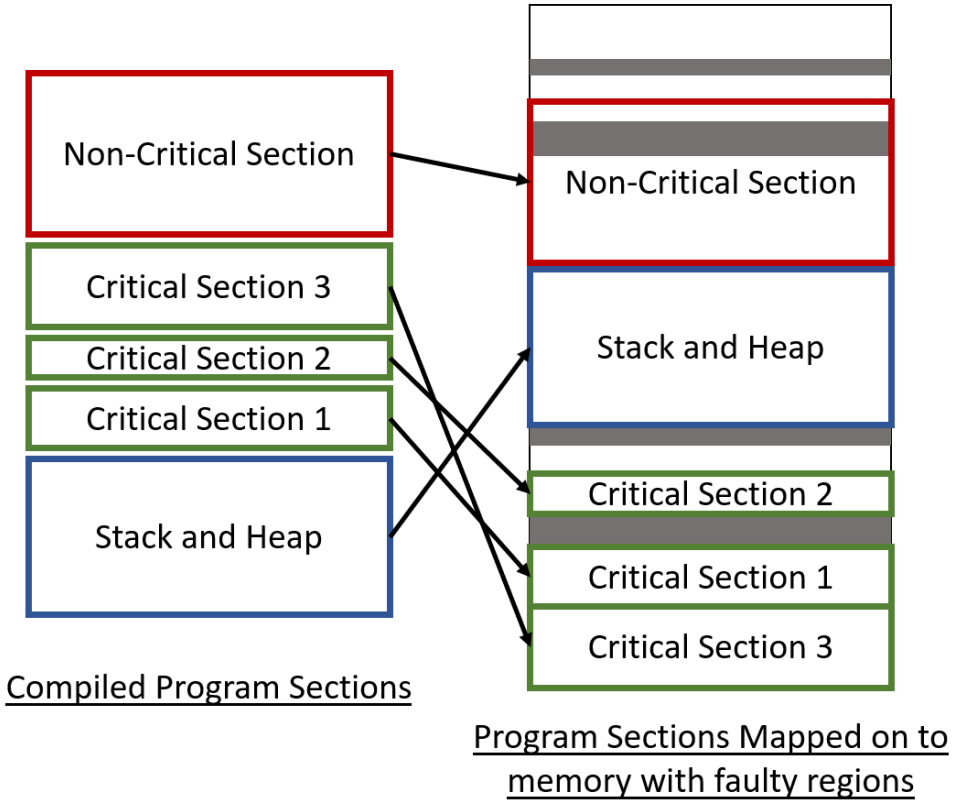


Figure 3.2: A sample section packing solution provided by SAME-Infer. The critical sections are placed in fault free memory segments while the non-critical sections intersect with faults (grey regions represent fault locations). The stack and heap is placed in the largest non faulty contiguous memory segment remaining after placing the critical sections.

We computed the per filter quantization noise gain for a large CNN (with 6 convolution layers and 3 fully connected layers). The network architecture is $32C3 - 32C3 - MP2 - 64C3 - 64C3 - MP2 - 128C3 - 128C3 - 256FC - 256FC - 10$ using CIFAR-10 dataset [92]. The layer 1 and 2 filter wise quantization noise gain results obtained using Equation 3.1 are shown in Figure 3.3. From the results it can be seen that the sensitivity of the weights across kernels varies significantly. Since the precision assignment matches the quantization noise gain profile on a logarithmic scale, using the same number of least significant bits for error tolerance for every weight in a layer might lead to under utilization of available network redundancy for energy efficiency. In Section 3.5.2, we analyze the min-VDD benefits of weight splitting.

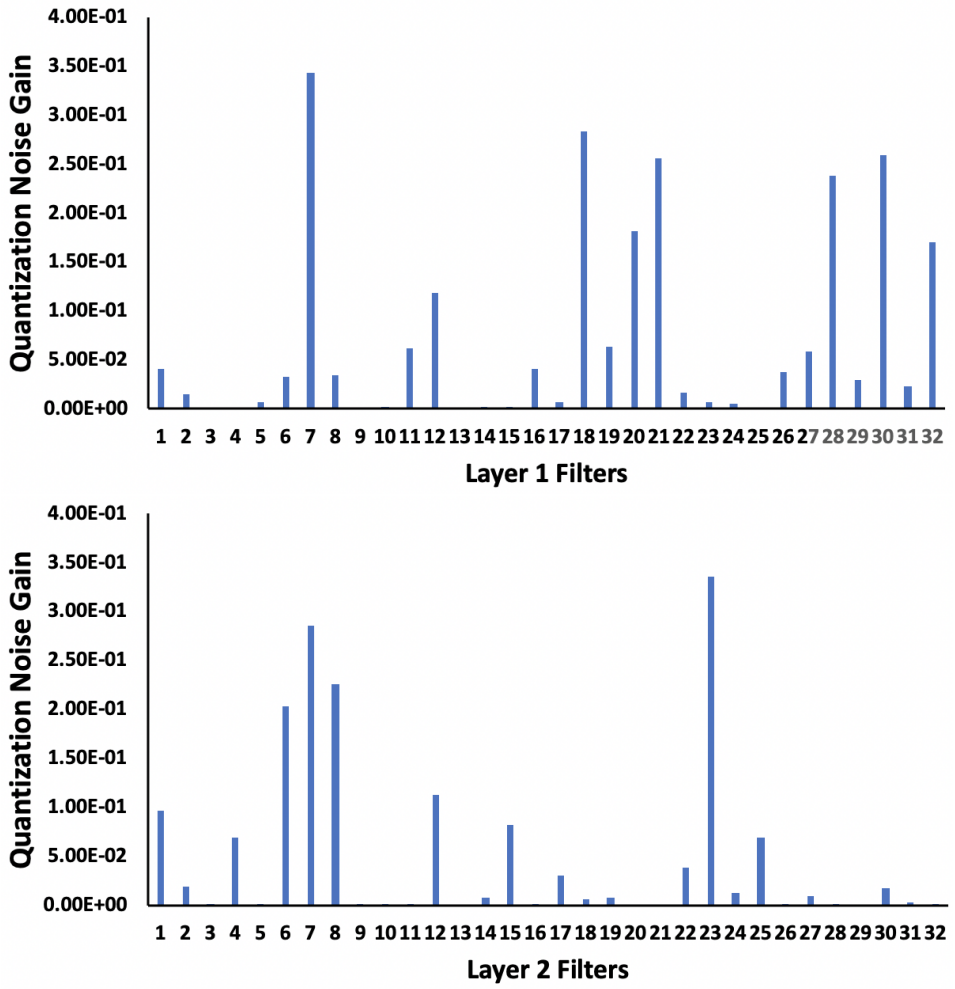


Figure 3.3: Weight quantization noise gain per filter - layers 1 and 2 of a nine layer CNN.

Finding the best split size: The smallest granularity at which the non critical sections need to be split in order to be able to run at a given voltage can be determined analytically, as provided in the next subsection. Note that going to finer-grained splitting than what is ”natural” for neural networks (layer/filter), would require fairly intrusive code changes which we want to avoid. Furthermore, making the split granularity much smaller than section sizes in the code part of memory (dictated by the code and not weights/activations) is not useful as code memory will limit the voltage scalability in that case(Figure 3.6). In this work, we limit splitting to a filter granularity.

Performance and Code Size Overheads We evaluated the performance and code size overheads of splitting up data and code sections on a per-variable/per-function basis and placing them in

non-contiguous memory segments. The performance impact is almost negligible ($\sim 0.1\%$). This is because of static allocation of program sections. Also, since stack and heap are not split, there is no additional performance overhead due to increased pointer chasing. There is no impact on code size since the source code remains unchanged and hence, the size of the final executable that is loaded into the memory also remains unchanged. Even splitting to a filter granularity resulted in minimal code changes and negligible ($\sim 1\%$) code size overhead.

3.3.4 Analytical Critical and Non-critical Section Packing Estimation

As determined by our evaluations in FaultLink, the section packing mostly fails when the largest program section is larger than the largest non-faulty contiguous memory segment. We extend the packing failure probability model developed in FaultLink to account for multiple criticality levels.

For that we need to iteratively perform the estimation for each section since the fault tolerance capability of different sections is different. We use the same Equation 2.2 that we developed in FaultLink for approximating the packing failure probability. However, `size` and s varies between program sections. For example, if each weight of a particular layer is n bits and it can tolerate errors in upto $k-bits$ from the LSB, then

$$s = \left((1-b)^{(n-k)}\right)^{32/n} \tag{3.2}$$

This value of s is then substituted in Equation 2.2. We start with the most critical weights and activations. For that layer, the size of the memory is taken as $size - size_{crit}$ bytes where $size_{crit}$ is the sum of the sizes of all program sections more critical than the current one being packed. After computing each layer, the size of the memory is reduced by the total size of weights and outputs of that layer since we will try to pack the next layer (in terms of criticality) after the previous layer has already been placed in the memory.

This analytical approach, combined with the sensitivity analysis results, can be used to estimate the achievable accuracy and packing yield at a particular VDD or BER and hence predict before deployment, fault tolerance and/or energy benefits of SAME-Infer for a specific hardware platform and neural network.

3.4 Experimental Setup

We evaluate SAME-Infer on ten micro-controller class test chips, same as FaultLink. Each chip contains a single ARM Cortex-M3 core, 176 KB of on-chip data memory, 64 KB of instruction memory. They were fabricated in a 45nm SOI technology with dual-Vth libraries the chip floorplan and test board are shown in Figure 2.2. We characterized the voltage scaling-induced bitwise fault maps for these ten chips using detailed March-SS tests [50].

For most of our experiments, we used four 8-bit networks as given in Table 3.1. The first network is a minimally sized perceptron (MLP) with one hidden layer and is tested using MNIST dataset [93]. The second is a convolutional neural network (CNN) with two convolution layers and one fully-connected layer and is tested using Google Speech Command dataset [94]. The third and the fourth networks are bigger convolutional neural networks with three and six convolution layers and one and three fully-connected layers, respectively. Both these networks are tested using CIFAR-10 dataset [92]. All the layer weights and activation data of all networks are quantized to 8-bit precision. The first three networks are implemented using the ARM CMSIS-NN [95] library, version 5.6.0, optimized for Cortex-M processors and run on the test chip. The fourth network was too big to fit on our test chip. Hence, a fault injection framework was developed using PyTorch 0.4.1. In Table 3.1, 32CONV5-MP2 means 32 5x5 filters and 2x2 max pooling layer while 12FC/10FC means fully connected layer with 12/10 output neurons. The networks were trained using PyTorch 0.4.1.

For all these networks we considered all weights and activation data to be non-critical (i.e., where sensitivity to errors would be calculated to assign them a criticality level) and all other parameters and instructions to be critical.

The toolchain, by default, packs the entire program code sequentially in the memory with no notion of faults. The supply voltage of each chip is reduced from the nominal 1V to 600mV in step size of 25mV. As the voltage is reduced, the hard fault rate in the memory increases. For each step, the binary is loaded and ran till completion to note the accuracy drop with decrease in voltage.

We then performed SAME-Infer for all 10 chips at every 25mV voltage step from 1V to 600mV and ran the customized binary on the chips to measure the final accuracy for all the voltage levels

Table 3.1: DL networks used in our experiments

Model (Precision)	Architecture	Dataset
MLP (8-bit, 1-bit)	784-128-10	MNIST
CNN-1 (8-bit)	32CONV5-MP2 32CONV5-MP2 12FC	SPEECH
CNN-2 (8-bit)	32CONV5-MP2 32CONV5-MP2 64CONV5-MP2 10FC	CIFAR-10
CNN-3 (8-bit)	32CONV3-32CONV3 MP2-64CONV3-64CONV3 MP2-128CONV3-128CONV3 256FC-256FC-10FC	CIFAR-10

which had feasible packing solutions. The layer-wise precision results were obtained using a Theano [96] based framework. We used CPLEX [47] to solve the ILP packing problem in our experiments. All code is packed in the instruction memory while all data is packed in the data memory. If the section packing (solved iteratively for the multiple criticality levels) failed, we considered SAME-Infer not to have a feasible solution for that network on that particular chip at voltage equal and/or less than the current one. Since CNN-3 was too large for the test chip, the accuracy results were obtained using our PyTorch based fault injection framework. We created a series of randomly-generated *synthetic fault maps* for memory of size 1 MB. We synthesized 10 fault maps in 10 mV increments for a total of 10 *"synthetic test chips."* using detailed Monte Carlo simulation of SRAM bit-cell noise margins in the corresponding 45 nm technology. The fault maps were fed into our PyTorch framework and inference with faulty weights and activations were run to get the final accuracy.

We extended our analysis to binarized networks which can often be used for very resource-

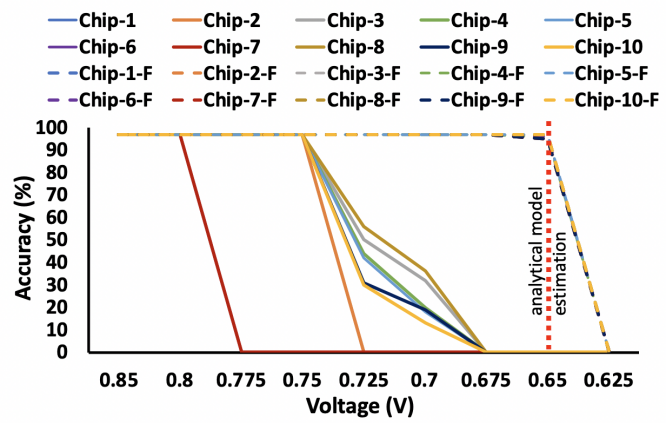
constrained embedded devices. As an example, we analyze binarized versions (dense and one with 85% sparsity) of the two layer MLP network. Since binarized networks, especially the sparse one, have the least amount of redundancy in them and are much smaller in size as compared to the 8-bit networks, they will be less resilient to faults, while at the same time, would be easier to fit in the memory. As binarized networks are already at the lowest precision level, error sensitivity is not calculated for these networks. Instead, while packing the weights, we try to maximize the intersection of 1's with stuck at one faults and the 0 values with stuck at zero faults.

3.5 Results

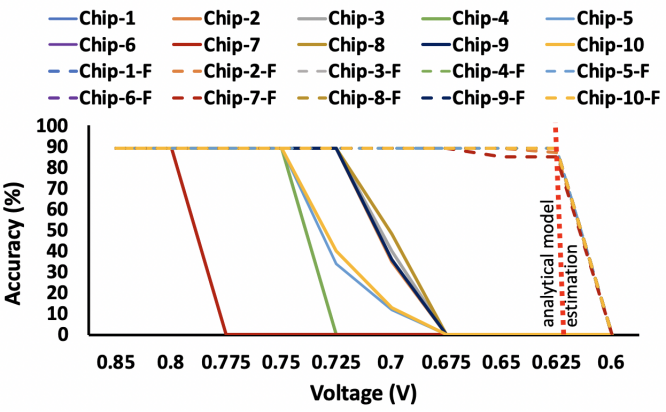
As the supply voltage of each of the ten chips is reduced from the nominal 1V to 600mV in step size of 25mV, the hard fault rate starts increasing. The first faults start appearing around 800-850mV and the fault rate increases exponentially beyond 750mV.

3.5.1 Reduction in voltage with SAME-Infer

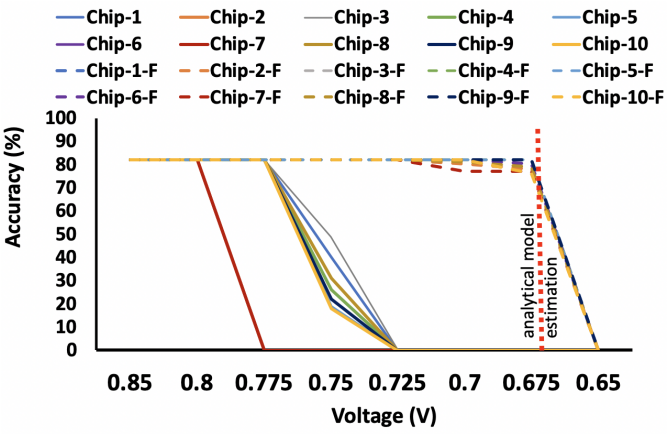
Using the default toolchain placement (without SAME-Infer), all three networks are ran for each voltage step on all ten chips and the results are shown by the solid lines in Figures 3.4a, 3.4b and 3.4c for MLP, CNN-1 and CNN-2, respectively. For the 2-layer MLP and the 2-layer CNN (CNN-1), network accuracy remains almost unchanged till above 750mV for 8 out of 10 chips and drops drastically at 725mV. This is because from 750mV to 725mV, the hard fault rate (bit error rate) increases by 2.7x. For the three-layer CNN (CNN-2), the network accuracy starts dropping at around 750mV. This is because of the larger size of the network resulting in a higher number of intersections with faults. Also, in CNNs, if a particular weight or an input is erroneous, it affects multiple output values due to the high amount of reuse. Thus, the weights and activation data in CNNs have higher impact on the final accuracy as compared to MLP. Two, out of the ten test chips had intersecting faults with critical code sections at 775mV and hence, these chips start failing at higher voltage compared to the rest.



(**a**) 8-bit MLP



(**b**) 8-bit CNN-1



(**c**) 8-bit CNN-2

Figure 3.4: Change in inference accuracy with voltage. Dotted lines are results with SAME-Infer while the solid lines are without SAME-Infer.

The results with SAME-Infer are shown in Figures 3.4a, 3.4b and 3.4c using dotted lines. The red vertical line shows the minimum voltage estimated by the analytical model that can be scaled down to while having minimal impact on accuracy. For the two layer MLP and the two layer CNN, there is minimal impact on accuracy above 650mV and 625mV respectively. At 600mV, for all the ten test chips, the critical section packing failed. Thus, with SAME-Infer, about 100mV-150mV voltage reduction was achieved for the two layer MLP and 125mV-175mV reduction was achieved for the two layer CNN network. The min-VDD estimated by the analytical model for the two layer CNN (CNN-1) is 620mV, but since we used step size of 25mV, we have not shown the exact result at 620mV. However, the model estimation falls within our obtained min-VDD range.

For the three layer CNN (CNN-2), the voltage could be scaled down to 675mV with no impact on accuracy (100mV lower than baseline). At 650mV, the non-critical section packing failed. From the sensitivity analysis results, the layer-2 weights and activation have the highest quantization noise gain or the highest minimum precision requirement. Therefore, for this layer, the number of tolerable faulty bit positions is 1 from the least significant bit (LSB) for weights and activation data. The weights in this layer also form the largest data section. As a result, packing the layer's weights in a contiguous memory segment with where each memory location can have at most one faulty LSB becomes infeasible at 650mV. The best case reduction achieved for this network was 125mV. This is similar to the minimum voltage estimated by the analytical model (670mV), thus, validating the model. Once again because we used step sizes of 25mV, we have not shown the results at 670mV, but we tested on three chips at 670mV and the network accuracy gets minimally ($<2\%$) affected at that voltage.

For all three networks, SAME-Infer achieved more than 100mV min-VDD reduction. Since memories consume significant fraction of the total system energy, 100mV-175mV reduction in min-VDD of the SRAM based scratchpad memories would lead to dramatic decrease in overall system energy consumption. Also, SAME-Infer can now tolerate upto 350x higher Bit Error Rate (BER). This is critical for tolerating aging induced or other in-field failures. If a built-in-self-test (BIST) engine can periodically upload fault maps to the cloud, SAME-Infer can be run remotely and the new failures can be avoided with a simple inexpensive software patch instead of an expensive faulty hardware replacement or in-field repair.

3.5.2 Splitting up Weights to Achieve Better Packing

As seen in the three layer CNN, at 650mV, SAME-Infer fails to pack the largest and the most sensitive weight section. As mentioned in Section 3.3.3, one way to further reduce min-VDD and achieve better packing would be to split per layer's weight sections into smaller sections. The sensitivity analysis of the weights is extended to compute weight quantization noise gains on a per layer per filter basis. The results for the three layer CNN-2 are shown in Figure 3.5. An additional 50mV reduction in min-VDD was achieved for all 10 chips tested at negligible (<1%) code space overhead and no impact on accuracy compared to SAME-Infer with layer-wise monolithic weight sections.

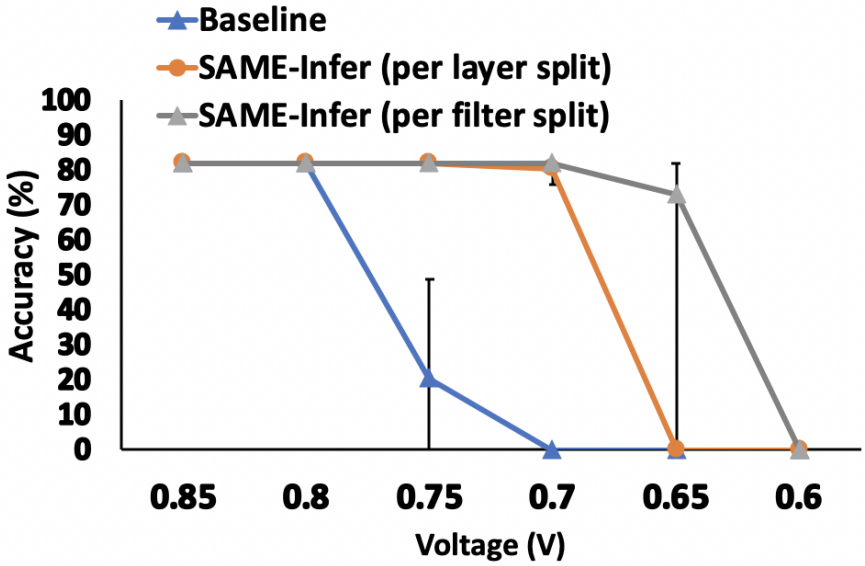


Figure 3.5: Change in three layer CNN-2 inference accuracy as voltage on the test chips is scaled down. The result shown here is the average accuracy across 10 test chips for each test case. The test cases are - (1) without SAME-Infer (2) with SAME-Infer and layerwise monolithic weight sections (3) when the weight sections are split up on per filter basis in every layer.

The weights can be further split up by granularity finer than a kernel. The best case is when each weight section is as small as a memory word. However, splitting up the data sections into such small sizes require non-negligible code modifications. Also, as can be seen in Figure 3.6, for the bit-error rate measured in our test chips and the three layer CNN-2 network, splitting up the non-critical sections to a size smaller than the kernel size doesn't really provide much benefit as the

packing gets limited by the critical sections. Therefore, splitting on a per kernel basis is often good enough and results in least intrusive code changes.

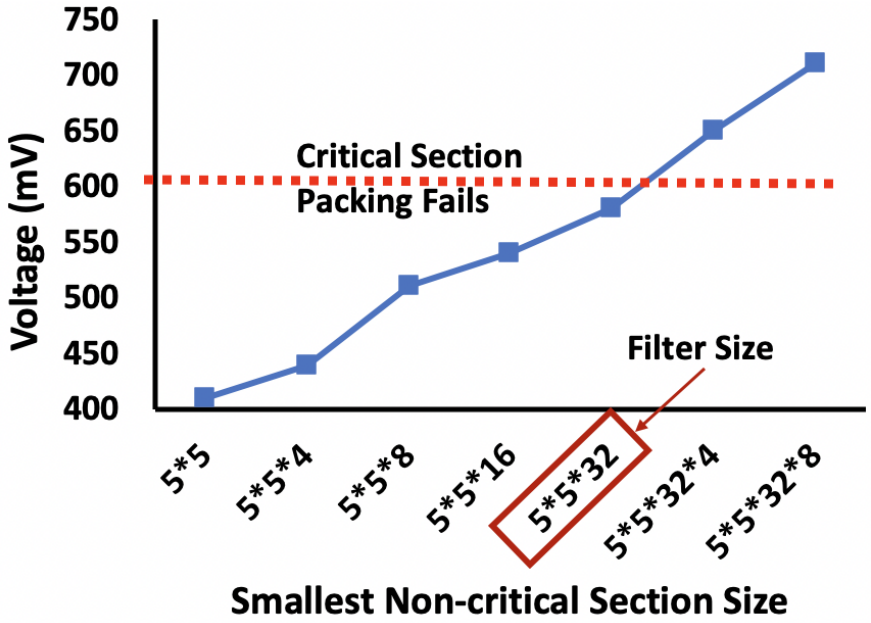


Figure 3.6: Achievable min-VDD as the smallest non-critical section size is reduced for the three layer CNN. The min-VDD is obtained using Equation 2.2 while the min-VDD for critical section is obtained from the test chip results.

We extended the analysis for a larger CNN (with 6 convolution layers and 3 fully connected layers). The network architecture is $32C3-32C3-MP2-64C3-64C3-MP2-128C3-128C3-256FC-256FC-10$. Since the network was too large for the test chip, we created a series of randomly-generated *synthetic fault maps* for memory of size 1 MB. We synthesized 10 fault maps in 10 mV increments for a total of 10 *"synthetic test chips."* We used detailed Monte Carlo simulation of SRAM bit-cell noise margins in the corresponding 45 nm technology. The filter-wise precision results were obtained in Theano [96]. The accuracy results were obtained by running inference with faulty weights and activations using our PyTorch based fault injection framework and the synthetic fault map. The results are shown in Figure 3.7. With only SAME-Infer and monolithic layer weights, the desired packing could not be obtained for layers 5, 6 and 7 below 750mV. With split weight sections, in 8 out of 10 synthetic test chips, the min-VDD achieved was 550mV with desired precision, thus, having almost no impact on accuracy. The overall min VDD reduction with

split weights was as much as 200mV compared to simple layerwise weight packing.

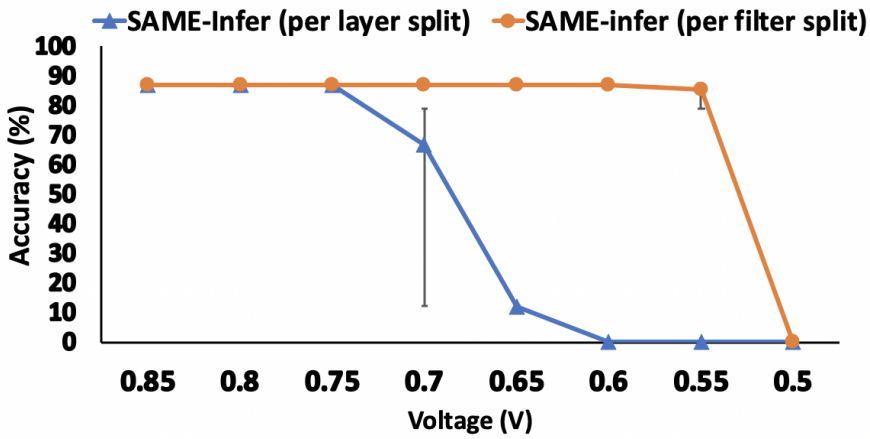


Figure 3.7: Change in nine layer CNN inference accuracy as voltage on the synthetic chips is scaled down. The result shown here is the average accuracy across 10 synthetic chips for each test case. The test cases are - (1) with SAME-Infer and layerwise monolithic weight sections (2) when the weight sections are split up on per filter basis in every layer.

3.5.3 Importance of Sensitivity Analysis of Fault Tolerant sections

For comparison against naive placement strategy, we tried placing the non-critical weights and activation data sections at the first available memory region (greedy placement - first available unused memory segment) with no notion of sensitivity or bitwise intersection with faults at 700mV for the three layer CNN network on five test chips. Thus, instead of having 5 levels of criticality, we only had two levels. The first level is for the critical sections, and the second level is for all non-critical sections with no upper bounds on the number of faulty least significant bits. So, for the non-critical sections, even the most significant bit could have a fault.

The impact on accuracy was significant (shown in Figure 3.8) because a large number of weights and activation data were intersecting with faults in the most significant bits, thus, making the chips unusable at 700mV. With the intelligent placement of the fault tolerant sections we can run the network at 700mV with negligible impact on accuracy. Thus, doing a sensitivity analysis of the fault tolerant weights and activation data and placing these sections such that only the tolerable bits intersect with faults result in more than 50mV reduction in voltage.

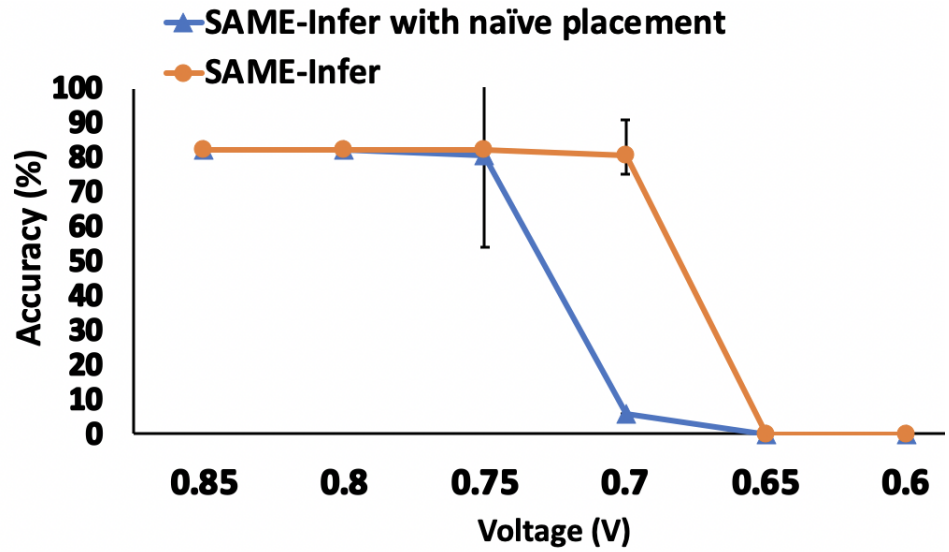


Figure 3.8: Change in the three layer CNN-2 inference accuracy as voltage on the test chips is scaled down. The result shown here is the average accuracy across 5 test chips for each test case. The test cases are - (1)when the fault tolerant sections are naively placed (greedy placement) in the memory while the critical text sections are placed in non-faulty memory regions (2) SAME-Infer with criticality aware placement.

3.5.4 Analytical Model to Estimate for Larger Sized Memories

Using the analytical model we estimated the minimum voltage (maximum BER) that the three layer CNN-2 network can tolerate if the size of the memory is increased. In most cases, the target inference systems are of standard sizes while the network sizes vary greatly. The results are shown in Figure 3.9. It can be seen that for a memory size of 512KB (instead of the 176KB in our test chip), the voltage can be scaled down to 640mV (from 670mV with 176KB) and a 2.5x higher BER can be tolerated. Once again, we used detailed Monte Carlo simulation of SRAM bit-cell noise margins in the corresponding 45 nm technology to calculate the bit error rate.

3.5.5 Evaluation for Binarized Dense and Sparse Networks

To evaluate the impact of SAME-Infer on networks that are expected to be less fault tolerant (quantized and/or sparse networks), we extended our analysis to binarized dense and sparse versions

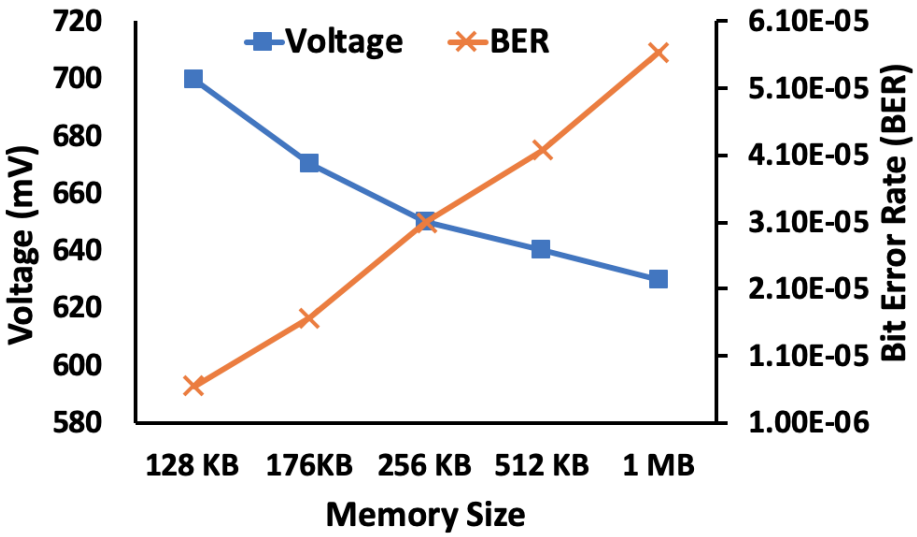


Figure 3.9: Voltage reduction or BER tolerance estimation by the analytical model for the three layer CNN-2 on different memory sizes.

of the two layer perceptron network, tested using MNIST dataset (results shown in Figures 3.10 and 3.11). For the 8-bit version of the same network, at 750mV, the network accuracy almost remains unaffected for all 10 chips. The same is true for the dense binarized version. However, in the binarized sparse MLP network, we start seeing an impact on accuracy at 750mV. This is because most of the redundant weights have been removed from the network and only the critical weights are used. Therefore, any intersection with faults results in an impact on accuracy, causing the network to have very low tolerance to faults. However, all three versions of the network can be scaled down to 650mV with SAME-Infer. The non-critical sections in the sparse network are the smallest in size and hence, can be packed perfectly even at 650mV.

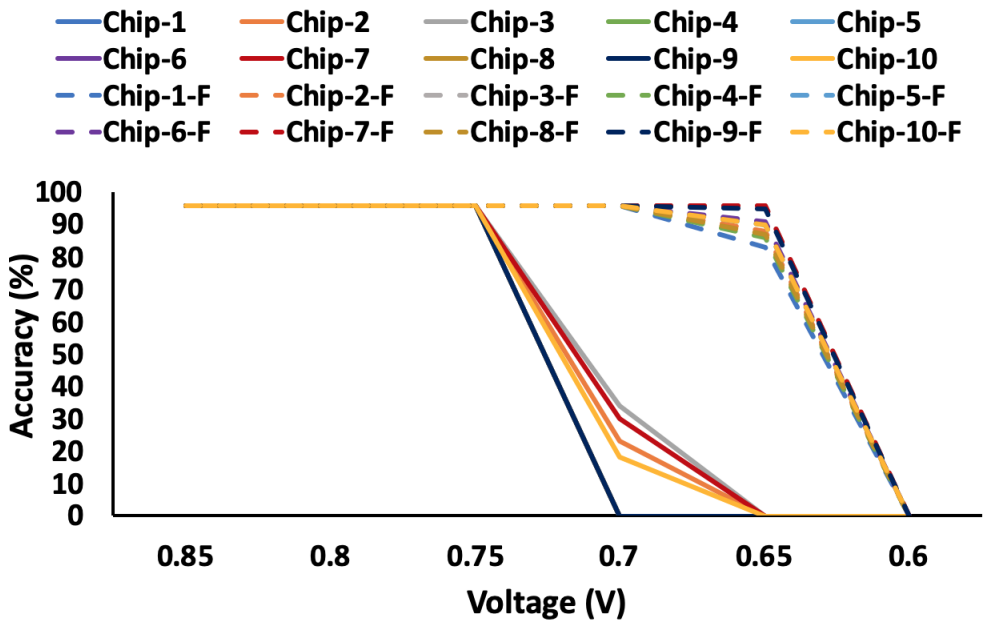


Figure 3.10: Change in dense binarized MLP inference accuracy as voltage on the test chips is scaled down.

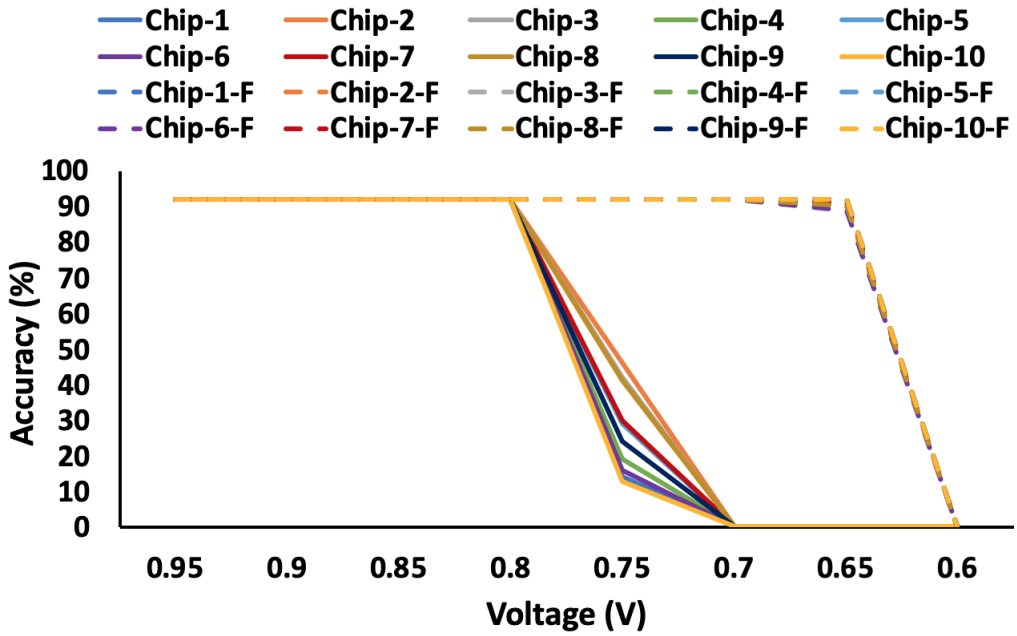


Figure 3.11: Change in sparse binarized MLP inference accuracy as voltage on the test chips is scaled down.

3.5.6 Comparison with Past Works

3.5.6.1 Treating all data sections as critical

In FaultLink (Chapter 2), we treat all program sections (data and instructions) as critical and try to fit them in fault-free segments of the memory. Doing so has two primary disadvantages. Firstly, it fails to exploit the inherent redundancy in deep learning inference (or any other approximation tolerant) applications. As a result, the packing solution would fail at a much lower fault rate (higher voltage) than what it can actually tolerate. Secondly, since this solution does not allow any intersection with faults, the actual size of the memory needs to be much higher than the size of the binary to be able to successfully pack all sections in fault free memory segments at low voltages. We compared FaultLink with SAME-Infer and the results are in Figure 3.12. SAME-Infer allows scaling by more than 100mV (average) for most applications. The 100mV voltage scaling translates to >25x higher fault tolerance as well. Thus, SAME-Infer delivers much higher energy reduction/fault tolerance as compared to FaultLink. Also, for the same memory size, SAME-Infer will be able to fit a larger sized network than FaultLink when running at the same voltage. This is critical as network sizes that are being deployed on these edge devices is increasing rapidly.

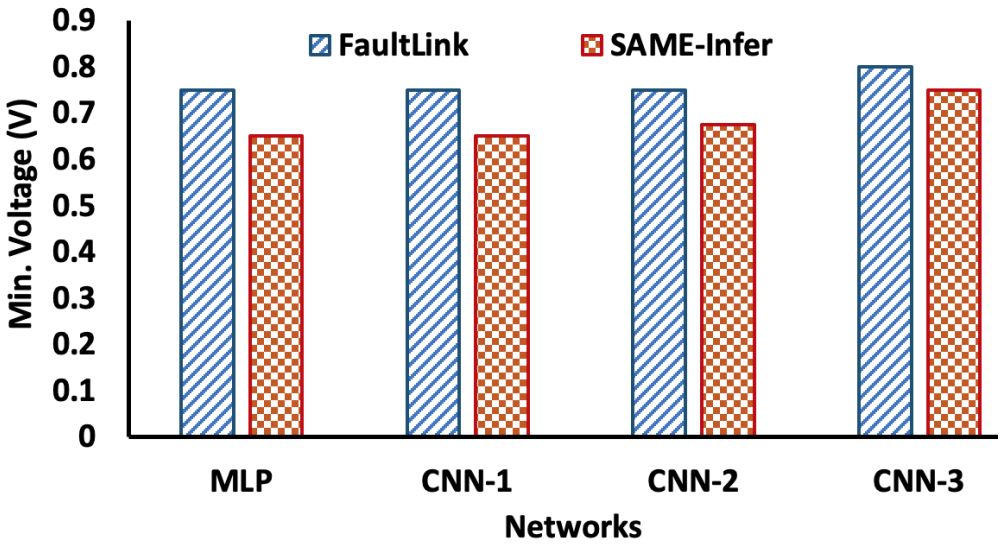


Figure 3.12: SAME-Infer achieves lower min voltage as compared to FaultLink [1] with negligible impact on accuracy because SAME-Infer allows intersection with faults in the less critical LSB bits of non-critical fault tolerant data sections.

3.5.6.2 Fault Injection During Training as an Alternative to SAME-Infer

In order to boost DNN's error tolerance, [88] proposed a curricular retraining approach. This mechanism injects errors into the DNN training procedure to boost the error tolerance of the network. A similar fault-aware training has also been proposed in [97]. We assumed a random uniform distribution of bit errors and measured the average bit error rate (BER) at every voltage across our 10 test chips. This is because training on the target faulty edge devices is impractical and hence, exact fault maps cannot be used while retraining. The authors also make a similar assumption in [88]. The results are shown in Figure 3.13. The baseline here refers to running the original trained network (without curricular retraining) on the chip at reduced voltage. The baseline min-VDD is compared against the min-VDD obtained with curricular retraining and with SAME-Infer. For the 10 test chips tested, curricular retraining helps to lower the voltage by at most 50mV in only 5 chips (a few of them see a non-negligible impact on accuracy), whereas, SAME-Infer allows voltage reduction in all of them.

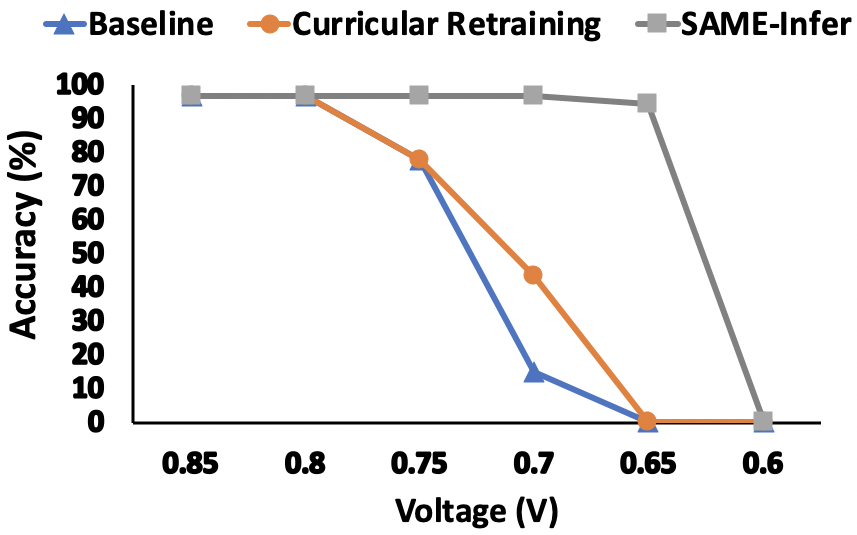


Figure 3.13: MLP (2 layer) with MNIST - average accuracy measured across 10 chips for each test case - (1) Baseline (2) Curricular Retraining (3) SAME-Infer

From the results it can be seen that curricular retraining using uniform random bit error distribution provides negligible improvement in this case. This is because, at lower voltage, faults in SRAMs tend to be correlated (shown in Figure 3.14). Hence, unless it is retrained on the target

platform, the retraining mechanism would provide very limited improvement. But retraining on target platform is likely impractical (e.g., training may have been done with proprietary datasets on high performance GPUs). Most edge platforms lack the computational power required for training these networks [98]. Large networks require tens of exaFLOPS of compute across the entire training cycle [99], making them infeasible to be run on edge devices. Moreover, faults appear in code memory as well. Therefore, only making network data (weights and activations) more resilient to faults is insufficient. In the next section, we discuss how error-injection based training may be helpful.

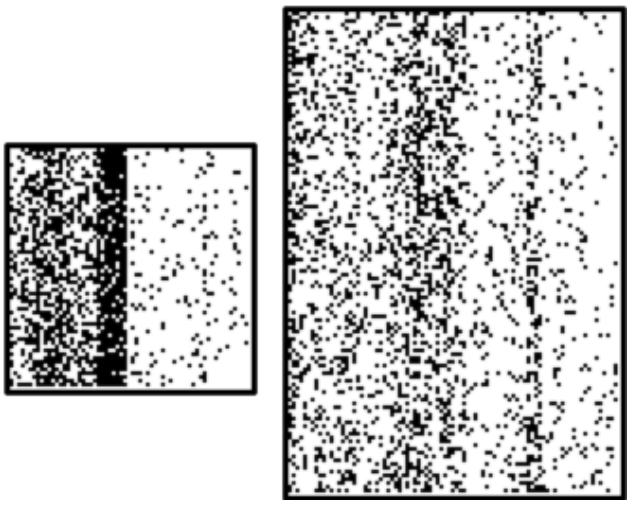


Figure 3.14: Hard Fault Map of the 64KB instruction memory (left) and the 176KB data memory (right) of a test chip. The black dots represent the faulty byte locations.

Overall we see that SAME-Infer not only allows energy saving through voltage scaling, it is also an efficient fault tolerance technique as it tolerates 25x average increase in byte error rate with minimal impact on inference accuracy. For some chips it tolerates upto 350x increase in BER with minimal to no impact on network accuracy. Since edge devices may have long lifetimes, aging becomes a concern for the reliability of the device. SAME-Infer can be used as an in-field repair technique where fault maps are periodically sampled using BIST and uploaded to cloud. SAME-Infer is then run remotely for aging induced faults and the updated binary is deployed during software updates with minimal disruption to the customers. SAME-Infer also helps to increase the increase the yield of chips by allowing usage of faulty chips leading to significant cost savings.

3.6 Discussion

In this section, we briefly discuss some of the possible extensions to SAME-Infer, which though not explored thoroughly in our current set of experimental results, can provide additional fault tolerance and/or power benefits as well as easier deployment.

3.6.1 Fault Injection During Training to Tolerate Soft Errors

As we saw in Section 3.5.6.2, curricular retraining by injecting bit errors while training does not provide much benefit against correlated hard faults at lower voltage when used on its own. However, curricular retraining or random fault injection during training can help with tackling soft errors (random memory bit flips during runtime). At lower voltage, susceptibility to radiation-induced soft faults increases because critical charge, which is the charge threshold to cause a soft error, decreases [100, 101]. Curricular retraining can be used to augment SAME-Infer to tolerate this increased soft error rate at lower voltage.

3.6.2 Improving Packing by Optional Reversing of Non-Critical Sections

SAME-Infer is able to tolerate faults in the least significant bit regions of the memory but largely leaves the most significant bit regions untouched. As a result, faults in roughly half the memory remain unaddressed by SAME-Infer. An interesting way to extend SAME-Infer is to optionally reverse the weights/activations (i.e., essentially reverse the order of bits in the byte).

In our framework, we try to pack our non-critical weight/activation sections in contiguous memory segments with no faults in the desired most significant bits. If the number of error tolerant least significant bits for a particular weight is 2, every address in the memory segment used to pack this section needs to have fault-free 5 most significant bits. Now if the weights have the option of being reversed before being stored in the memory, the packing algorithm has the option of storing the weights in either a memory segment where every address has fault free 5 MSBs or in a memory segment where every address has fault free 5 LSBs. If the latter is chosen, the weights need to be reversed. This doesn't require a pre-compilation modification to the source code for every chip. The

way to do this is to have custom load and store procedures for weights. When storing or loading weights, a particular address location is checked. If the value stored there is 1, then the weights are not reversed, if 0, then the weights need to be read from or written to in a reversed fashion. The value to be stored in that particular address can be done during link time based on the packing solution. Therefore, this is a one-time source code modification, every-time link solution (like the rest of our methodology). The probability of there not being a memory segments that is large enough to store the largest program section decreases and the updated Equation 2.2 will be:

$$P\left(L_{\texttt{size}/4} < \frac{m_{\max}}{4}\right) \approx e^{-2s^{\left(\frac{m_{\max}}{4} - \log_{s-1}\left(\frac{\texttt{size}}{4}(1-s)\right)\right)}}. \tag{3.3}$$

We evaluated this for a limited number of synthetic test chips with the nine layer CNN. For two out of ten chips, it helped to reduce the min-VDD by 50mV as compared to the baseline SAME-Infer (with no weight splitting). The obvious drawback of this approach is the additional runtime and code size overhead of checking and reversing. The code size overhead is small though the runtime overhead can be noticeable since every load operation now translates to branch, load and rotate operations. As a result, we did not explore this option further. However, this approach is useful in very high fault rate or very low power scenarios.

3.6.3 Universal Packing Solution to Allow Dynamic Voltage Scaling and Tolerate Aging Induced Faults

Hard faults in SRAMs due to voltage scaling are inclusive, that is, faults that appear at a higher voltage remain as the voltage is lowered [65]. The fault map at 600mV would include all the faults that appeared at voltages higher than 600mV (along with some additional faulty bits). Therefore, if the SAME-Infer packing is done for the lowest possible VDD, the same packing solution can be used when running the chip at higher voltages. This allows having a universal packing solution for a given chip. The application is loaded into the memory based on this solution during deployment and the voltage can be dynamically scaled during runtime without having to repack every time.

We tested the solution out on three test chips where the packing solution at 650mV was used when running at 700mV and the accuracy was the same as what was achieved when specifically

packed using the memory fault map and packing solution of 700mV.

Memory chips go through multiple rounds of burn-in [102] testing which involves a series of full chip read and write operations. We suggest having something similar for the memories in embedded chips. During burn-in, a March test equivalent can be run. As embedded memories are much smaller than standard DRAM chips, burn-in testing overhead should be small. Based on the stored fault map, the application can be packed for the lowest possible supply voltage. Having this not only saves the effort of repacking every time the voltage is scaled during runtime, it also provides protection against in-field aging induced failures. This is because the weaker cells are expected to fail and get captured in the low voltage fault map and the universal packing solution would take care of it.

3.6.4 Addressing the Code Memory Bottleneck

In several of our benchmark/chip combinations (especially for smaller networks), SAME-Infer packing failure is due to code (which is all considered critical) that is unable to get packed in code memory. There are two possible ways to address this. First, microcontroller designs can allow for separate power delivery network for code memory (so that data memory can be independently voltage scaled). Second, more intrusive changes to the machine learning code can be made to build it from smaller functions. This would have a negligibly small impact on code size and runtime but will result in more packable code in presence of faults in code memory (as every function can be mapped to different memory segment).

3.6.5 Use of Error Correcting Codes (ECC)

ECC is a common approach for error detection and correction in memories. However, they are better suited for random, temporary faults and incur area, performance and energy overheads. If a t-bit error correcting code is used, i.e., errors upto t-bits can be detected and corrected by the code, all k-bit messages get encoded into (k+r)-bit codewords before they are stored in the memory. The extra r-bits of parity are added onto the original message to enable error correction. As the code becomes stronger or the requirement for t increases, the number of parity bits (r) also increases.

During a read operation, the encoded message is loaded from the memory and decoded such that the original k-bit message can be recovered and errors upto t-bit can be detected and corrected. The additional parity bit storage as well as the encoding and decoding overheads are non-negligible and increase rapidly as the correction capability of the code increases. As a result, they can be an overkill (and therefore a bad approach) for permanent faults.

Our experiments revealed that the hard faults at lower voltages normally tend to be correlated. As a result, multi-bit error correction would be required. While SAME-Infer can tolerate up to 4-bit faults on 8-bit weights/activations with negligible performance overheads, an double-bit error correcting (DEC) code would require 8 additional bits of parity and have 2 cycles of encoding and decoding latency. For example, the 9-layer CNN-3 network has a total size of $\sim$700KB. To fit this network at nominal voltage with no faults, at least 1.4MB of memory is needed if DEC code is used for protection. Moreover, around 650mV-670mV, triple bit errors start appearing and thus, lowering the voltage further will lead to loss in accuracy as the un-correctable errors can coincide with MSBs. On the other hand, with SAME-Infer, we managed to fit in the entire network within a 1MB memory and could lower our voltage to less than 600mV with no impact on accuracy (Figure 3.7). However, if ECC is available, it should augment SAME-Infer to address soft errors during runtime.

3.6.6 Extending SAME-Infer to Other Approximation Tolerant Applications

The SAME-Infer framework can be easily extended to other approximation tolerant applications. There are several applications in the field of approximate computing that can tolerate controlled relaxation of correctness for improving performance or energy efficiency. For such workloads, already existing frameworks (such as Approxilyzer [103]) can be used to identify the non-critical approximation friendly sections of the code that can intersect with faults without impacting output quality. Once the one-time analysis of the code is done, SAME-Infer can be used to correctly map the critical and non-critical sections in non-faulty and faulty memory segments respectively, leading to lower energy or higher fault tolerance.

3.7 Conclusion

Design of edge devices is driven by the need for the lowest possible cost and energy consumption, which are both strongly affected by on-chip memories. Further, many of these may be deployed in harsh environments where in-field replacement is difficult due to faults. The proposed SAME-Infer methodology addresses both these issues for embedded scratchpad memories running machine learning applications. SAME-Infer uses the linker to map the critical code/layers onto the non-faulty segments of the memory while the fault-tolerant sections of data are placed in faulty memory segments. This allows SAME-Infer to tolerate upto 350x (average 25x) higher bit error rate without degrading inference accuracy. Our evaluations on 10 real micro-controller class chips and 10 larger synthetic chips show that up-to 175mV reduction in voltage can be achieved without any loss in accuracy for a fully connected network and for two convolutional neural networks. Thus, SAME-Infer helps to tolerate higher hard fault rate by exploiting the redundancies in the DL applications and helps in cost savings by making error prone memory chips usable.

CHAPTER 4

Software-Defined Error-Localizing Codes (SDELC): Lightweight Recovery from Soft Faults at Run-Time

For embedded memories, it is always challenging to address reliability concerns as additional area, power, and latency overheads of reliability techniques need to be minimized as much as possible. *Software-Defined Error-Localizing Codes* (SDELC) is a hybrid hardware/software technique that deals with single-bit soft faults at run-time using novel *Ultra-Lightweight Error-Localizing Codes* (UL-ELC) with a software-defined error handler that knows about the UL-ELC construction and implements a heuristic recovery policy. UL-ELC codes are stronger than basic single-error-detecting (SED) parity, yet they have lower storage overheads than a single-error-correcting (SEC) Hamming code. Like SED, UL-ELC codes can detect single-bit errors, yet they can additionally *localize* them to a *chunk* of the erroneous codeword. UL-ELC codes can be explicitly designed such that chunks align with meaningful message context, such as the fields of an encoded instruction. SDELC then relies on *side information* (SI) about application memory contents to heuristically recover from the single-bit fault. It focuses on heuristic error recovery that is suitable for microcontroller-class IoT devices.

Collaborators:

- Dr. Mark Gottscho, UCLA/Google
- Dr. Clayton Schoeny, UCLA/Square
- Prof. Lara Dolecek, UCLA
- Prof. Puneet Gupta, UCLA

4.1 Introduction

For embedded systems at the edge of the Internet-of-Things (IoT), hardware design is driven by the need for the lowest possible cost and energy consumption, which are both are strongly affected by on-chip SRAM-based memories [34]. With technology scaling the feature size of each SRAM memory cell is reducing. Thus, the amount of critical charge required to flip the cell contents is also decreasing. At the same time, the total memory capacity in these devices is increasing, thereby, consuming significant fraction of chip real estate. As a result, the probability of radiation induced soft faults affecting memory cells is also rapidly increasing. Overall, the rate of soft faults affecting the on-chip memory during runtime is rising. One way to recover from soft errors is by having error detection and correction (EDAC) schemes. However, the standard EDAC techniques used in today's systems usually incur large area, power, and performance overheads.

Our goal in this chapter is to recover from unpredictable single-bit soft faults in embedded memory at minimal cost. We propose *Software-Defined Error-Localizing Codes* (SDELC), a hybrid hardware/software technique that allows the system to heuristically recover from unpredictable single-bit soft faults in instruction and data memories. SDELC relies on *side information* (SI) about application memory contents, i.e., observable patterns and structure found in both instructions and data. We describe the novel class of *Ultra-Lightweight Error-Localizing Codes* (UL-ELC) that are used by SDELC. UL-ELC codes are stronger than basic single-error-detecting (SED) parity, yet they have lower storage overheads than a single-error-correcting (SEC) Hamming code. Like SED, UL-ELC codes can detect single-bit errors, yet they can additionally *localize* them to a *chunk* of the erroneous codeword. UL-ELC codes can be explicitly designed such that chunks align with meaningful message context, such as the fields of an encoded instruction.

We find that our SDELC technique recovers from up to 90% of random single-bit soft faults in 32-bit data memory words and up to 70% of errors in instruction memory using a 3-bit UL-ELC code (9.375% storage overhead). SDELC can even be used to recover up to 70% of errors using a basic SED parity code (3.125% storage overhead). In contrast, a full Hamming SEC code incurs a storage overhead of 18.75%.

4.2 Background

This section highlights the essential background on the error correcting codes (ECC), and error-localizing codes that is needed to understand our contributions.

4.2.1 Error-Correcting Codes (ECCs)

ECCs are mathematical techniques that transform *message* data stored in memory into *codewords* using a hardware encoder to add redundancy for added protection against faults. When soft faults affect codewords, causing bit flips, the ECC hardware decoder is designed to detect and/or correct a limited number of errors. ECCs used for random-access memories are typically based on linear block codes.

The encoder implements a binary generator matrix $\mathbf{G}$ and the complementary decoder implements the parity-check matrix $\mathbf{H}$ to detect/correct errors. To encode a binary message $\vec{m}$, one multiplies its bit-vector by $\mathbf{G}$ to obtain the codeword $\vec{c}$: $\vec{m}\mathbf{G} = \vec{c}$. To decode, one multiplies the stored codeword (which may have been corrupted by errors) with the parity-check matrix to obtain the syndrome $\vec{s}$, which provides error detection and correction information: $\mathbf{H}\vec{c}^{T} = \vec{s}$. Typical ECCs used for memory have the generator and parity-check matrices in systematic form, i.e., the message bits are directly mapped into the codeword and the redundant parity bits are appended to the end of the message. This makes it easy to directly extract message data in the common case when no errors occur.

Typical ECC-based approaches can tolerate random bit-level soft faults but they quickly become ineffective when multiple errors occur due to hard faults. Meanwhile, powerful schemes like ChipKill [104] have unacceptable overheads and are not suited for embedded memories. In this work, we propose novel ECC constructions that have very low overheads, making them suitable for low-cost IoT devices that may experience occasional single-bit SEUs.

4.2.2 Error-Localizing Codes

In 1963, Wolf et al. introduced *error-localizing codes* (ELC) that attempt to detect errors and identify the erroneous fixed-length chunk of the codeword. Wolf established some fundamental bounds [105] and studied how to create them using the tensor product of the parity-check matrices of an error-detecting and an error-correcting code [106]. ELC has been adapted to byte-addressable memory systems [107] but until now, they had not gained any traction in the systems community.

To the best of our knowledge, ELCs in the regime between SED and SEC capabilities has not been previously studied. We describe the basics of *Ultra-Lightweight ELC* (UL-ELC) that lies in this regime and apply specific constructions to recover from a majority of single-bit soft faults.

4.3 Software-Defined Error-Localizing Codes (SDELC): Recovering Soft Faults at Run-Time

In today's systems, either basic SED parity is used to detect random single-bit errors or a Hamming SEC code is used to correct them. Unfortunately, Hamming codes are expensive for small embedded memories: they require six bits of parity per memory word size of 32 bits (an 18.75% storage overhead). On the other hand, basic parity only adds one bit per word (3.125% storage overhead), but without assistance by other techniques it cannot correct any errors.

SDELC is a novel solution that lies in between these regimes. A key component is the new class of *Ultra-Lightweight Error-Localizing Codes* (UL-ELCs). UL-ELCs have lower storage overheads than Hamming codes: they can detect and then *localize* any single-bit error to a chunk of a memory codeword. We construct distinct UL-ELC codes for instruction and data memory that allows a software-defined recovery policy to heuristically recover the error by applying different semantics depending on the error location. The policies leverage available *side information* (SI) about memory contents to choose the most likely *candidate codeword* resulting from a localized bit error. In this manner, we attempt to correct a majority of single-bit soft faults without resorting to a stronger and more costly Hamming code.

This section covers, in detail, the SDELC architecture, the concept of UL-ELC codes, and two

SDELC recovery policies for instruction and data memory.

4.3.1 Architecture

The SDELC architecture is illustrated in Figure 4.1 for a system with split on-chip instruction and data SPMs (each with its own UL-ELC code) and a single-issue core that has an in-order pipeline.

When a codeword containing a single-bit soft fault is read, the UL-ELC decoder detects and localizes the error to a specific chunk of the codeword and places error information in a *Penalty Box* register (shaded in gray in the figure). A precise exception is then generated, and software traps to a handler that implements the appropriate SDELC recovery policy for instructions or data, which we will discuss shortly.

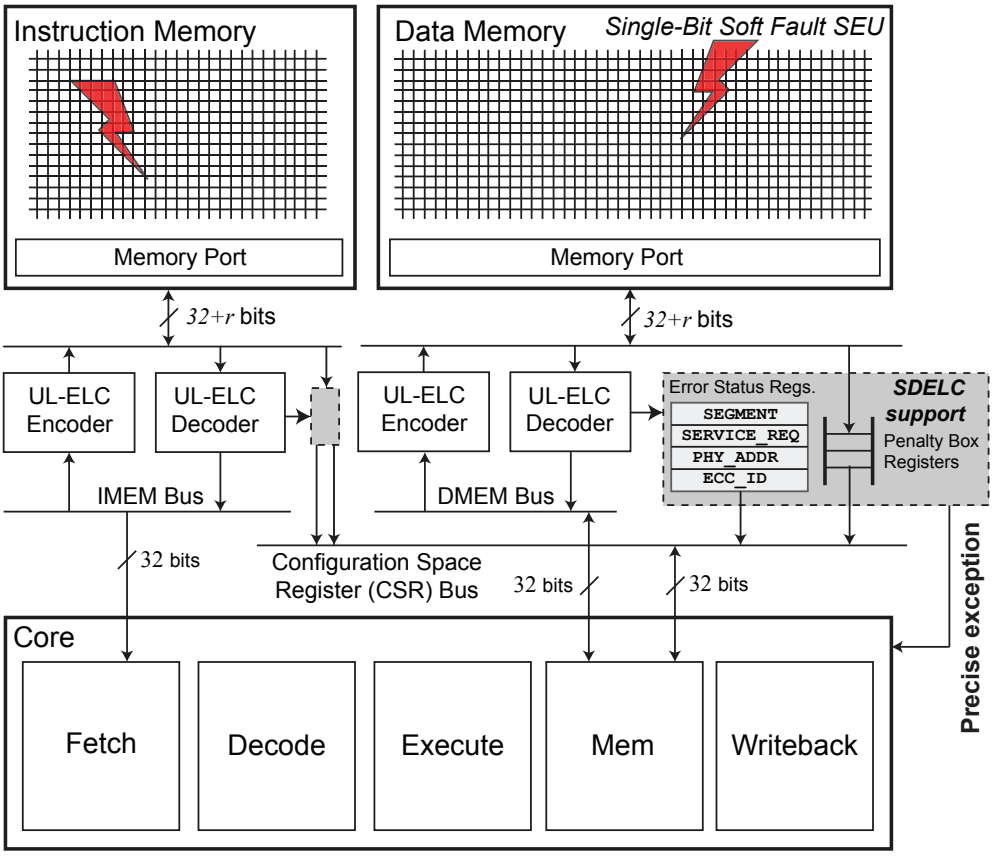


Figure 4.1: Architectural support for SDELC on an microcontroller-class embedded system.

Once the trap handler has decided on a candidate codeword for recovery, it must correctly commit the state in the system such that it appears *as if* there was no memory control flow disruption. For

instruction errors, because the error occurred during a fetch, the program counter (pc) has not yet advanced. To complete the trap handler, we write back the candidate codeword to instruction memory. If it is not accessible by the load/store unit, one could use hardware debug support such as JTAG. We then return from the trap handler and re-execute the previously-trapped instruction, which will then cause the pc to advance and re-fetch the instruction that had been corrupted by the soft error. On the other hand, data errors are triggered from the memory pipeline stage by executing a load instruction. We write back the chosen candidate codeword to data memory to scrub the error, update the register file appropriately, and manually advance pc before returning from the trap handler.

4.3.2 Ultra-Lightweight Error-Localizing Codes (UL-ELC)

Localizing an error is more useful than simply detecting it. If we determine the error is from a *chunk* of length ℓ bits, there are only ℓ *candidate codewords* for which a single-bit error could have produced the received (corrupted) codeword.

A naïve way of localizing a single-bit error to a particular chunk is to use a trivial segmented parity code, i.e., we can assign a dedicated parity-bit to each chunk. However, this method is very inefficient because to create C chunks we need C parity bits: essentially, we have simply split up memory words into smaller pieces.

We create simple and custom *Ultra-Lightweight* ELCs (UL-ELCs) that – given r redundant parity bits – can localize any single-bit error to one of $C = 2^r - 1$ possible chunks. This is because there are $2^r - 1$ distinct non-zero columns that we can use to form the parity-check matrix $\mathbf{H}$ for our UL-ELC (for single-bit errors, the error syndrome is simply one of the columns of $\mathbf{H}$). To create a UL-ELC code, we first assign to each chunk a distinct non-zero binary column vector of length r bits. Then each column of $\mathbf{H}$ is simply filled in with the corresponding chunk vector. Note that r of the chunks will also contain the associated parity-bit within the chunk itself; we call these *shared chunks*, and they are precisely the chunks whose columns in $\mathbf{H}$ have a Hamming weight of 1. Since there are r shared chunks, there must be $2^r - r - 1$ *unshared chunks*, which each consist of only data bits. Shared chunks are unavoidable because the parity bits must also be protected against faults,

just like the message bits.

UL-ELCs form a middle-ground between basic parity SED error-detecting codes (EDCs) and Hamming SEC ECCs. In the former case, $r = 1$, so we have a $C = 1$ monolithic chunk ($\mathbf{H}$ is a row vector of all ones). In the latter case, $\mathbf{H}$ uses each of the $2^r - 1$ possible distinct columns exactly once: this is precisely the $(2^r - 1, 2^r - r - 1)$ Hamming code. An UL-ELC code has a minimum distance of two bits by construction to support detection and localization of single-bit errors. Thus, the set of candidate codewords must also be separated from each other by a Hamming distance of exactly two bits. (A minimum codeword distance of two bits is required for SED, while three bits are needed for SEC, etc.)

For *an example* of an UL-ELC construction, consider the following $\mathbf{H}_{\text{example}}$ parity-check matrix with nine message bits and $r = 3$ parity bits:

$$\mathbf{H}_{\text{example}} = \begin{array}{c} \\ \\ c_1 \\ c_2 \\ c_3 \end{array} \begin{array}{c} \begin{array}{cccccccccccc} S_1 & S_2 & S_3 & S_4 & S_4 & S_5 & S_6 & S_6 & S_7 & S_5 & S_6 & S_7 \\ d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 & d_9 & p_1 & p_2 & p_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array},$$

where d_i represents the ith data bit, p_j is the jth redundant parity bit, c_k is the kth parity-check equation, and S_l enumerates the distinct error-localizing chunk that a given bit belongs to. Because $r = 3$, there are $N = 7$ chunks. Bits $d_1, d_2,$ and d_3 each have the SEC property because no other bits are in their respective chunks. Bits d_4 and d_5 make up an unshared chunk S_4 because no parity bits are included in S_4. The remaining data bits belong to shared chunks because each of them also includes at least one parity bit. Notice that any data or parity bits that belong to the same chunk S_l have identical columns of $\mathbf{H}$, e.g., d_7, d_8, and p_2 all belong to S_6 and have the column $[0;1;0]$.

The two key properties of UL-ELC (that do not apply to generalized ELC codes) are: *(i)* the length of the data message is independent of r, and *(ii)* each chunk can be an arbitrary length. The freedom to choose the length of the code and chunk sizes allow the UL-ELC design to be highly adaptable. Additionally, UL-ELC codes can offer SEC protection on up to $2^r - r - 1$ selected

Table 4.1: Proposed 7-Chunk UL-ELC Construction with $r = 3$ for Instruction Memory (RV64G ISA v2.0)

bit →	31 27	26 25	24 20	19 15	14 12	11 7	6 0	-1 -3
Type-UJ	imm[20|10:1|11|19:12]					rd	opcode	parity
Type-U	imm[31:12]					rd	opcode	parity
Type-I	imm[11:0]			rs1	funct3	rd	opcode	parity
Type-R	funct7		rs2	rs1	funct3	rd	opcode	parity
Type-S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	parity
Type-SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	parity
Type-R4	rs3	funct2	rs2	rs1	funct3	rd	opcode	parity

Chunk	C_1 (shared)	C_2 (shared)	C_3 (shared)	C_4	C_5	C_6	C_7	C_3	C_2	C_1
Parity-	11111	00	00000	11111	000	11111	1111111	1	0	0
Check	00000	11	00000	00000	111	11111	1111111	0	1	0
H	00000	00	11111	11111	111	00000	1111111	0	0	1

message bits by having the unshared chunks each correspond to a single data bit.

4.3.3 Recovering SEUs in Instruction Memory

We describe an UL-ELC construction and recovery policy for dealing with single-bit soft faults in instruction memory. The code and policy are jointly crafted to exploit SI about the ISA itself. Our SDELC implementation targets the open-source and free 64-bit RISC-V (RV64G) ISA [52], but the approach is general and could apply to any other fixed-length or variable-length RISC or CISC ISA. Note that although RISC-V is actually a little-endian architecture, for sake of clarity we use big-endian in this chapter.

Our UL-ELC construction for instruction memory has seven chunks that align to the finest-grain boundaries of the different fields in the RISC-V codecs. These codecs, the chunk assignments, and the complete parity-check matrix **H** are shown in Table 4.1. The bit positions -1, -2, and -3 correspond to the three parity bits that are appended to a 32-bit instruction in memory. The `opcode`, `rd`, `funct3`, and `rs1` fields are the most commonly used – and potentially the most critical – among the possible instruction encodings, so we assign each of them a dedicated chunk that is unshared

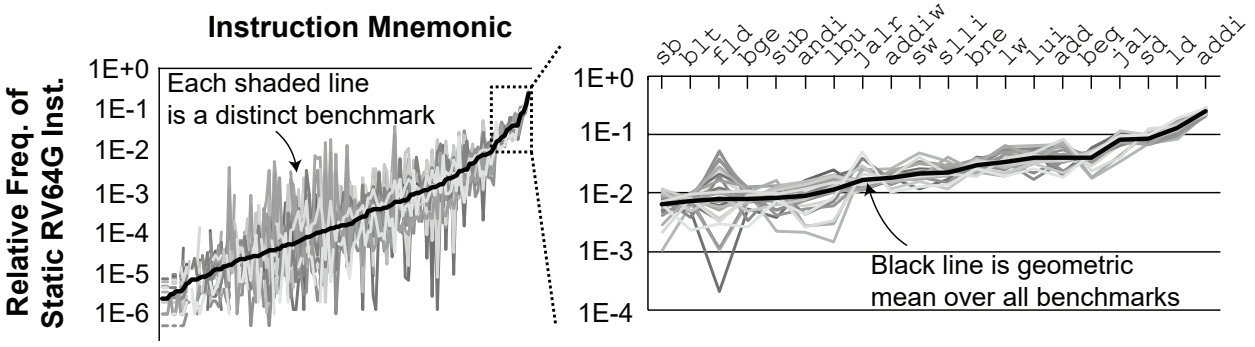


Figure 4.2: The relative frequencies of static instructions roughly follow power law distributions. Results shown are for RISC-V with 20 SPEC CPU2006 benchmarks; we observed similar trends for MIPS and Alpha, as well as dynamic instructions.

with the parity bits. The fields which vary more among encodings are assigned to the remaining three shared chunks, as shown in the table. The recovery policy can thus distinguish the impact of an error in different parts of the instruction. For example, when a fault affects shared chunk C_1, the fault is either in one of the five MSBs of the instruction, or in the last parity bit. Conversely, when a fault is localized to unshared chunk C_7 in Table 4.1, the UL-ELC decoder can be certain that the `opcode` field has been corrupted.

Consider another example with a fault in the unshared chunk C_6 that guards the `rd` destination register address field for most instruction codecs. Suppose bit 7 (the least-significant bit of chunk C_6/`rd`) is flipped by a fault. Assume the original instruction stored in memory was `0x0000beef`, which decodes to the assembly code `jal t4, 0xb000`. The 5-bit `rd` field is protected with our UL-ELC construction using a dedicated unshared chunk C_6. Thus, the candidate messages are the following instructions:

```
<0x0000b66f> jal a2, 0xb000
<0x0000ba6f> jal s4, 0xb000
<0x0000beef> jal t4, 0xb000
<0x0000bc6f> jal s8, 0xb000
<0x0000bf6f> jal t5, 0xb000.
```

Our instruction recovery policy can decide which destination register is most likely for the `jal`

instruction based on program statistics collected a priori via static or dynamic profiling (the SI). The instruction recovery policy consists of three steps.

- Step 1. We apply a software-implemented instruction decoder to filter out any candidate messages that are illegal instructions. Most bit patterns decode to illegal instructions in three RISC ISAs we characterized: 92.33% for RISC-V, 72.44% for MIPS, and 66.87% for Alpha. This can be used to dramatically improve the chances of a successful SDELC recovery.

- Step 2. Next, we estimate the probability of each valid message using a small pre-computed lookup table that contains the relative frequency that each instruction appears. We find that the relative frequencies of legal instructions follow power-law distribution, as shown by Figure 4.2. This is used to favor more common instructions.

- Step 3. We choose the instruction that is most common according to our SI lookup table. In the event of a tie, we choose the instruction with the longest leading-pad of 0s or 1s. This is because in many instructions, the MSBs represent immediate values (as shown in Table 4.1). These MSBs are usually low-magnitude signed integers or they represent 0-dominant function codes.

If the SI is strong, then we would expect to have a high chance of correcting the error by choosing the right candidate.

4.3.4 Recovering SEUs in Data Memory

In general-purpose embedded applications, data may come in many different types and structures. Because there is no single common data type and layout in memory, we propose to simply use evenly-spaced UL-ELC constructions and grant the software trap handler additional control about how to recover from errors, similar to the general idea from SuperGlue [108].

We build SDELC recovery support into the embedded application as a small C library. The application can push and pop custom SDELC error handler functions onto a registration stack. The handlers are defined within the scope of a subroutine and optionally any of its callees and can define

specific recovery behaviors depending on the context at the time of error. Applications can also enable and disable recovery at will.

When the application does not disable recovery nor specify a custom behavior, all data memory errors are recovered using a default error handler implemented by the library. The default handler computes the average Hamming distance to nearby data in the same 64-byte chunk of memory (similar to taking the intra-cacheline distance in cache-based systems). The candidate with the minimum average Hamming distance is selected. This policy is based on the observation that spatially-local and/or temporally-local data tends to also be correlated, i.e., it exhibits *value locality* [109] that has been used in numerous works for cache and memory compression [110–112]. The Hamming distance is a good measure of data correlation, as shown later in Figure 4.5.

The application-defined error handler can specify recovery rules for individual variables within the scope of the registered subroutine. They include globals, heap, and stack-allocated data. This is implemented by taking the runtime address of each variable requiring special handling. For instance, an application may wish critical data structures to never be recovered heuristically; for these, the application can choose to force a crash whenever a soft error impacts their memory addresses. The SDELC library support can increase system reliability, but the programmer is required to spend effort annotating source code for error recovery. This is similar to annotation-based approaches taken by others for various purposes [51, 69, 71–73, 113].

4.4 Evaluation - Soft Fault Recovery using SDELC

To evaluate SDELC, we modified Spike simulator [114] to produce representative memory access traces of 11 benchmarks as they run to completion. Five benchmarks are `blowfish` and `sha` from the mibench suite [49] as well as `dhrystone`, `matmulti` and `whetstone`. The remaining six benchmarks were added from the AxBench approximate computing C/C++ suite [51]: `blackscholes`, `fft`, `inversek2j`, `jmeint`, `jpeg`, and `sobel`. Each trace was analyzed offline using a MATLAB model of SDELC. For each workload, 1000 instruction fetches and 1000 data reads were randomly selected from the trace and exhaustively all possible single-bit faults were applied to each of them.

SDELC recovery of the random soft faults was evaluated using three different UL-ELC codes

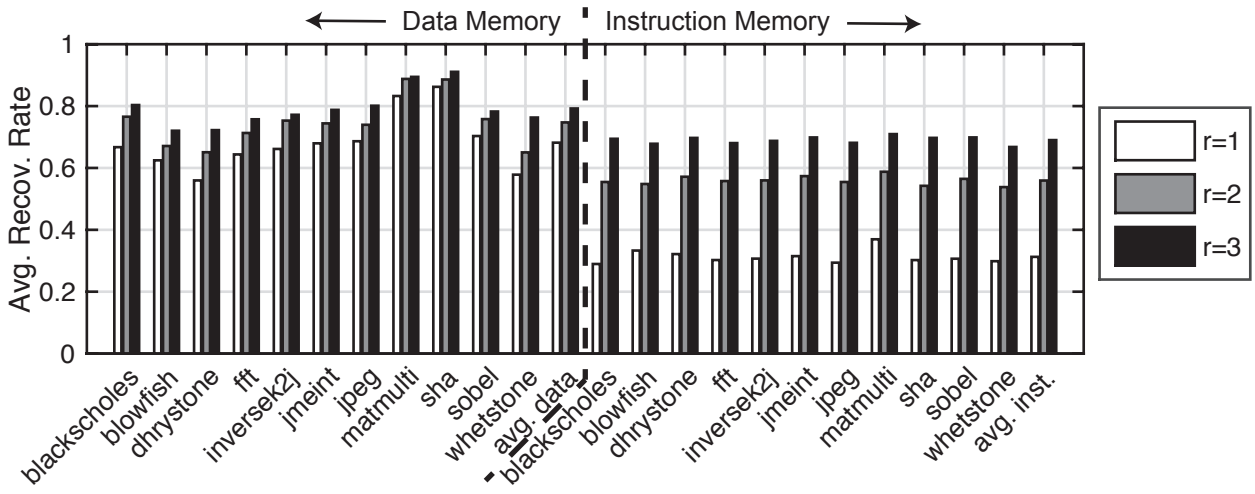


Figure 4.3: Average rate of recovery using SDELC from single-bit soft faults in instruction and data memory. r is the number of parity bits in the UL-ELC construction.

($r = 1,2,3$). Recall that the $r = 1$ code is simply a single parity bit, resulting in 33 candidate codewords. (For basic parity, there are 32 message bits and one parity bit, so there are 33 ways to have had a single-bit error.) For the data memory, the UL-ELC codes were designed with the chunks being equally sized: for $r = 2$, there are either 11 or 12 candidates depending on the fault position (34 bits divided into three chunks), while for $r = 3$ there are always five candidates (35 bits divided into seven chunks). For the instruction memory, chunks are aligned to important field divisions in the RV64G ISA. Chunks for the $r = 2$ UL-ELC construction match the fields of the Type-U instruction codecs (the `opcode` being the unshared chunk). Chunks for the $r = 3$ UL-ELC code align with fields in the Type-R4 codec (as presented in Table 4.1). A *successful recovery* for SDELC occurs when the policy corrects the error; otherwise, it fails by accidentally mis-correcting.

4.4.1 Overall Results

The overall SDELC results are presented in Figure 4.3. The recovery rates are relatively consistent over each benchmark, especially for instruction memory faults, providing evidence of the general efficacy of SDELC. One important distinction between the memory types is the sensitivity to the number r of redundant parity bits per message. For the data memory, the simple $r = 1$ parity yielded surprisingly high rates of recovery using our policy (an average of 68.2%). Setting r to three parity

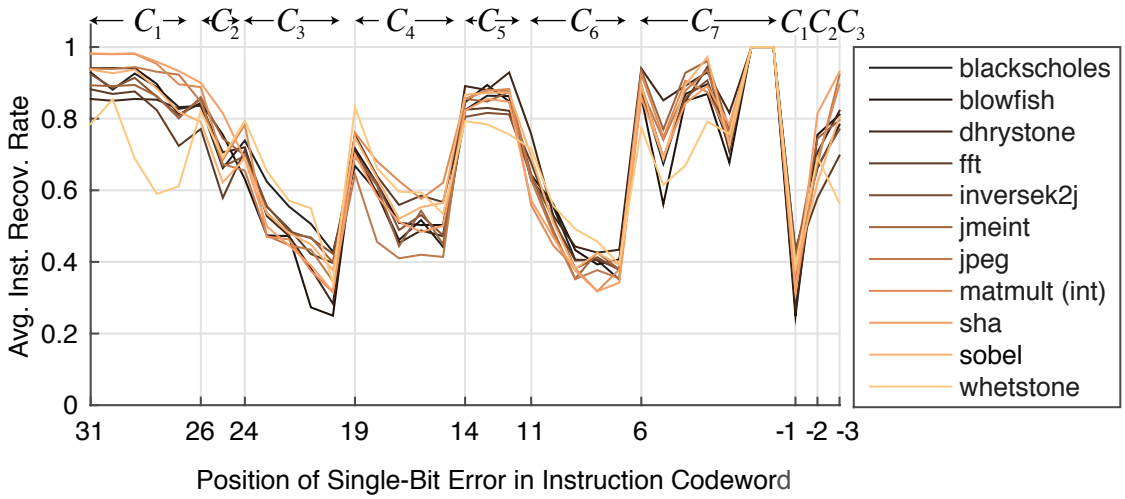


Figure 4.4: Sensitivity of SDELC instruction recovery to the actual position of the single-bit fault with the $r = 3$ UL-ELC construction.

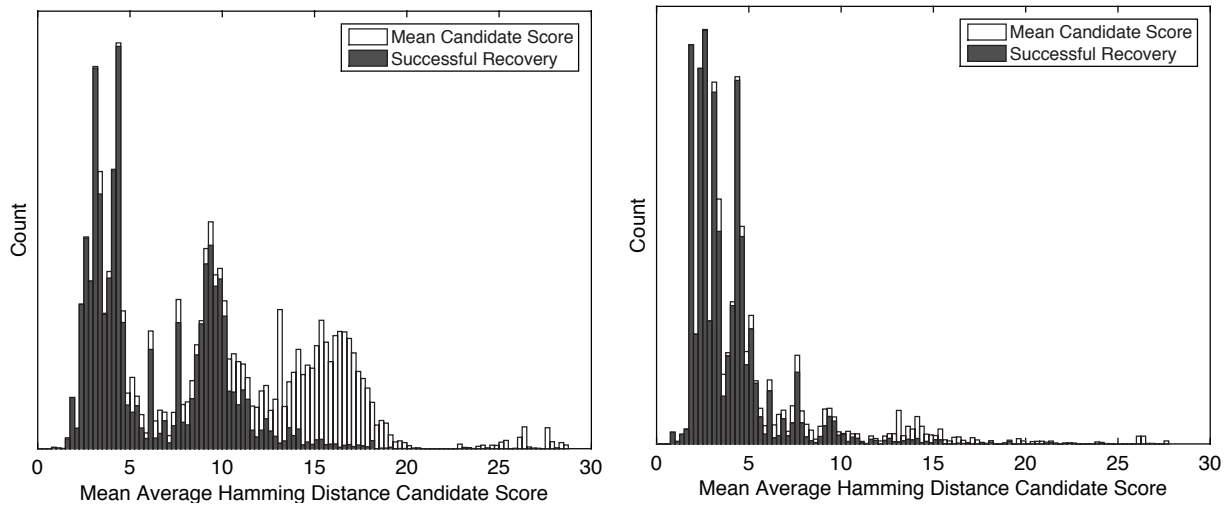
bits increases the average recovery rate to 79.2% thanks to fewer and more localized candidates to choose from. On the other hand, for the instruction memory, the average rate of recovery increased from 31.3% with a single parity bit to 69.0% with three bits.

These results are a significant improvement over a guaranteed system crash as is traditionally done upon error detection using single-bit parity. Moreover, these results are achieved using no more than half the overhead of a Hamming SEC code, which can be a significant cost savings for small IoT devices. Based on these results, using $r = 1$ parity for data seems reasonable, while $r = 3$ UL-ELC constructions can be used to achieve 70% recovery for both memories with minimal overhead.

4.4.2 Recovery Policy Analysis

The average instruction recovery rate as a function of bit error position for all benchmarks is shown in Figure 4.4. Error positions -1, -2, and -3 correspond to the three parity bits in the UL-ELC construction from Table 4.1.

It is observed that the SDELC recovery rate is highly dependent on the erroneous chunk. For example, errors in chunk C_7 – which protects the RISC-V `opcode` instruction field – have high rates of recovery because the power-law frequency distributions of legal instructions are a very strong



(a) `blowfish` **(b)** `sha`

Figure 4.5: Sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks and $r = 1$ parity code.

form of side information. Other chunks with high recovery rates, such as C_1 and C_5, are often (part of) the `funct2`, `funct7`, or `funct3` conditional function codes that similarly leverage the power-law distribution of instructions. Moreover, many errors that impact the `opcode` or function codes cause several candidate codewords to decode to illegal instructions, thus filtering the number of possibilities that the recovery policy has to consider. For errors in the chunks that often correspond to register address fields (C_3, C_4, and C_6), recovery rates are less because the side information on register usage by the compiler is weaker than that of instruction relative frequency. However, errors towards the most-significant bits within these chunks recover more often than the least-significant bits because they can also correspond to immediate operands. Indeed, many immediate operands are low-magnitude signed or unsigned integers, causing long runs of 0s or 1s to appear in encoded instructions. These cases are more predictable, so they are recovered frequently, especially for chunk C_1 which often represents the most-significant bits of an encoded immediate value.

The sensitivity of SDELC data recovery to the mean candidate Hamming distance score for two benchmarks is shown in Figure 4.5. White bars represent the relative frequency that a particular Hamming distance score occurs in our experiments. The overlaid gray bars represent the fraction of those scores that were successfully recovered using the described policy.

When nearby application data in memory is correlated, the mean candidate Hamming distance is low, and the probability that we successfully recover from the single-bit soft fault is high using the Hamming distance-based policy. Because applications exhibit spatial, temporal, and value locality [109] in memory, correct recovery is possible in a majority of cases. On the other hand, when data has very low correlation – essentially random information — SDELC does not recover any better than taking a random guess of the bit-error position within the localized chunk, as expected.

4.4.3 Risk of SDCs from SDELC

SDELC introduces a risk of mis-correcting single-bit soft faults that cannot be avoided unless one resorts to a full Hamming SEC code. However, for low-cost IoT devices running approximation-tolerant applications, SDELC reduces the parity storage overhead by up to 6× compared to Hamming while still recovering most single-bit faults. Similar to observations by others [115], we found that no more than 7.2% of all single-bit instruction faults and 2.3% of data faults result in an intolerable silent data corruption (SDC), i.e., an SDC with more than 10% output error [51]. The rest of the faults are either successfully corrected, benign, or cause crashes/hangs. The latter are no worse than crashes from commonly-used SED parity. Current SED-based systems' reliability could be improved with remote software updates to incorporate our techniques.

4.5 Conclusion

SDELC implements low-overhead heuristic error correction to cope with random single-event upsets in memory without the higher area and energy costs of a full Hamming code. Our SDELC technique recovers from up to 90% of random single-bit soft faults in 32-bit data memory words and up to 70% of errors in instruction memory using a 3-bit UL-ELC code (9.375% storage overhead). SDELC can even be used to recover up to 70% of errors using a basic SED parity code (3.125% storage overhead). In contrast, a full Hamming SEC code incurs a storage overhead of 18.75%. For SDELC, one could design more sophisticated recovery policies using stronger forms of SI, and use profiling methods to automatically annotate program regions that are likely to experience faults.

CHAPTER 5

Parity++: Lightweight Error Correction for Last Level Caches

As the size of on-chip SRAM caches is increasing rapidly and the physical dimension of the SRAM devices is decreasing, reliability of caches is becoming a growing concern. This is because with increased size of caches, the likelihood of radiation-induced soft faults also increases. As a result, information redundancy in the form of Error Correcting Codes (ECC) is becoming extremely important, especially to protect the larger sized last level caches (LLCs). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There is additional encoding (while writing data) and decoding (while reading data) procedures required as well. In caches, these additional area, power and latency overheads need to be minimized as much as possible. To address this problem, we present in this chapter Parity++: a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". This protection scheme provides Single Error Detection (SED) for all messages and Single Error Correction (SEC) for a subset of messages. Thus, it is stronger than just a basic SED parity and has $\sim$9% lower storage overhead and much lower error detection energy than a traditional Single Error Correcting, Double Error Detecting (SECDED) code. We also propose a memory speculation procedure that can be used with any ECC scheme to hide the decoding latency while reading messages when there are no errors.

Collaborators:

- Dr. Clayton Schoeny, UCLA/Square
- Prof. Lara Dolecek, UCLA
- Prof. Puneet Gupta, UCLA

5.1 Introduction

As demand and size of on-chip caches is increasing rapidly and the physical dimension and noise margins are decreasing, reliability of caches is increasingly becoming an important issue. As given in [116, 117], the vulnerability of SRAM caches to soft errors grows with increase in size. Also with reduction in physical dimensions of these devices, the critical charge required to flip the content of a cell due to a particle strike decreases. As a result, the soft error rate is higher for large capacity caches. The widely used technique to guarantee reliability of storage devices is using information redundancy in the form of Error Correcting Codes (ECC). In typical ECCs, extra redundancy bits are added to every row to detect and correct errors. There are additional encoding (while writing data) and decoding (while reading data) procedures required as well. Thus ECCs come with encoding and decoding mechanisms that incur additional overheads in terms of latency and energy. Both these overheads are critical for caches and hence, ECC protection was not widely used in caches till recently. However, due to the increased reliability concerns of large capacity caches and processor performance degradation due to occurrence of errors, cache protection using ECC schemes is becoming increasingly popular. Nevertheless, these additional area, power and latency overheads need to be minimized in caches as much as possible.

In this chapter, we present Parity++: a novel unequal message protection scheme for last level caches that preferentially provides stronger error protection to certain "special messages". As the name suggests, this coding scheme requires one extra bit above a simple parity Single Error Detection (SED) code while providing SED for all messages and Single Error Correction (SEC) for a subset of messages. Thus, it is stronger than just basic SED parity and has much lower parity storage overhead (3.5X and 4X lower for 32-bit and 64-bit memories respectively) than a traditional Single Error Correcting, Double Error Detecting (SECDED) code. Error detection circuitry often lies on the critical path and is generally more critical than error correction circuitry as error occurrences are rare even with an increasing soft error rate. Our coding scheme has a much simpler error detection circuitry that incurs lower energy and latency costs than the traditional SECDED code. Thus, Parity++ is a lightweight ECC code that is ideal for large capacity last level caches. We also evaluate Parity++ with a memory speculation procedure [118] that can be generally

applied to any ECC protected cache to hide the decoding latency while reading messages when there are no errors.

5.2 Background and Related Work

5.2.1 Error Correcting Codes

Error-correcting codes (ECCs) increase the resiliency of communication and storage systems by adding redundant bits (or symbols, but in this work we focus on the binary regime). A code $\mathscr{C}$ can be thought of as an injective mapping of *messages* of length k to *codewords* of length n. Let r be the number of redundant bits, i.e., $r = n - k$. A binary code is considered *linear* if the sum of any two codewords in $\mathscr{C}$ is also a codeword in $\mathscr{C}$.

A linear block code is described by either its $(k \times n)$ *generator matrix* $\mathbf{G}$ or its $(r \times n)$ *parity-check matrix* $\mathbf{H}$, with the relation $\mathbf{G}\mathbf{H}^T = \mathbf{0}$. A particular message $\mathbf{m}$ is encoded to its corresponding codeword $\mathbf{c}$ by multiplying it with the generator matrix as follows: $\mathbf{m}\mathbf{G} = \mathbf{c}$. Each row of $\mathbf{H}$ is a parity-check equation that all codewords must suffice, thus $\mathbf{H}\mathbf{c}^{\mathrm{T}} = \mathbf{0}$. We define the *received vector* at the output of the channel as $\mathbf{y} = \mathbf{c} + \mathbf{e}$, in which $\mathbf{e}$ is the error-vector representing which bits have been flipped. The receiver calculates the *syndrome*, $\mathbf{s} = \mathbf{H}\mathbf{y}^{\mathrm{T}}$, and if $\mathbf{s} \neq \mathbf{0}$, then it is known that the received vector is not a valid codeword. At this point, the decoder can either attempt to determine the most likely originally transmitted codeword or it can simply raise a flag that an error was detected (depending on the system goals and design). We say a code is *systematic* if a message is directly embedded in the codeword, i.e., each message bit is equal to a specific codeword bit.

A useful parameter of a linear code is its *minimum distance*, d_{min}, which is the minimum Hamming distance between any two (non-identical) codewords. Additionally, since a linear code must include the $\mathbf{0}$ codeword, the minimum distance of a linear code is simply the minimum weight of any (non-zero) codeword in the code:

$$d_{min} = \min_{\substack{\mathbf{c_1},\mathbf{c_2} \in \mathscr{C}; \\ \mathbf{c_1} \neq \mathbf{c_2}}} [d_H(\mathbf{c_1}, \mathbf{c_2})] = \min_{\substack{\mathbf{c} \in \mathscr{C}; \\ \mathbf{c} \neq \mathbf{0}}} [wt(\mathbf{c})].$$

A linear code guarantees correction of up to $t = \lfloor \frac{1}{2}(d_{min} - 1) \rfloor$ bit-errors, or detection of up to

$(d_{min}-1)$ bit-errors (without any correction guarantees). For even values of d_{min}, a linear code simultaneously guarantees correction of up to t bit-errors and detection of up to $(t+1)$ bit-errors. Further explanation of the fundamental properties of codes can be found in classic textbooks [119, 120].

5.2.2 SRAM Reliability and Error Detection and Correction in Caches

As mentioned before, SRAM reliability concerns are growing. Although the soft error rate of SRAM cell has almost been constant at 10^{-3} FIT/bit [121, 122], the likelihood of a particle striking the array is increasing with increase in size. Most of the recent processors with large capacity caches have ECC protected L2 and/or L3 caches. Some of the common and recent examples include Qualcomm's Centriq 2400 processor [123], AMD's Athlon [124] and Opteron [125] processors as well as IBM Power 4 [126] processors. Most of the commercially available processors use traditional (72,64) SECDED [127] code on each 64-bit word in the cache line. A lot of past works have suggested decoupling error detection and correction mechanisms so as to reduce the complexity and overhead of error detection since that is more critical than error correction. In [128], the authors suggest using SRAM for only error detection and storing the ECC correction bits within the memory hierarchy to reduce the overhead. In another work on ECC in caches, the authors of [129] suggest protecting only those cache lines that have been recently used. Thus, they trade-off protection with area and energy. Some past works like [130] have also focused on ECC protection schemes for L1 cache.

5.2.3 Application Characteristics

Data or instructions in applications are generally very structured. Frequencies of instructions in most applications follow power law distribution [131]. This means that some instructions get more frequently accessed than the rest. If the opcode (that primarily determines the action taken by the instruction) in a certain instruction set architecture (ISA) is, for example, the first x bits, then the relative frequency of the opcodes of the common instructions are high. This means most instructions in the memory would have the same prefix of x-bits. Table 5.1 shows the fraction of

the two most frequently occurring opcode over each of the benchmark suites. The benchmarks were compiled for 32-bit RISC-V (RV32G) [52] instruction set v2.0 were the least significant 7 bits are designated as the opcode. This is true not just for instructions but also for data. In most applications, the data in the memory is usually low-magnitude signed data of a certain data type. However, these values get represented inefficiently, for e.g., 4-byte integer type used to represent values that usually need only 1-byte. Thus, in most cases, the MSBs would be a leading-pad of 0s or 1s. Table 5.1 shows that, for a wide range of data sets, most stored data starts with a leading pad of zeros. Our approach of utilizing these characteristics in applications complements recent research on data compression in cache and main memory systems such as frequent value/pattern compression [132, 133], base-delta-immediate compression [134] and bit-plane compression [135]. However, our main goal is to provide stronger error protection to these special messages that are chosen based on the knowledge of data patterns in context.

Table 5.1: Fraction of Special Messages per Benchmark Within Suite

	Top Two Most Freq Opcodes (Data Memory)		**First 6 bits are 0 (Instruction Memory)**	
Benchmark Suite	**Max**	**Mean**	**Max**	**Mean**
AxBench	0.51	0.46	0.92	0.86
SPEC CPU2006	0.56	0.37	0.99	0.89

5.3 Lightweight Error Correction Code

5.3.1 Theory

The code we developed in this work, which we call Parity++, is a type of *unequal message protection* code, in that we *a priori* designate specific messages to have extra protection against errors as can be seen in Figure 5.1. As in [136], there are two classes of messages, normal (non-special) and special, and they are mapped to normal (or non-special) and special codewords, respectively. When dealing with the importance or frequency of the underlying data, we refer to the messages; when discussing error detection/correction capabilities we refer to the codewords.

Codewords in Parity++ have the following error protection guarantees: normal codewords

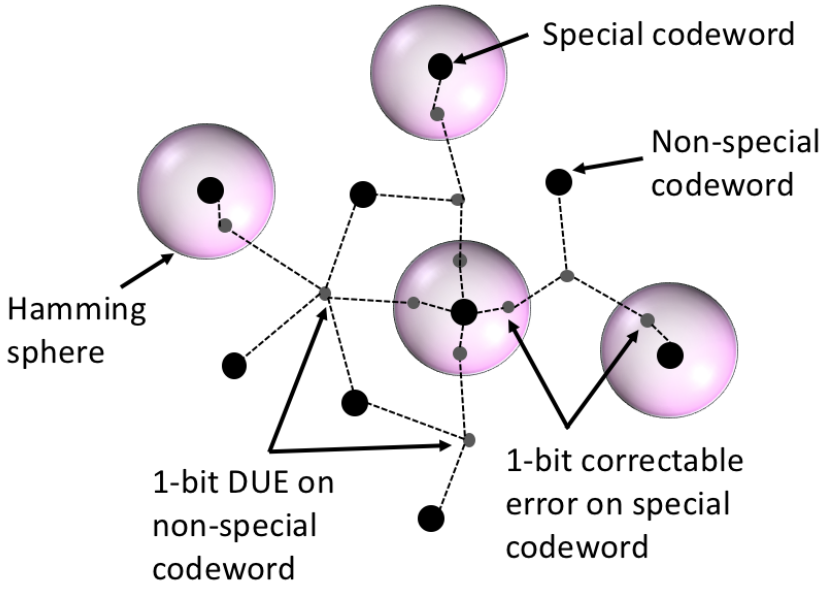


Figure 5.1: Conceptual Illustration of Parity++ for 1-bit error

have single-error detection; special codewords have single-error correction. Let us partition the codewords in our code $\mathscr{C}$ into two sets, $\mathscr{N}$ and $\mathscr{S}$, representing the normal and special codewords, respectively. The minimum distance properties necessary for the aforementioned error protection guarantees of Parity++ are as follows:

$$\min_{\mathbf{u},\mathbf{v}\in\mathscr{N},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 2, \tag{5.1}$$

$$\min_{\mathbf{u}\in\mathscr{N},\mathbf{v}\in\mathscr{S}} d_H(\mathbf{u},\mathbf{v}) \geq 3, \tag{5.2}$$

$$\min_{\mathbf{u},\mathbf{v}\in\mathscr{S},\mathbf{u}\neq\mathbf{v}} d_H(\mathbf{u},\mathbf{v}) \geq 3. \tag{5.3}$$

A second defining characteristic of the Parity++ code, is that the length of a codeword is only two bits longer than a message, i.e., $n = k+2$. Comprehensive comparisons between Parity++ and other popular ECCs are included in some of the subsequent sections.

For the context of this chapter, let us assume that our Parity++ always has message length k as a power of 2. The overall approach to constructing our code is to create a Hamming subcode of a SED code [137]; when an error is detected, we decode to the neighboring special codeword. The overall code has $d_{min} = 2$, but a block in $\mathbf{G}$, corresponding to the special messages, has $d_{min} \geq 3$. For the sake of notational convenience, we will go through the steps of constructing the $(34,32)$

Parity++ code (as opposed to the generic $(k+2,k)$ Parity++ code).

We begin by creating the generating matrix for the Hamming code whose message length is at least as large as the message length in the desired Parity++ code; in our case, we use the $(63,57)$ Hamming code. Let α be a primitive element of GF(2^6) such that $1+x+x^6=0$, then our generator polynomial is simply $g_S(x)=1+x+x^6$ (and we construct our generator matrix using the usual polynomial coding methods). We then shorten this code to $(32,26)$ by expurgating and puncturing (i.e., deleting) the right and bottom 31 columns and rows. Now, we add a column of 1s to the end, resulting in a generator matrix, which we denote as $\mathbf{G}_S$, for a $(33,26)$ code with $d_{min}=4$.

For the next step in the construction of the generating matrix of our $(34,32)$ Parity++ code, we add $\mathbf{G_N}$ on top of $\mathbf{G}_S$, where $\mathbf{G_N}$ is the first 6 rows of the generator matrix using the generator polynomial $g_N(x)=1+x$, with an appended row of 0s at the end. Note that $\mathbf{G_N}$ is the generator polynomial of a simple parity-check code. By using this polynomial subcode construction, we have built a generator matrix with overall $d_{min}=2$, with the submatrix $\mathbf{G}_S$ having $d_{min}=4$. At this point, notice that messages that begin with 6 0s only interact with $\mathbf{G}_S$; these messages will be our special messages. Note that Conditions 5.1 and 5.3 are satisfied; however, Condition 5.2 is not satisfied. To meet the requirement, we add a single non-linear parity-bit that is a NOR of the bits corresponding to $\mathbf{G_N}$, in our case, the first 6 bits.

The final step is to convert $\mathbf{G}_S$ to systematic form via elementary row operations. Note that these row operations preserve all 3 of the required minimum distance properties of Parity++. As a result, the special codewords (with the exception of the known prefix) are in systematic form. For example, in our $(34,32)$ Parity++ code, the first 26 bits of a special codeword are simply the 26 bits in the message (not including the leading run of 6 0s).

At the encoding stage of the process, when the message is multiplied by $\mathbf{G}$, the messages denoted as special must begin with a leading run of $\log_2(k)+1$ 0's. However, the original messages we deem to be special do not have to follow this pattern as we can simply apply a pre-mapping before the encoding step, and a post-mapping after the decoding step.

In our $(34,32)$ Parity++ code, observe that there are 2^{26} special messages. Generalizing, it is easy to see that for a $(k+2,k)$ Parity++ code, there are $2^{k-\log_2(k)-1}$ special messages.

5.3.2 Error Detection and Correction

We separate the received–possibly erroneous–vector $\mathbf{y}$ into two parts, $\bar{\mathbf{c}}$ and η, with $\bar{\mathbf{c}}$ being the first $k+1$ bits of the codeword and η the additional nonlinear redundancy bit ($\eta = 0$ for special messages and $\eta = 1$ for normal messages). There are three possible scenarios at the decoder: no (detectable) error, correctable error, or detected but uncorrectable error.

First, due to the Parity++ construction, every valid codeword has even weight. Thus, if $\bar{\mathbf{c}}$ has even weight, then the decoder concludes no error has occurred, i.e., $\bar{\mathbf{c}}$ was the original codeword. Second, if $\bar{\mathbf{c}}$ has odd weight and $\eta = 0$, the decoder attempts to correct the error. Since $\mathbf{G}_S$ is in systematic form, we can easily retrieve $\mathbf{H}_S$, its corresponding parity-check matrix. The decoder calculates the syndrome $\mathbf{s_1} = \mathbf{H}_S^T \bar{\mathbf{c}}$. If $\mathbf{s_1}$ is equal to a column in $\mathbf{H}_S$, then that corresponding bit in $\bar{\mathbf{c}}$ is flipped. Third, if $\bar{\mathbf{c}}$ has odd weight and either $\mathbf{s_1}$ does not correspond to any column in $\mathbf{H}_S$ or $\eta = 1$, then the decoder declares a DUE (detected but un-correctable error).

The decoding process described above guarantees that any single-bit error in a special codeword will be corrected, and any single-bit error in a normal codeword will be detected (even if the bit in error is η).

Let's take a look at two concrete examples for the $(10,8)$ Parity++ code. Without any premapping, a special message begins with $\log_2(3)+1 = 4$ zeros. Let our original message be $\mathbf{m} = (00001011)$, which is encoded to $\mathbf{c} = (1011010110)$. Note that the first 4 bits of $\mathbf{c}$ is the systematic part of the special codeword. After passing through the channel, let the received vector be $\mathbf{y} = (1001010110)$, divided into $\bar{\mathbf{c}} = (1001010110)$ and $\eta = 0$. Since the weight of c is odd and $\eta = 0$, the decoder attempts to correct the error. The syndrome is equal to the 3rd column in $\mathbf{H}_S$, thus the decoder correctly flips the 3rd bit of $\bar{\mathbf{c}}$.

For the second example, let us begin with $\mathbf{m} = (11010011)$, which is encoded to (0011111101). After passing through the channel, the received vector is $\mathbf{y} = (0011011101)$. Since the weight of $\bar{\mathbf{c}}$ is odd and $\eta = 1$, the decoder declares a DUE. Note that for both normal and special codewords, if the only bit in error is η itself, then it is implicitly corrected since $\bar{\mathbf{c}}$ has even weight and will be correctly mapped back to $\mathbf{m}$ without any error detection or correction required.

5.3.3 Architecture

For a cache with error detection and correction (EDAC) mechanism, there is additional error detection/correction latency. Error detection latency is more critical than error correction as occurrence of an error is a rare event when compared to the processor cycle time and doesn't fall in the critical path. The data/instruction being read from the cache goes through the ECC error detection engine first. If there are no errors then the decoded message moves ahead. In case of an error, the received message goes through an additional correction engine to retrieve the correct message and then the message can be used in the rest of the computation flow.

When using Parity++, the flow almost remains the same. Parity++ can detect all single bit errors but has correction capability for "special messages". When a single bit flip occurs on a message, the error detection engine first detects the error and stalls the pipeline. If the non-linear bit says it is a "special message"(non-linear bit is '0'), the received message goes through the Parity++ error correction engine which outputs the corrected message. This marks the completion of the cache access. If the non-linear bit says it is a non-special message (non-linear bit is '1'), then a DUE is declared and it is checked if the cache line is clean. If so, the cache line is simply read back from the lower level cache or the memory and the cache access is completed. However, if the cache line is dirty and there are no other copies of that particular cache line, it leads to a crash or a roll back to checkpoint. Note that both Parity++ and SECDED have equal decoding latency of one cycle that is incurred during every read operation from an ECC protected cache. The encoding latency during write operation does not fall in the critical path and hence, is not considered in our analyses.

Next in this chapter we present a memory speculation scheme that helps to hide the latency incurred by the error detection engine when there are no errors.

5.3.3.1 Memory Speculation

Figure 5.2 shows the flow of a read operation when the memory speculation scheme is used. The basic idea behind this speculation scheme is to predict the original message from the encoded codeword without having to go through the decoding/error detection circuitry in order to hide the additional latency incurred by the decoding/detection mechanism. While the decoding happens,

the predicted instruction/data can move forward to the next stages in the pipeline. If the predicted value is correct, then no action is required and pipeline goes ahead as usual without any additional stalls. In case an error is detected, the mis-predicted instruction or all the dependent instructions that received the mis-predicted data needs to be squashed. This prediction scheme for ECC protected caches is similar to what was proposed in [118] for stronger error protection in on-chip memories.

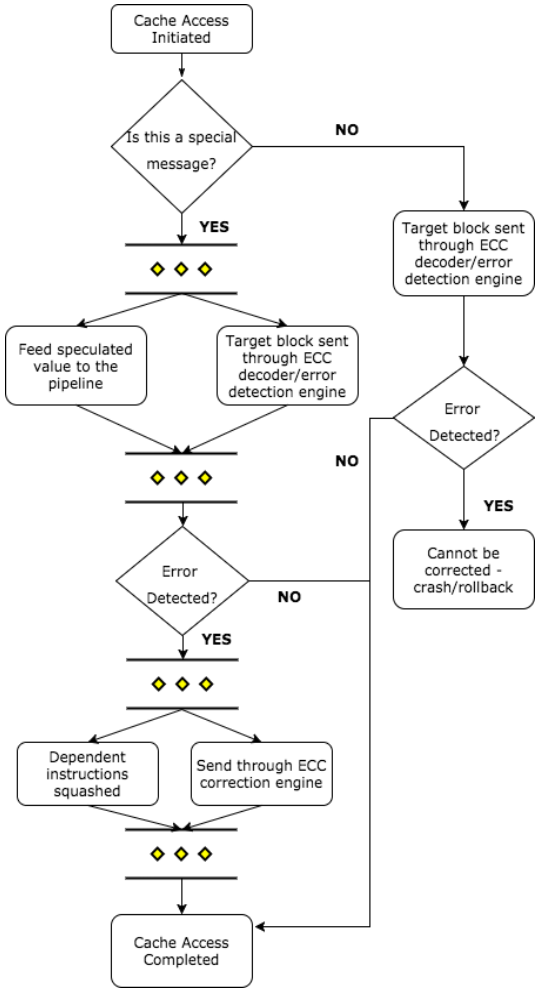


Figure 5.2: Flow of read operation in cache with memory speculation and Parity++ protection schemes

This speculation scheme is most effective when the encoded ECC codewords are systematic. When systematic, the original message can be easily retrieved by truncating the additional redundant bits that are generally added to the end of the actual message in case of no errors in the received codeword. Instead of waiting for the decoding to get done, the original message can be speculated by

truncating the redundant bits. Thus, the computation moves ahead with the predicted data/instruction without any stalls while the decoding for error detection happens in parallel. A major difference between SECDED and our scheme, Parity++ is that all codewords under SECDED are systematic while only the special messages for Parity++ are systematic. As a result, for Parity++, speculation is used only if the message is special. If not, computation is stalled for one cycle while decoding/error detection happens. Special messages can be distinguished from non-special messages using the non-linear bit.

5.3.3.2 Additional Cache Support for Speculation

Figure 5.3 depicts the additional circuitry that needs to be added to a traditional cache to support the memory speculation scheme with Parity++.

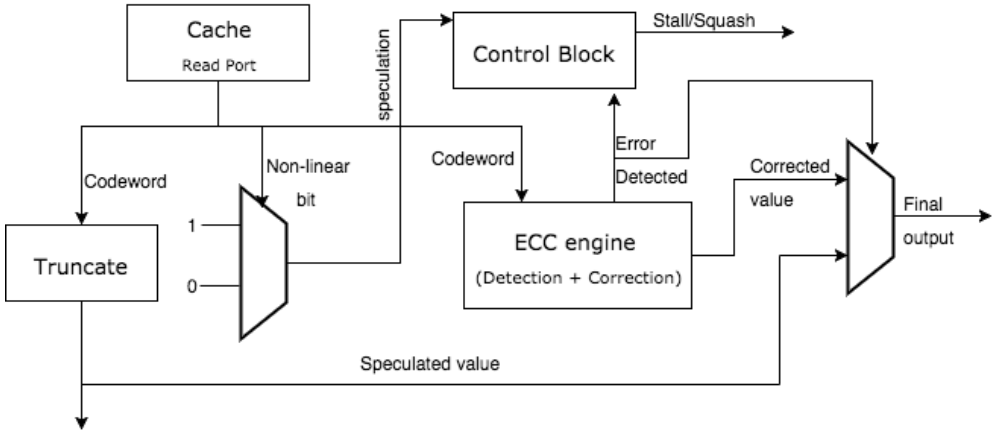


Figure 5.3: Cache architecture to implement Parity++ with memory speculation

The non linear bit is first checked. If it is a special message, then speculation is triggered and the speculated value is forwarded to the next stage. This speculated value comprises of the lower 26-bits of the received codeword to which the special prefix is separately appended. Meanwhile, the decoding and the error detection circuitry works in parallel. If an error is detected, the control module initiates a squash operation to squash all the dependant instructions that used the mis-predicted data and the ECC correction engine provides the correct output. The control module also stalls the pipeline when the non linear bit indicates that the message is not special and hence, the codeword is not systematic. Therefore, speculation cannot be used and the pipeline needs to be

stalled for one cycle till the original message is decoded. The stall latency is, of course, greater than one cycle when an error is detected and the ECC correction engine needs to be triggered. This additional control module is simple and has minimal overhead in terms of area and energy.

5.3.4 Coverage and Overheads

5.3.4.1 Detection/Correction Coverage

As given in Table 5.2 single-bit parity detects any single-bit error. Our Parity++ scheme keeps this single-bit error detection guarantee, and additionally provides single-bit error correction for special messages. Also, any 2-bit error on a special message in our Parity++ scheme is guaranteed detectable.

The coverage of SECDED and DECTED codes can be understood from their names. SECDED codes guarantee correction of any single bit error and detection of any double bit error; DECTED codes guarantee correction of any double bit error and detection of any triple bit error.

Table 5.2: Error Detection and Correction Coverage for Parity++ along with some widely used ECC schemes

ECC scheme	Error Bits Detected	Error Bits Corrected
Single Error Detecting (SED)	1	0
Parity++	Special Messages - 2 Non-Special Messages - 1	Special Messages - 1 Non-Special Messages - 0
SECDED	2	1
DECTED	3	2

5.3.4.2 Storage Overhead

Single-error detection requires only a single parity bit; our Pairty++ scheme adds an additional parity-bit for a total of 2. The most efficient SEC code is the Hamming code. Assuming our message length, k, is a power of 2, then the number of redundancy bits required for the (shortened) Hamming

code is $\log(k)+1$. Since the Hamming code has a minimum distance of 3, we can create a SECDED code—the extended Hamming code—with the addition of a single parity bit, yielding a total of $\log(k)+2$ redundancy bits. Similarly, we can use a (shortened) extended BCH code as a DECTED code, with $2\log(k)+3$ redundancy bits. The parity storage overhead of these schemes for different cacheline sizes is given in Figure 5.4

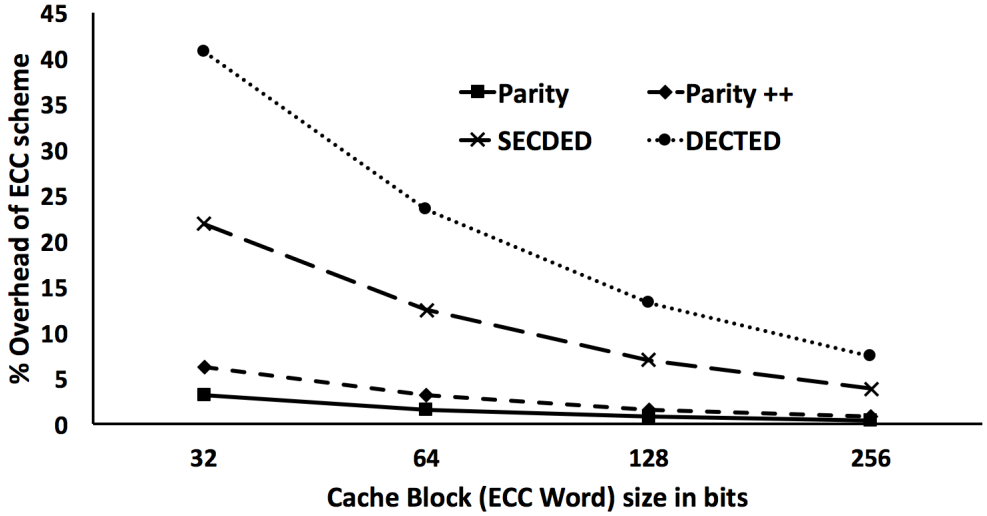


Figure 5.4: Storage overhead of different commonly used ECC schemes along with our scheme Parity++

5.3.4.3 Latency and Energy Overhead

The encoding and decoding latencies when writing to/reading from the memory are almost identical for Parity++ and SECDED. They would both require an additional one cycle for each of the two operations. Error correction in case of Parity++ requires an extra matrix multiplication. However, this latency is not critical as occurrence of errors is a rare event compared to the cycle time of the processor. With the proposed memory speculation scheme, SECDED incurs no additional decoding latency when there are no errors. For Parity++ the one cycle extra decoding latency happens only when it is a non special message (only 20-25% of messages are typically non-special).

The encoding energy overhead is almost similar for both Parity++ and SECDED. The decoding energy overheads are slightly different. For SECDED, the original message can be retrieved from the received codeword by simply truncating the additional ECC redundant bits. However, all received

codewords need to be multiplied with the H-matrix to detect if any errors have occurred. For Parity++, the original message can be retrieved using truncation when it is a special messages. For the 20-25% non special messages, the non-systematic received codeword needs to be multiplied with a decoder matrix to get the original message. This decoder matrix multiplication, when sythesized using an industrial 45nm library has $\sim$4x higher energy overhead than the H-matrix multiplication of SECDED since the Parity++ decoder is larger than the SECDED H-matrix. However, for Parity++, the error detection scheme is much simpler. It is just a chain of XOR gates and the synthesized detection engine consumes $\sim$10x lower energy than the H-matrix of SECDED required for error detection. For Parity++, all messages go through the chain of XOR gates for error detection and only the non special messages need to be multiplied with the decoder matrix to retrieve the original message. Since the error detection in Parity++ is much cheaper in terms of energy overhead than SECDED and the non special messages only constitute about 20-25% of the total messages, the overall read energy in Parity++ turns out to be much lesser than SECDED. Also, with reduced array size for caches with Parity++ due to lower storage overhead, the leakage energy is also less than that in caches with SECDED.

5.4 Experimental Methodology

We evaluated Parity++ over applications from the SPEC 2006 benchmark suite. Two sets of core micro-architectural parameters (provided in Table 5.3) were chosen to understand the performance benefits in both a lightweight in-order(InO) processor and a larger out-of-order(OoO) core. Performance simulations were run using Gem5 [138], fast forwarding for 1 billion instructions and executing for 2 billion instructions.

The first processor is a lightweight single in-order core architecture with a 32kB L1 cache for instruction and 64kB L1 cache for data. Both the instruction and data caches are 4-way associative. The LLC is a unified 1MB L2 cache which is also 8-way associative. The second processor is a dual core out-of-order architecture. The L1 instruction and data caches have the same configuration as the previous processor. The LLC comprises of both L2 and L3 caches. The L2 is a shared 512kB SRAM based cache while the L3 is a shared 2MB cache which is 16-way associative. For both the

baseline processors it is assumed that the LLCs (L2 for the InO processor and L2 and L3 for the OoO processor) have SECDED ECC protection.

The performance evaluation was done only for cases where there are no errors. Thus, latency due to error detection is taken into consideration but not error correction as correction is rare when compared to the processor cycle time and doesn't fall in the critical path. In order to compare the performance of the systems with Parity++ against the baseline cases with SECDED ECC protection, the size of the LLCs were increased by $\sim$10% due to the lower storage overhead of Parity++ as provided in Section 5.3.4. We call this iso-area since the additional area coming from reduction in redundancy is used to increase the total capacity of the SRAM. The iso-area evaluation was done for both with and without memory speculation. The analysis was also done for the iso-capacity where the memory capacity of the systems with Parity++ and SECDED remain same and their performances are measured. As mentioned before, SECDED allows speculation in all cases and thus, incurs no additional read latency due to error detection when there is no error. But for Parity++, only the special messages are systematic and thus, for all non-special messages, there is an additional one cycle read latency due to the error detection circuitry. This additional latency for non-special messages was also taken into consideration for our simulations.

5.5 Results and Discussion

In this section we discuss the performance results obtained from the Gem5 simulations (as mentioned in Section 5.4). Figures 5.5 and 5.6 show the comparative results for the two different sets of core micro-architectures across a variety of benchmarks from the SPEC2006 suite when using memory speculation. In both the evaluations, performance of the system with Parity++ was compared against that with SECDED. The evaluation was further split into iso-area and iso-capacity as explained in Section 5.4.

For both the core configurations, the observations for the iso-area case are almost similar. With memory speculation it is seen that with additional memory capacity for iso-area, the system with Parity++ has upto $\sim$4% better performance (lower execution time) than the one with SECDED. This improvement in performance happens in spite of the additional one cycle latency incurred

Table 5.3: Core Micro-architectural Parameters

	Processor-I	**Processor-II**
Cores	1, InO (@ 2GHz)	2, OoO (@ 2GHz)
L1 Cache per core	32KB I$ 64KB D$ 4-way	32KB I$ 64kB D$ 4-way
L2 Cache	1MB (unified) 8-way	512KB (shared, unified) 8-way
L3 Cache	-	2MB (shared) 16-way
Cache Line Size	64B	64B
Memory Configuration	4GB of 2133MHz DDR3	8GB of 2133MHz DDR3
Nominal Voltage	1V	1V

on non special messages in the case of Parity++. The applications showing higher performance benefits are mostly memory intensive. Hence, additional cache capacity with Parity++ reduces overall miss rate to an extent such that the slight increase in average LLC hit time gets offset. For most of these applications, this performance gap widens as the LLC size increases for Processor-II. The applications showing roughly similar performances on both the systems are the ones which already have a considerably lower LLC miss rate. As a result, increase in LLC capacity due to Parity++ doesn't lead to a significant improvement in performance. The same evaluation was also done for the case where there is no memory speculation, i.e., both Parity++ and SECDED protected caches have additional hit latency of one cycle for all read operations. The results show that with the exact same hit latency, Parity++ has upto 7% lower execution time than SECDED due to additional memory capacity.

A more significant result is the iso-capacity case with memory speculation. It is seen that even with additional one cycle latency for non special messages in Parity++, the performance of the system with Parity++ is at par with that of SECDED. This means that by using our lightweight error

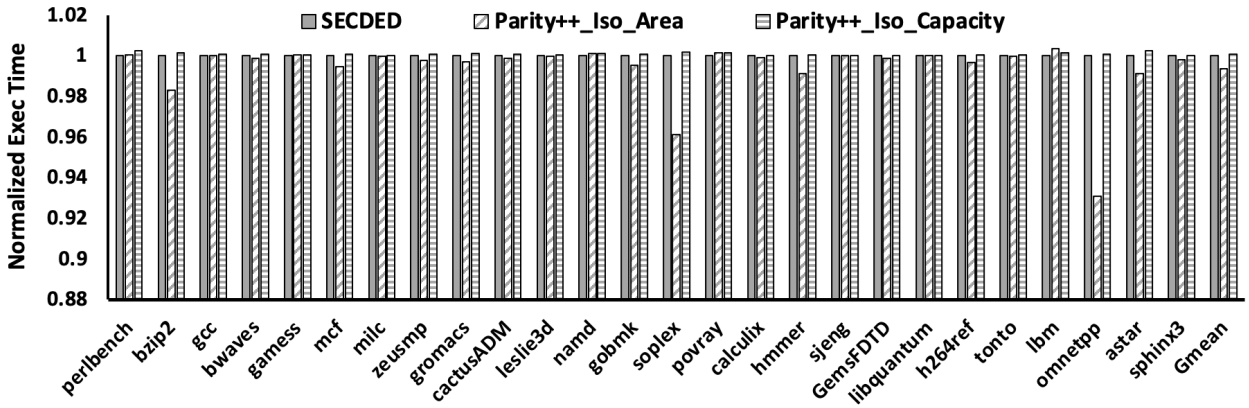


Figure 5.5: Comparing Normalized Execution Time of Processor-I with SECDED and Parity++ (with memory speculation)

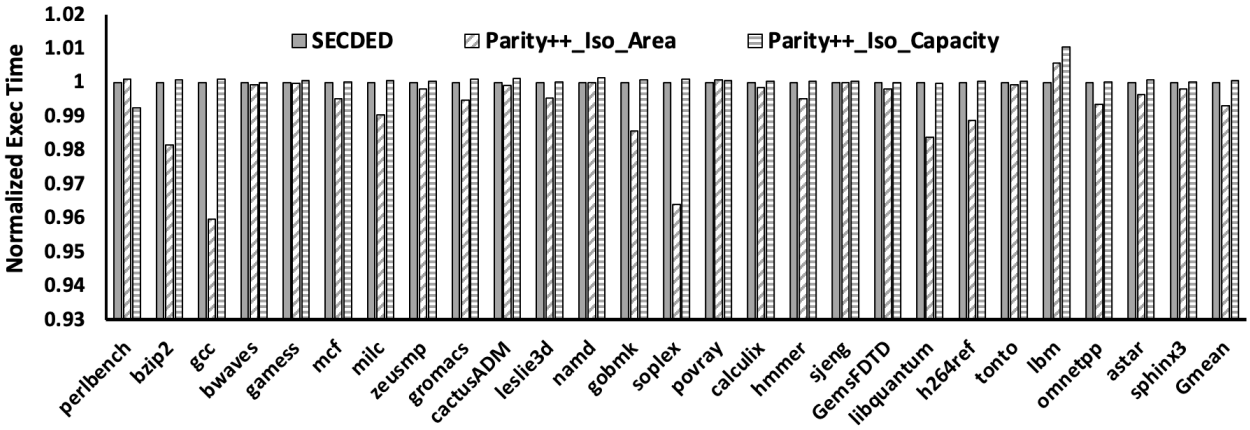


Figure 5.6: Comparing Normalized Execution Time of Processor-II with SECDED and Parity++ (with memory speculation)

correction scheme, we manage to save about 5-9% last level cache area (excluding decoder and peripheral circuit area) with negligible hit in performance. Since the LLCs constitute more than 30% of the processor chip area, the cache area savings translate to a considerable amount of reduction in the chip size. This additional area benefit can either be utilized to make an overall smaller sized chip or it can be used to pack in more compute tiles to increase the overall performance of the system.

Parity++ in Lightweight Approximation-Friendly Embedded Memory

Instead of limiting ourselves to last level on-chip caches, we extended the evaluation to on-chip memories in embedded devices. Embedded systems at the edge of the Internet-of-Things (IoT) is driven by the need for low cost and low energy consumption. On-chip memories in these lightweight embedded systems consume a significant portion of system energy. As a result, having strong error correction schemes like SECDED or ChipKill [104] is too costly, in terms of overheads, for such devices. Based on the iso-capacity results, Parity++ (with 3.5X lower parity storage overhead than SECDED in a 32-bit memory) seems to be a good fit for SRAM based embedded memories. Since Parity++ helps in reducing area (in turn reducing SRAM leakage energy) and also has lower error detection energy, it provides a better protection mechanism in such devices than SECDED. It is also stronger than a single-error detecting (SED) Parity code and hence can reduce the number of crashes/hangs when there is a single bit flip in the memory.

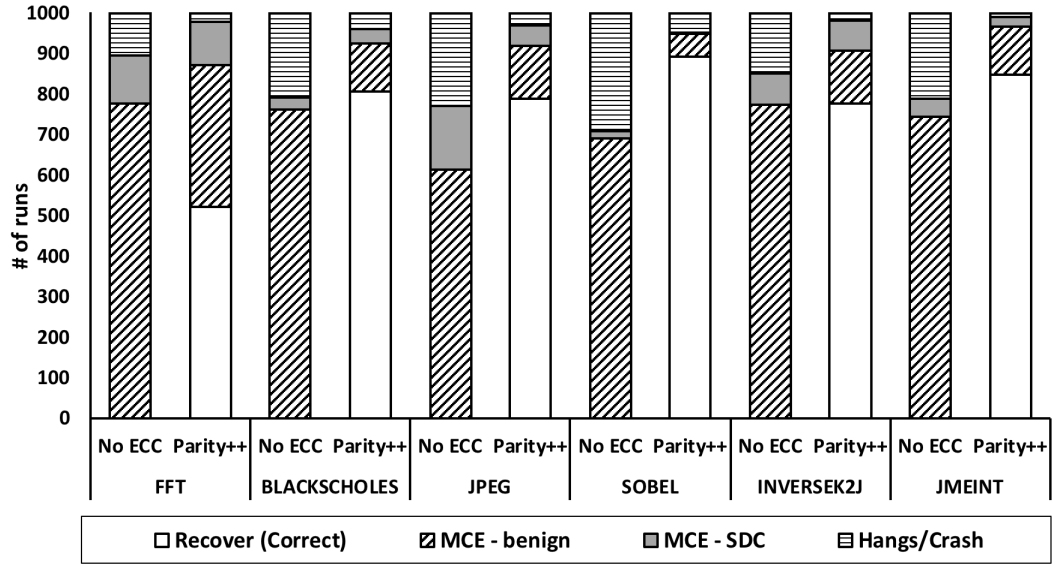


Figure 5.7: Output quality of AxBench benchmarks for memory with no ECC vs with Parity++

Most of the applications that run on these low-cost IoT devices are approximation-tolerant. Hence, we analyzed the benefits of using Parity++ in such devices on 6 applications from AxBench [51], an approximate benchmark suite. The AxBench benchmarks were compiled for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [52] using the official tools [139]. Each benchmark was ran till completion 1000 times on top of the RISC-V proxy kernel [140] using the Spike simula-

tor [114] that was modified to produce representative memory access traces. For each run, a single bit error was randomly injected on a demand data memory read. We compared Parity++ against the case when there is no ECC protection and the program continues with erroneous message. In case of non-special messages in Parity++, even though a single bit flip is detected, the program continued with the wrong message instead of crashing immediately since these applications are approximation-tolerant. The results are shown in Figure 5.7. It can be seen that Parity++ reduces intolerable Silent Data Corruption (SDC), that is, an SDC with more than 10% output error, by upto 84.2%(avg. 32.5%). It significantly reduces the number of crashes/hangs by upto 95.3%(avg. 85.6%). This means Parity++ not only improves the quality of output, the system will be much more resilient to hangs/crashes in case of unpredictable single bit flips during runtime.

5.6 Conclusion

In this work, we present a novel lightweight error protection scheme, Parity++, for last level caches based on unequal message protection. From our analysis, we find that about 80% of messages/words have same prefix bits (leading 0's) and we denote these as special messages. For a 64 bit word, Parity++ uses only 2 additional redundant bits and provides SECDED protection for these special messages while providing only SED for the non-special messages. In iso-area evaluations, up to about 4% performance benefit can be obtained, while iso-capacity evaluations showed almost negligible ($<$0.2% in all but one case) performance degradation with $\sim$9% lower storage overhead than a traditional SECDED scheme which translates to about 5% cache area savings.

CHAPTER 6

COMET: On-die and In-controller Collaborative Memory ECC Technique for Stronger and Safer Correction of DRAM Errors

DRAM manufacturers have started adopting on-die error correcting coding (ECC) to deal with increasing error rates. The typical single error correcting (SEC) ECC on the memory die is coupled with a single-error correcting, double-error detecting (SECDED) ECC in the memory controller. Unfortunately, the on-die SEC can miscorrect double-bit errors (which would have been safely detected but uncorrected errors in conventional in-controller SECDED) resulting in triple bit errors more than 45% of the time which are then undetectable or miscorrected in the memory controller >55% of the time resulting in silent data corruption. We introduce COllaborative Memory ECC Technique (COMET), a novel method to efficiently design either the on-die or the in-controller ECC code, that, for the first time, will ensure complete protection from silent data corruption when a double-bit error happens within the DRAM. Further, we propose a collaboration mechanism between the on-die and in-controller ECC decoders that corrects most of the double-bit errors without adding any additional redundancy bits to either of the two codes. Overall, COMET can eliminate all double-bit error induced silent data corruptions and correct virtually all (99.9997%) double-bit errors with negligible area, power, and performance impact.

Collaborators:

- Prof. Puneet Gupta, UCLA

6.1 Introduction

With increasing rate of scaling induced errors [24–28], the traditional method of row/column sparing used by DRAM vendors to tolerate manufacturing faults [29] has started to incur large overheads. To improve yields and provide protection against single-bit failures in the DRAM array at advanced technology nodes, memory manufacturers have started incorporating on-die error correction coding (on-die ECC) that helps to correct single-bit errors [24, 26]. The ECC encoding/decoding happens within the DRAM chip. The parity bits are stored in redundant storage on-chip and are not sent out of the chip; only the actual data, post correction, is sent out of the DRAM, making on-die ECC transparent to the outside world. Though DRAM manufacturers do not usually reveal their on-die ECC design and implementation, prior works [25, 26, 141–143] and industry whitepapers [24] indicate the most commonly used scheme is (136,128) Single Error Correcting (SEC) Hamming code [144] which corrects any single-bit error that occurs in 128 bits of actual data with the help of 8 bits of additional parity. On-die ECC is typically paired with rank-level single error correction, double error detection (SECDED) code in the memory controller. The main focus of in-controller ECC is to correct errors that are visible outside the memory chip, mostly due to failures in pins, sockets, buses, etc.

With the inclusion of on-die SEC, single-bit errors (SBE) get corrected within the DRAM chip. Though SBEs are still the most dominant failure mode in the DRAM arrays [24, 145], with increasing error rates [25, 26, 28], double-bit errors (DBE) within the array are no longer a rarity. However, a double error correcting (DEC) code incurs large overhead and is not practical for DRAM manufacturers to have an on-die DEC mechanism. In today's high reliability systems, the rank-level in-controller coding scheme is expected to detect DBEs and the system then restarts or rolls back to a checkpoint [146]. However, the on-die SEC code reduces the efficacy of in-controller DBE detection and significantly increases the chances of silent data corruption (SDC). As shown in Figure 6.1, without on-die SEC, the data goes through a single round of decoding inside the memory controller where the SECDED decoder flags the DBE. Now, with on-die SEC, the data goes through two rounds of decoding. The SEC decoder in the first round only ensures protection against SBEs. For DBEs, the decoder has a $>$45% (on average based on 10 random SEC constructions) chance

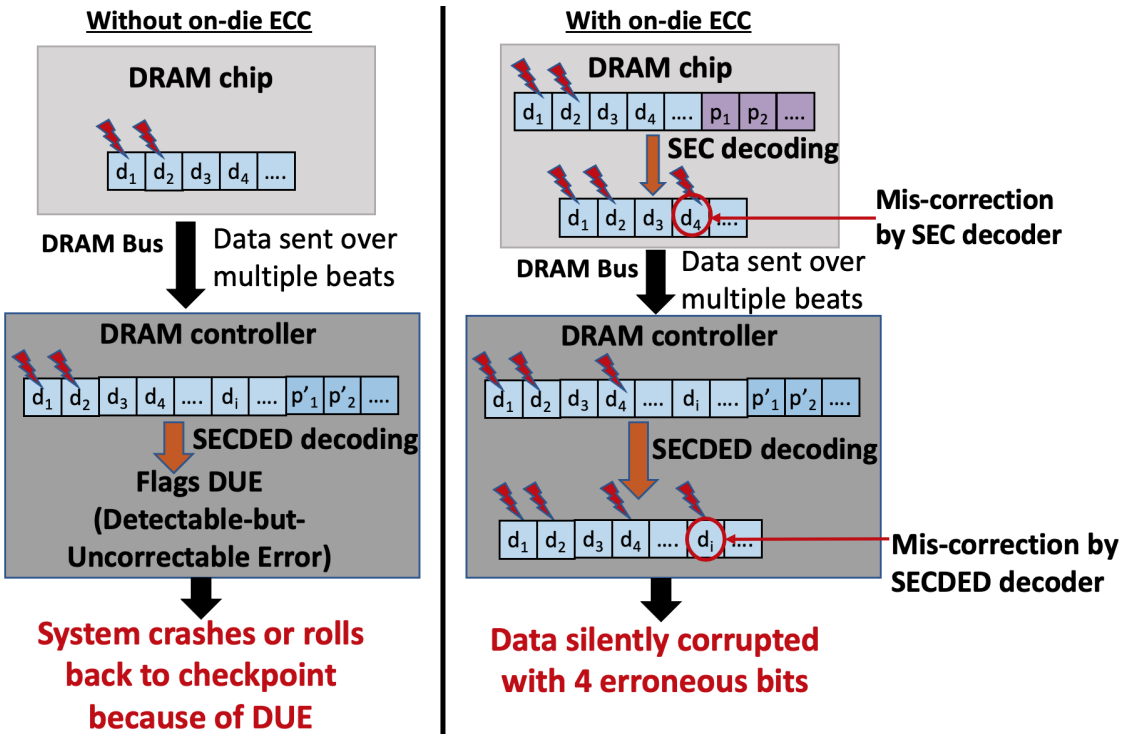


Figure 6.1: Example showing the difference when a DBE occurs in DRAMs with and without on-die SEC. Both systems have in-controller SECDED. Assumption: data and parity bits that get decoded in the controller in one cycle are sent from the same DRAM chip across multiple beats.

of miscorrection resulting in a triple-bit error. In the second round of decoding, the in-controller SECDED decoder has a ∼55% (on average, based on 10 random SECDED constructions) chance of considering the triple-bit error as a SBE and silently corrupts the data further. For a raw bit error rate of 10^{-4} that is often seen in recent works and industrial studies [25, 26, 28, 145, 147], we can expect SDC once every ∼ 300,000 SECDED decoding cycles (or 64-bit memory access) in a system with a single DRAM chip that has on-die (136,128) SEC and in-controller (72, 64) SECDED protection mechanisms. Thus, on-die ECC actually *worsens* memory reliability in the case of DBEs.

Furthermore, for every 128-bits of data, with on-die and in-controller ECC schemes combined, we now have 8 more bits of parity as compared to only in-controller ECC. These extra 8-bits for on-die SEC help only in the rare case when a single-bit fault outside the memory array (e.g. link/pin failure) coincides with a single-bit error in the chip. Other than that, the on-die SEC is not improving protection on top of what the in-controller code was already doing.

In this work, we propose a Collaborative Memory ECC Technique (COMET) that allows one to efficiently design the on-die and/or in-controller ECC that will not only correct single-bit errors but will also correct majority of double-bit errors and completely avoid silent data corruption with no additional parity bits. This chapter makes the following key contributions:

- For the first time (to the best of our knowledge), we provide a detailed on-die SEC code construction technique that completely eliminates DBE-induced SDCs at no additional overhead. The design technique exploits the overall memory system architecture and steers the miscorrected bit when a DBE occurs in such a way that the in-controller SECDED, irrespective of its actual implementation, never encounters all three bits of errors in the same decoding cycle, thereby guaranteeing no miscorrection.

- If the on-die SEC code does not guarantee protection from SDC in the case of DBEs, we show how the in-controller SECDED can be designed to take care of it. We provide a detailed construction of the SECDED code for a given on-die SEC implementation and memory system architecture.

- We develop a collaborative DBE correction technique. The SEC code needs to be designed with an additional constraint and the memory controller needs to send a special command with additional information once a DUE is flagged. This collaborative technique can correct almost all (99.9997%) double-bit errors in addition to ensuring no miscorrections.

- SEC-COMET implementations require no additional parity bits, have less than 5% decoder area and latency overheads and less than 10% power overhead as compared to the most efficient SEC construction. The COMET correction mechanism has negligible performance impact (less than 1.5% across multiple benchmarks) even for a high bit error rate of 10^{-4}.

6.2 Background

This section provides an overview of DRAM operation, coding theory background and types of ECC codes seen in today's DRAM based memory subsystems.

6.2.1 DRAM Operation

Dynamic Random Access Memory (DRAM) chip cell stores a single bit of data in a capacitor [148,149]. These cells are organized in two dimensional arrays called banks. A read/write command accesses a small subset of columns in a row and includes multiple steps. First the entire row is read into a row buffer using the ACTIVATE command. Then a READ/WRITE command is sent with the column address to initiate the data transfer. An xN DRAM chip uses N data pins (DQs) in parallel during data transfer [150, 151]. Typically, more than one DRAM chip is accessed in parallel to improve bandwidth and they together form a rank. A single memory access takes multiple cycles – during each cycle a *beat* of data (N bits from every chip in a rank) is transferred. The number of beats transferred in each access constitutes the memory burst length. The number of cycles per access and the width of a data beat accessed in each cycle depends on the memory device and the data access protocol. If a rank consists of 8 x8 DRAMs and the burst length is 8 beats, it translates to 64-bits of data transfer per beat and a total of 64B transfer per READ/WRITE command.

6.2.2 Linear Hamming Error Correcting Codes

Error correcting code (ECC) detects and/or corrects by adding redundant parity bits to the original data. A (n,k) Hamming code protects a k-bit dataword (original data) by encoding the data through linear transformation to form a n-bit codeword. The number of parity bits is equal to $n-k$. Increasing the number of parity bits increases the minimum Hamming distance between two legal n-bit codewords. A code of minimum distance d_{min} is guaranteed to correct is $t = \lfloor \frac{1}{2}(d_{min}-1) \rfloor$ erroneous symbols. The encoding is done by multiplying the dataword ($\vec{m}$) with the generator matrix $\mathbf{G}$: $\vec{m}\mathbf{G} = \vec{c}$ and the resulting codeword $\vec{c}$ is written to memory. When the system reads the memory address of interest, the ECC decoder hardware obtains the *received codeword* $\vec{x} = \vec{c} + \vec{e}$. Here, $\vec{e}$ is an error-vector of length n that represents where memory faults, if any, have resulted in changed bits/symbols in the codeword. The decoder multiplies the received codeword $\vec{x}$ with parity check matrix $\mathbf{H}$ to calculate the *error syndrome*: $\vec{s} = \mathbf{H}\vec{x}^{\mathrm{T}}$. The following conclusions can be drawn from the syndrome:

- s = 0: No error.

- s $\neq$ 0: Error detected; syndrome is matched with columns of the parity check matrix **H** to determine the exact bit-location of the error. If the syndrome matching is unsuccessful, the decoder declares it as a detectable-but-uncorrectable error (DUE).

The syndrome is generated without any knowledge about the exact number of errors in the received codeword. If the number of errors exceeds the correction capability of the code and the syndrome matching is successful it would mean one of the following scenarios have occurred:

- s = 0: The decoder declares the codeword error-free and all bits of errors go undetected.
- s $\neq$ 0 and points to a bit: This bit can be one of the erroneous bits or a non-erroneous bit. In either case, the decoder will flag a CE and miscorrect that bit.

This leads to silent data corruption (SDC) where the decoder wrongly declares data with errors as correct. In this work, we attempt to reduce such SDC events when double-bit errors occur.

6.2.3 SEC vs. SECDED

Single-Error Correcting (SEC) codes ($d_{min} = 3$) correct all possible SBEs. The columns in the parity-check matrix **H** of a linear SEC code are distinct and the minimum number of columns to form a linearly dependent set is 3. This ensures that every legal codeword is at-least 3 bit flips away from each other. Single-Error Correcting, Double-Error Detecting (SECDED) codes ($d_{min} = 4$) [152] can correct all SBEs and detect all possible DBEs. The minimum number of columns to form a linearly dependent set in the parity-check matrix **H** of a linear SECDED code is 4. Every legal codeword is at least four bit flips away from each other. Both these codes can correct SBEs. In case of DBEs, SEC code either declares a DUE or miscorrects by flipping a third bit. SECDED, on the other hand, always declares a DUE when a DBE occurs.

6.3 Motivation

6.3.1 Miscorrections by On-Die ECC

Let us consider the common example of a DRAM device with a (136,128) SEC Hamming code. This SEC code can correct any SBEs. However, in case of a multi-bit error, there are two possible outcomes: (1) The errors go undetected and is equivalent to not having an on-die ECC mechanism. (2) The multi-bit error aliases to a single-bit error. This happens when the sum of the columns in the H-matrix of the decoder corresponding to the error positions is equal to another column in the matrix.

In order to better understand the second case, consider the following example SEC $\mathbf{H}_{\text{example}}$ parity-check matrix with 128 message bits and $r = 8$ parity bits:

$$\mathbf{H}_{\text{example}} = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \end{array} \left[\begin{array}{cccccc|cccccccc} d_1 & d_2 & d_3 & d_4 & d_5 \ldots d_{127} & d_{128} & p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 & p_8 \\ \hline 1 & 0 & 0 & 1 & \ldots. & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & \ldots. & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \ldots. & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ldots. & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \ldots. & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \ldots. & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & \ldots. & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & \ldots. & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right],$$

where d_i represents the ith data bit, p_j is the jth redundant parity bit and c_k is the kth parity-check equation. Now, if a double-bit error occurs in bits 1 and 2, the resulting codeword c' is equivalent to adding error patterns e_1 and e_2 to the original codeword c. By the definition of a linear block code, $H.c = 0$ for all legal codewords c. Therefore, error patterns e_1 and e_2 isolate columns 1 and 2 of the SEC H matrix (i.e., $\mathbf{H}_{example_{*,1}}$ and $\mathbf{H}_{example_{*,2}}$) and as shown in Equation 6.1, the resulting syndrome is the sum of the two columns.

$$\mathbf{s} = \mathbf{H}_{\text{example}}.\mathbf{c}' = \mathbf{H}_{\text{example}}.(\mathbf{c} + \mathbf{e_1} + \mathbf{e_2})$$

$$= \mathbf{H}_{\text{example}}.\left(\mathbf{c} + \begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix} + \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix} \right) = 0 + \mathbf{H}_{example_{*,1}} + \mathbf{H}_{example_{*,2}} = \begin{bmatrix}1\\0\\0\\0\\0\\0\\1\\0\end{bmatrix} = \mathbf{H}_{example_{*,4}} \tag{6.1}$$

Now, the sum of columns 1 and 2 of the $\mathbf{H}_{\text{example}}$ matrix is equal to column 4. Therefore, the generated syndrome **s** matches column 4. As a result, the decoder would consider it as a single bit error in bit position 4 and flip it as part of its correction mechanism. Thus, an originally double-bit error has now become a triple-bit error. On an average (across 10 random SEC Hamming code constructions), the chances of a DBE miscorrecting to a triple bit error is >45%. With increasing DRAM error rates, recent studies [26, 28, 145, 147] have shown that the probability of a DBE occurring within the 128-bit dataword can be as high as $\sim 8 \times 10^{-5}$, which translates to a DBE every 12500 SEC decoding cycles. Thus, the chances of a double-bit error converting to a triple-bit error are also high and will only increase in future.

6.3.2 SDC post in-controller SECDED decoding

Now let us look at the problems that arise because of this miscorrection. SECDED code inside the memory controller is not designed to detect more than double-bit errors. As a result, when the (136,128) SEC on-die ECC miscorrects and converts a DBE to a triple-bit error, there is a high probability (greater than 50% on an average over multiple SECDED codes) for the SECDED decoder to consider it as an SBE and further miscorrect. This will happen when the generated syndrome or the sum of three columns in the SECDED parity check matrix corresponding to the erroneous bits is equal to a fourth column. The probability of SDC depends on the exact

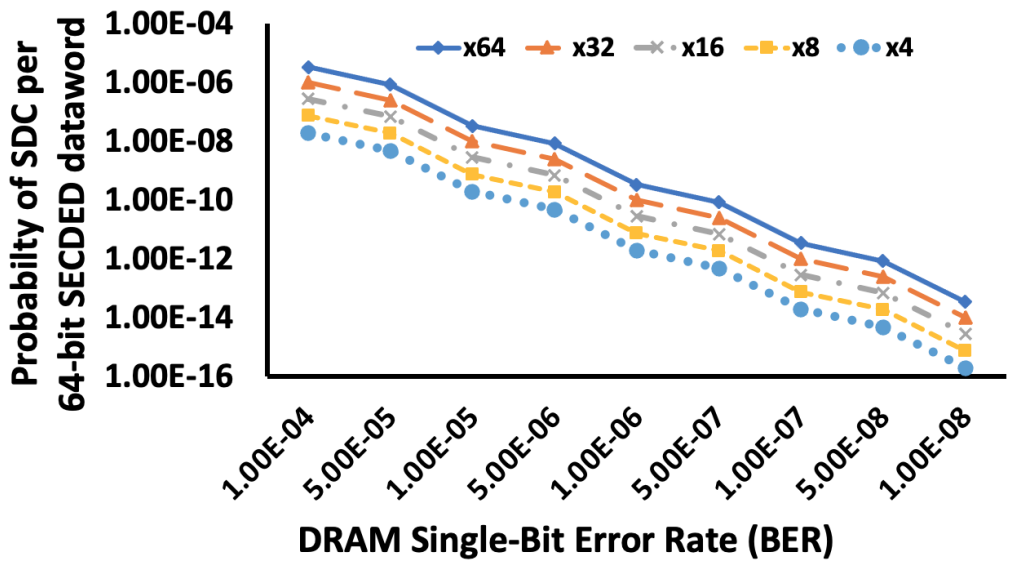


Figure 6.2: Probability of SDC every 64-bits of SCEDED dataword read from memory when a double-bit error occurs in a system with (136,128) on-die SEC and (72,64) in-controller SECDED coding schemes for different bit error rates and data access protocols is shown here.

SECDED code and the memory data transfer protocol. A widely used on-die ECC is (136, 128) SEC [24, 143] and in-controller ECC is (72, 64) SECDED [153]. For the rest of the chapter, we will use these two codes for explaining our proposed code construction mechanisms and DBE correction technique. However, our proposed constraints can be easily extended to other SEC and SECDED code constructions with different dataword and codeword lengths.

DBEs are becoming more probable with increasing bit error rate in recent DRAM generations built. Multiple recent experimental/industrial studies [26, 28, 145, 147] have considered DRAM raw bit error rate (BER) as high as 10^{-4}. For different memory system architectures and data access protocols, we evaluate the probability of SDC when a DBE occurs for BERs ranging from 10^{-4} to 10^{-8}. The result is shown in Figure 6.2. For this evaluation we consider the average miscorrection rate across ten different (136,128) on-die SEC and (72,64) in-controller SECDED implementations. We evaluate for different access protocols; x64 means all 64-bits of SECDED dataword come from the same DRAM chip while x4 means there are 16 DRAM chips and each DRAM chips sends 4-bits per beat of memory transaction. For a BER of 10^{-4}, the probability of silent data corruption in the case of x16 data access protocol is non-negligible and can happen once every 3 million 64-bit

accesses. As the data width per chip reduces, the SDC probability decreases. This is because the probability of a DBE, along with the miscorrected bit, aligning perfectly within the same beat boundary reduces with decrease in beat width. Without on-die SEC, the SDC probability, however, is 0 since all double-bit errors within the DRAM array, irrespective of location, would not get miscorrected and would be flagged as DUE by the in-controller SECDED decoder. Thus, while the SEC code does not help with detecting or correcting the double-bit errors in any scenario, it causes miscorrection and turns upto 25% of these DBE events into SDC.

6.4 COMET ECC Design to Eliminate Silent Data Corruption

In today's DDR or LPDDR based systems, during every read operation, the data that is read into the memory controller is typically striped across multiple DRAM dies. Each xN DRAM die sends N-bits data in parallel during each beat of memory transfer to construct the 72-bit controller codeword. But the on-die ECC decodes a 128-bit dataword inside each DRAM chip. Only a part of this 128-bit data is accessed by the memory controller per operation (see Figure 6.3) and therefore the data of on-die ECC eventually spans multiple in-controller SECDED codewords. This has significant implications on SDC probability. The DBE probability in a 128-bit word and the SEC-induced miscorrection rate remain constant across the DRAM dies having the exact same SEC implementation. However, the probability of the double-bit error and the miscorrected bit coinciding within the same in-controller 64-bit dataword decreases with the decrease in the amount of data from each on-die codeword that constitutes the in-controller codeword (as shown in Figure 6.2). If all 64-bits come from the same DRAM chip and, therefore, from the same 128-bit SEC dataword, the SDC probability is $\sim 11x$ higher than the case where 16 DRAM chips send 4-bits each in parallel.

In this work, we provide two possible solutions that exploit this data access pattern to completely avoid SDCs when DBE occurs. (1) An on-die SEC construction technique which ensures that the miscorrected bit is steered to a different beat. It does em not require knowledge of the exact in-controller code and is compatible with any SECDED implementation in the controller. (2) Our alternate solution outlines an in-controller SECDED construction technique that ensures that none of the on-die aliasing triplets result in SDC. However, unlike the first technique, this in-controller

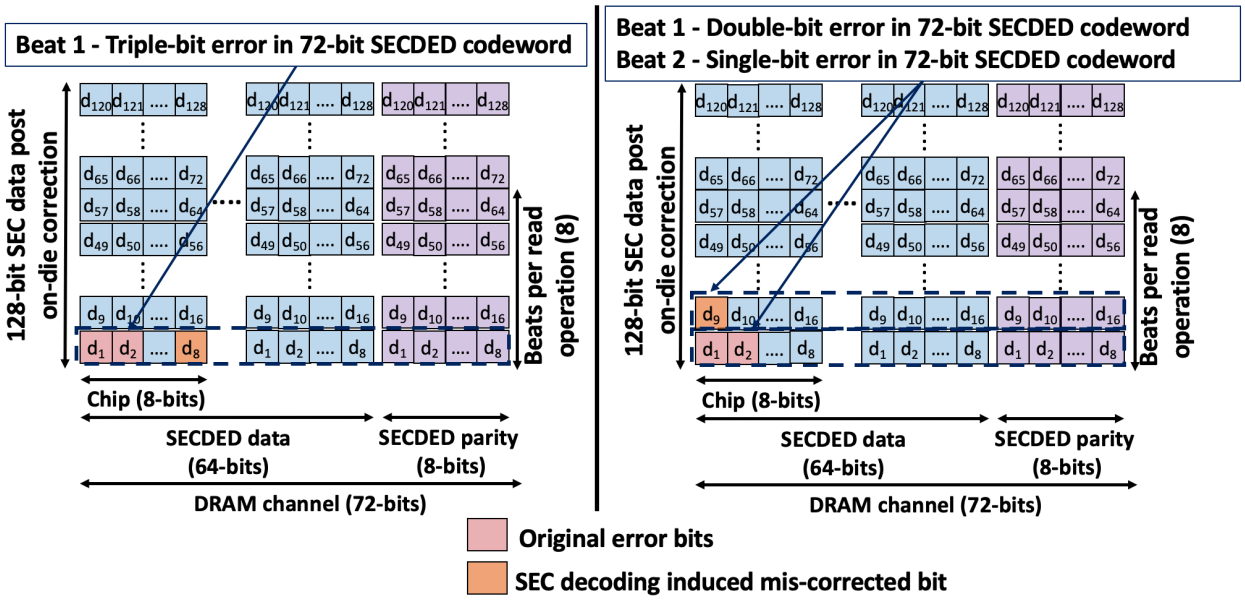


Figure 6.3: Example showing how steering the miscorrected bit to a different beat transfer boundary during SEC decoding prevents the SECDED decoder from encountering the problematic triple-bit error within the same 72-bit codeword.

SECDED construction needs to know the on-die SEC code.

6.4.1 On-die SEC-COMET ECC

In this work we exploit the data transfer protocol in DRAMs to take care of SDCs. As shown in Figure 6.3, if all the three erroneous bits in the 136-bit codeword do not get transferred and decoded in the memory controller in the same beat, the SECDED decoder will not encounter a triple-bit error and SDC can be avoided. Thus, the on-die SEC has to be carefully constructed so that the miscorrection from any DBE gets steered to a bit that is beyond the single beat transfer boundary. This will ensure that the three erroneous bits never coincide in the same 72-bit SECDED codeword.

In order to achieve this property in a (136, 128) SEC code, within every beat transfer boundary, the sum of any two columns in the parity check **H** *matrix should not be equal to a third column in the same set.*

Step-by-step code construction and mathematical guarantee for largest possible beat transfer size: With 8-bits of parity per 128-bits of dataword, the COMET-SEC additional constraint can be

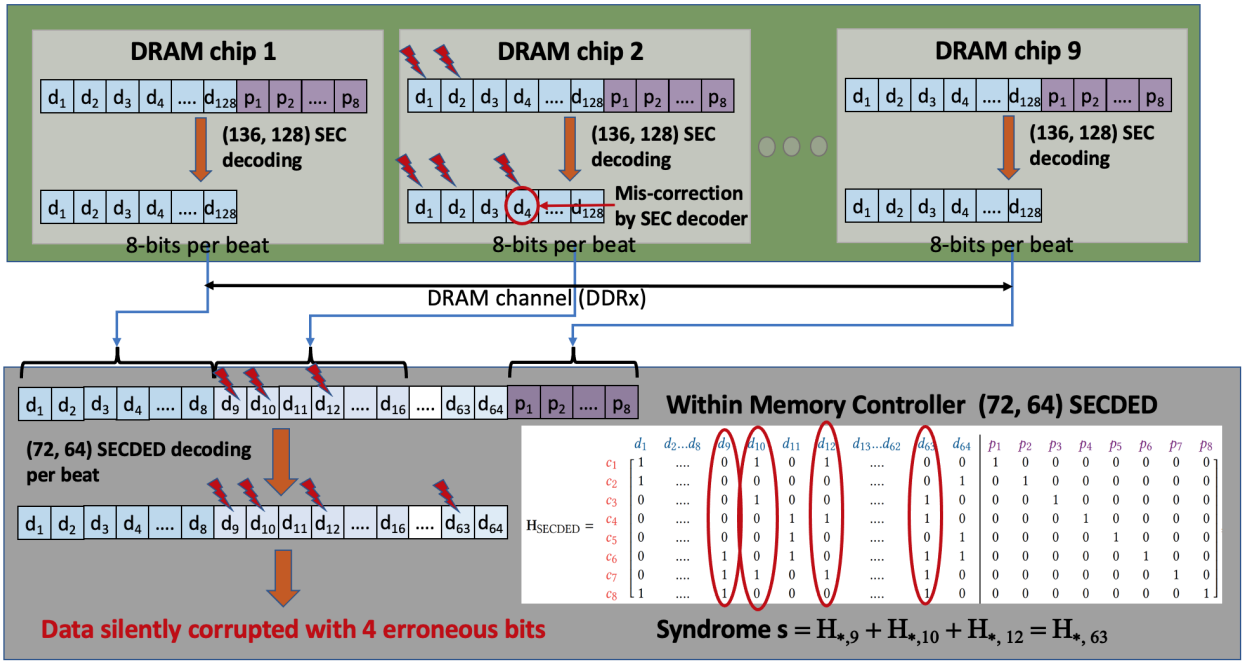


Figure 6.4: Example showing SDC occurring due to miscorrection introduced by on-die ECC. We have considered the SEC construction provided in Section 6.3.1 where the sum of columns 1 and 2 in the $H_{example}$ matrix is equal to column 4.

satisfied when designing the SEC code for any data transfer protocol as long as the beat transfer boundary (N) consists of 64-bits (64 columns) or less. When constructing the 8x136 parity check matrix H, we can choose the 136 8-bit columns from 128 odd-weight and 127 even-weight non-zero options. The DBE-induced miscorrection happens when the sum of two columns is equal to a third column in the H matrix. Either all these three columns would have even weights or two of them would have odd weights and the third would have even weight. If Thus, the two aliasing sets possible are: (1) (odd, odd, even), (2) (even, even, even). If we could construct the H matrix with all odd weight columns then no pair of columns would sum up to a third column and there would be no DBE-induced miscorrection. However, we do not have enough odd-weight columns for the entire matrix. Hence, when constructing the H matrix for our proposed SEC-COMET code for an xN DRAM architecture, we use a two-step approach.

- Out of the 128 odd weight columns, we use the single weighted columns for the last 8 identity sub-matrix columns. We use the remaining 120 odd-weight options for the first $min(128-N, 120)$ columns. None of these columns would have the problem of aliasing since they are all odd.

- For the remaining $X = max(N, 8)$ locations, we use only even weight columns. We randomly choose a bit position (say bit 0) and set it to '1' for all X columns. If bit 0 for all X columns is '1', then the sum of any pair of columns cannot equal a third column in this set as bit 0 of the sum would always be '0'.

The total number of such even weight columns possible $= \binom{7}{1} + \binom{7}{3} + \binom{7}{5} + \binom{7}{7} = 64$. Therefore, N can be as wide as 64 (largest possible factor of 128). Thus, on-die SEC-COMET code can be constructed for x4 to x64 DRAMs that can *guarantee* no silent data corruption. Note that this SEC-COMET construction requires no knowledge of the in-controller SECDED code.

6.4.2 In-controller SECDED-COMET ECC

An alternative to imposing the additional COMET constraint on on-die SEC ECC as in Section 6.4.1 is to redesign the in-controller SECDED code, albeit with the knowledge of the SEC code used in

the memory device (for example by using the recent work [26] which proposes an efficient way of reverse engineering the exact on-die SEC implementation). Once we know the SEC code, we will know all the bit positions pairs (H column pairs) that lead to a miscorrection (sum of columns equal to third column) within the same beat transfer boundary. These are the triplets that eventually can lead to SDC. We first list all such bit positions triplets.

For every triplet, we calculate the all the possible corresponding bit positions in the SECDED dataword. To understand that let us again look at $\mathbf{H_{example}}$ provided in Section 6.3.1. Errors in bits 1 and 2 in the 128-bit SEC dataword lead to miscorrection in bit 4. Now, in an x8 DRAM architecture, bits 1, 2, and 4 in the SEC dataword fall within the same beat transfer boundary and can correspond to any of the following bit positions (in their respective order) in the SECDED dataword (spanning 8 DRAM chips):

- Bits 1, 2 and 4
- Bits 9, 10 and 12
- ....
- Bits 57, 58 and 60

This is because bit 1 of the SEC dataword from chip 1 would be bit 1 of the SECDED dataword, but bit 1 of the SEC dataword from chip 2 would be bit 9 of the SECDED dataword. The same is true for the rest of the DRAM chips. We need to consider positions corresponding to all DRAM chips since they all would use the same on-die SEC code.

Now let us consider the example shown in Figure 6.4. A DBE affects bits 1, 2 in chip 2 and bit 4 gets miscorrected by the SEC decoder. Post data transfer, this translates to triple-bit error in bit positions 9, 10 and 12 in the SECDED codeword. This becomes an SDC since the sum of these columns in the SECDED H matrix is equal to another column (column 63 in the example). The decoder flips bit 63, declares the error correction as a success, and sends the corrupted data over to the processor. In order to prevent this SDC from happening in a system with this particular on-die SEC code, the SECDED parity check matrix has to be designed such that the sum of all the sets of

columns corresponding to the bit positions listed above do not match with any of the columns in the rest of the H matrix.

The process has to be repeated for all bit triplets in the SEC dataword that lead to three-bit errors in the final SECDED codeword. *For a given SEC code and system memory architecture, for every bit/column triplet in the SECDED* **H** *matrix that can cause SDC, the sum of the columns has to be such that it equals no other column in the* **H** *matrix.*

Using this technique, given the exact SEC implementation and the system architecture, it is possible to construct the SECDED code that would prevent SDC when double-bit errors happen.

6.5 COMET Double-bit Error Correction

On-die ECC adds 6.25% parity storage overhead without improving error correction capability. Previous studies have shown that there is almost no difference in reliability between DIMMs with 8 chips that have only on-die ECC and DIMMs with 9 chips that support both on-die ECC and rank-level in-controller SECDED ECC [25]. Thus, the two disjoint ECC schemes together do not reduce the overall system failure probability. Instead we have shown that, if one of them is not carefully designed, it causes additional SDCs. In this section, we show how DBE correction can be achieved with no extra parity overhead using the redundancy built within the two codes. We add one more constraint to the on-die SEC code construction and devise a controller-device collaborative correction scheme to get nearly perfect double-bit error correction. *It is important to note that even though the collaborative technique requires controller-device communication using a special command, the two ECC codes can be designed completely independently and does not require any special in-controller SECDED construction.*

6.5.1 Constructing on-die SEC code to enable Double-bit Error Correction (SEC-COMET-DBC)

In order to enable detection and correction of DBEs using syndrome matching we need to ensure that the sum of any pair of columns in the parity check matrix **H** generates a unique syndrome.

However, with just 8-bit redundancy for a 128-bit dataword, this can be achieved only for a small subset of columns. We add a constraint to SEC-COMET code construction from Section 6.4.1 to construct the SEC-COMET-DBC code: for every set of x consecutive columns, the sum of every pair of columns within that set should be unique. For a (136, 128) SEC code, the maximum value of x (that is also a factor of 128) for which this can be possible is 16. I.e., a valid SEC-COMET-DBC code can be constructed for x4, x8, x16 DRAM chips but not for x32. This is because, for every pair of columns to generate a unique syndrome in a set of 32 columns, $\binom{32}{2} = 496$ unique syndromes are required. This is not possible with 8-bits.

For such a SEC code, when a double-bit error occurs in bit positions that belong to the same x-bit chunk, the generated syndrome and the chunk position can be used to figure out the exact DBE locations. The syndrome is generated by the SEC decoder, but for the correction mechanism to work, the errors also have to be localized to the exact x-bit chunk which the SEC decoder is unable to do. For this localization we will exploit the memory data access architecture and utilize information from the in-controller SECDED decoder. For example, in a standard x8 DDR based ECC DIMM, the beat transfer width per chip is 8 and therefore, we use $x = 8$ in the (136, 128) SEC-COMET-DBC code. Now when a DBE happens within the same 8-bit chunk in one of the DRAM chips, the beat in which the decoder flags a DUE will help to point to the 8-bit chunk position where the DBE has occurred. Next, we discuss how this information can be sent to the DRAM chips and the the DBE correction flow. For better understanding we explain the mechanism using a x8 DDR architecture.

6.5.2 Collaborative DBE Correction

6.5.2.1 Detecting the DBE beat

Let us look at all the possible ways a double-bit error can happen in a 136-bit codeword in a particular DRAM chip and the possible outcomes after the on-die and in-controller decoding.

- Case 1: The two error bit positions are in two different 8-bit chunks and the miscorrected bit (if any) belongs to a third chunk. As a result the erroneous bits get decoded in the memory

controller in separate beats. In each of these beats, the SECDED decoder flags a CE and corrects the error. Eventually all the erroneous bits get corrected and no DUE gets flagged.

- Case 2: The two error bit positions are in two different 8-bit chunks and the miscorrected bit falls in the same chunk with one of the error bits. Now one 8-bit chunk that has two errors and one has single-bit error. The in-controller SECDED decoder will flag a CE when it decodes the chunk with SBE but will flag a DUE when the 8-bit chunk with two error bits is decoded.

- Case 3: The two error bit positions are in the same 8-bit chunk. The SEC-COMET constraint (provided in Section 6.4.1) will ensure that the miscorrected bit lands in a different 8-bit chunk. Thus, after SEC decoding the 128-bit dataword either has one 8-bit chunk with two errors (in the case of no miscorrection) or has an additional 8-bit chunk with a single-bit error. The SECDED decoder will flag a DUE when the 8-bit chunk with two error bits is decoded.

Overall we see that if any two of the error bits collide in the same codeword, the SECDED decoder would flag a DUE. Let us consider the example shown in Figure 6.3 (Case 3). A DBE occurs in DRAM chip 1 in bits 1 and 2. Because of our improved SEC construction (shown on the right), it is ensured that the SEC decoder would steer the miscorrection to a different 8-bit chunk (in this example the miscorrected bit is 9). Therefore, during the first beat of memory transaction, the SECDED decoder flags a DUE, while in the second beat it flags a CE and corrects bit 9. The memory controller communicates this information to the DRAM chips using a special error correction command where it sends the original read command address and the beat number in which the DUE was flagged. The SECDED decoder cannot localize the DBE to a particular chunk in the codeword. Therefore, the double-bit error could have occurred in any of the 9 DRAM chips. Every DRAM chip receives the information from the memory controller that a DUE has been flagged in beat 1. Therefore, each DRAM now knows that in the first 8-bits of its 128-bit SEC dataword there might be a double-bit error.

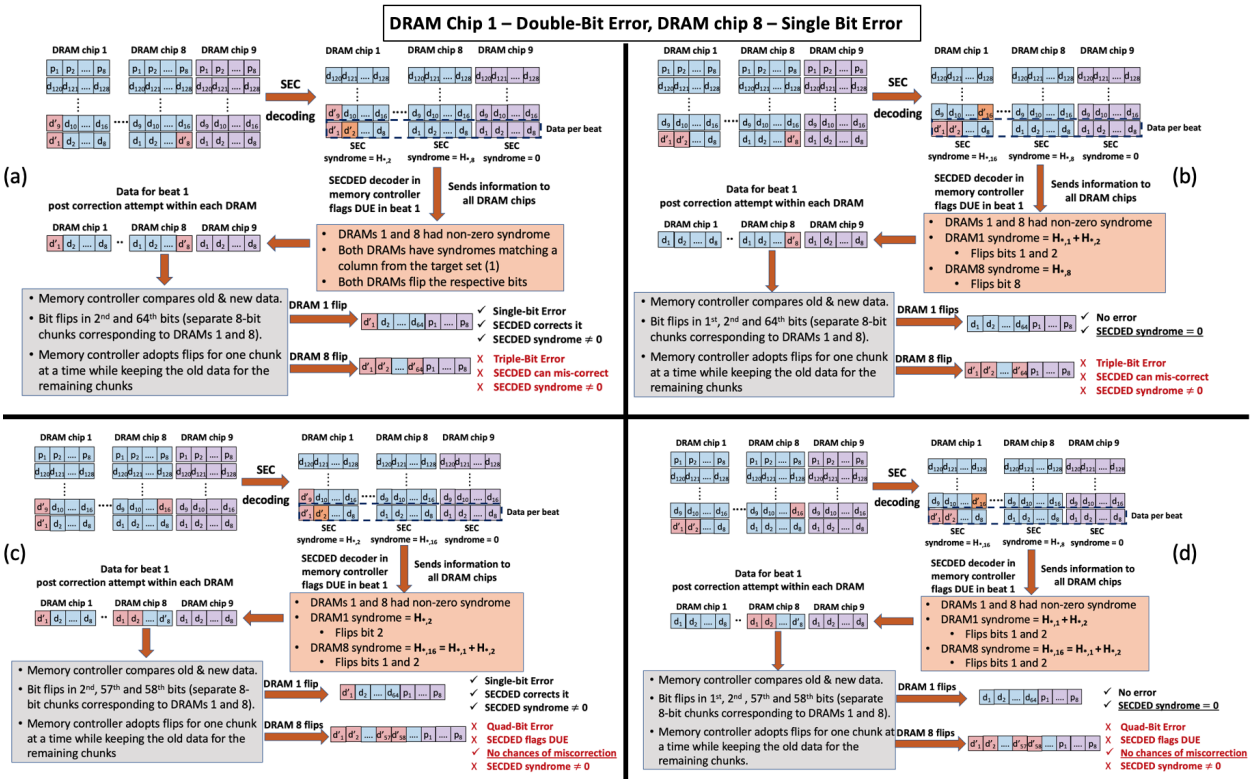


Figure 6.5: The different scenarios possible when one chip has double-bit error and another chip has single bit error that aligns in a way leading to multiple DRAM chips modifying data during DBE correction

6.5.2.2 Correction within each DRAM chip

Once the memory controller sends the special double-bit error correction command with the beat number, each DRAM chip checks the syndrome the SEC decoder had generated during the original read operation. We assume that the special DBE correction command immediately follows the original READ command. Therefore, the DRAM chips only need to store the last generated 8-bit syndrome and the 32-bit/64-bit data that was last read. Storing the original data has negligible overhead but prevents an extra ACTIVATE during correction and possible change in error signature in case of closed page policy. If the syndrome was zero, the DRAM knows that the DBE did not occur in its codeword. In our example (Figure 6.3), all DRAMs except chip 1 would have generated a zero syndrome. If the syndrome is non-zero, the correction mechanism within the chip tries to match the syndrome with one of the **H** matrix columns in the 8-column set that corresponds to the

received beat number. In this case, DRAM chip 1 tries to match the non-zero syndrome against columns 1-8 (beat 1) in the H matrix. We know that the miscorrected bit position is 9. Therefore, the generated syndrome would match with column 9. Since, this column falls outside the target set, the matching is unsuccessful. The decoder moves on to the next step where it matches the generated syndrome with the sum of every pair of columns from the target set. Because of our improved SEC construction, every pair of columns should sum up to a unique value. The pair of columns whose sum equals the generated syndrome (in this example it will be columns 1 and 2) represent the erroneous bit positions. The decoder would flip those two bits and send the corrected data over the DRAM bus to the memory controller. The rest of the DRAM chips would not take any action since they had zero syndrome and send the original 8-bit data.

While the example depicts Case 3, let's look at what happens in Case 2. In this scenario, the original double-bit errors are in two separate beat transfer chunks. But the miscorrected bit lands in the same 8-bit chunk as one of the two errors. Let's say this is the second 8-bit chunk. Thus, the SECDED controller flags DUE in the second beat and sends this information to the DRAM chips. When the erroneous chip matches the generated syndrome against columns 9 to 16 in the **H** matrix, it sees that the syndrome matches with the column corresponding to the miscorrected bit position. In this case, the DRAM chip would only flip that particular bit and send over the data to the DRAM controller. It will not be able to localize and correct the second error position within that 8-bit chunk. Considering the rest of the DRAM chips had zero syndrome, they send their unmodified data over in the same beat. Since the erroneous chip could only correct one bit, the overall data still has one-bit of error that SECDED will be able to correct.

6.5.2.3 Final Correction within the memory controller

The final correction step in the DRAM controller involves multiple rounds of SECDED decoding of the corrected data. This is to provision for the rare cases where DBE in one chip coincides with SBEs in other chips within the same 8-bit chunk and multiple DRAM chips encounter non-zero syndromes. The DRAM(s) in which the SBE falls within the same beat transfer boundary as the one in which DUE was flagged would match their generated syndrome with one of the columns in the

target set and end up flipping the corresponding bit. Thus, while the DRAM with double-bit error is able to correct one/two of the error bits, the other DRAMs with single bit errors and matching syndromes end up corrupting their data. This has to be dealt with in the DRAM controller in order to prevent SDC.

Once the controller receives the miscorrected 72-bit data from the DRAMs, it compares the corrected codeword with the one it had received during the original read. In the ideal case where only a single DRAM chip has double-bit error and no other chip has made any corrections, the two codewords would differ by one/two bits within a particular 8-bit boundary corresponding to the erroneous chip. However, if multiple DRAM chips send modified data, the controller, post comparison, would find bit flips in more than one 8-bit chunk. To prevent miscorrection and silent data corruption, the controller accepts changes corresponding to each chip one at a time. The possible scenarios are shown in Figure 6.5.

- (a) DBE Case 2 (Chip 1) + SBE (Chip 8) in same 8-bit chunk. Post correction, the data received by the memory controller is two flips away from the old data. Each of the two flips are in separate 8-bit chunks and, therefore, is assumed to be introduced by two separate DRAM chips. Chip 1 has corrected the miscorrected bit while chip 8 has accidentally flipped the previously corrected bit, making it wrong again. The controller accepts corrections corresponding to one chip at a time and sends the corrected data through the SECDED decoder. When chip 1 correction is considered, the resulting data ends up with a single-bit error. This is because, the rest of the data bits are the same as it was in the pre-correction data and therefore, the post-correction accidental flip by chip 8 has been replaced by the right data. The only error bit corresponds to one of the double-bit error locations and the SECDED decoder corrects it. However, when chip 8 correction is considered, the resulting data ends up with triple-bit errors. The SECDED decoder, in this case, either flags a DUE or considers it as a correctable SBE if the syndrome matches with an H column. If it flags a DUE, the controller rejects this case, accepts the corrections from chip 1, considers the SECDED correction as legal and moves ahead. If both attempts lead to SECDED correction, the controller panics and declares the DBE uncorrectable.

- (b) DBE Case 3 (Chip 1) + SBE (Chip 8) in same 8-bit chunk. Post correction, the data received by the memory controller is three flips away from the old data. Two of the bit flips are in the same 8-bit chunk while the third is in a different one. Chip 1 has corrected both double-bit errors while chip 8 has accidentally flipped the previously corrected bit, making it wrong again. The controller accepts corrections corresponding to one chip at a time and sends the corrected data through the SECDED decoder. When chip 1 correction is considered, the resulting data is error free. The SECDED decoder returns a zero syndrome. However, when chip 8 correction is considered, the resulting data ends up with triple-bit errors. The SECDED decoder, in this case, either flags a DUE or considers it as a correctable SBE if the syndrome matches with an H column. The controller rejects this case irrespective since it had generated a zero syndrome in one of the other cases, accepts the corrections from chip 1 and moves ahead.

- (c) DBE Case 2 (Chip 1) + SBE (Chip 8) in a different 8-bit chunk. In this case, even though the SBE in chip 8 is in a different 8-bit chunk (b_{16}), sum of two **H** columns in the target 8-bit chunk equal column 16. Therefore, during correction, the decoder would think that there are two errors in the target 8-bit chunk and flip the respective bits. Thus, post correction, chip 1 is able to correct one of the two errors but chip 8 has introduced two additional error bits. The data received by the memory controller is three flips away from the old data. Two of the bit flips are in the same 8-bit chunk while the third is in a different one. The controller accepts corrections corresponding to one chip at a time and sends the corrected data through the SECDED decoder. When chip 1 correction is considered, the resulting data ends up with a single-bit error. The SECDED decoder corrects the error. However, when chip 8 correction is considered, the resulting data ends up with quad-bit errors. The SECDED decoder, in this case, flags a DUE[1]. The controller rejects this case, accepts the corrections from chip 1 and moves ahead.

- (d) DBE Case 3 (Chip 1) + SBE (Chip 8) in a different 8-bit chunk. Same as in (c), chip 8 has accidentally flipped two bits in its data post correction. Chip 1, however, manages to correct

[1]It is assumed that the SECDED H matrix consists of only odd weight columns and can, therefore, detect all quad-bit errors.

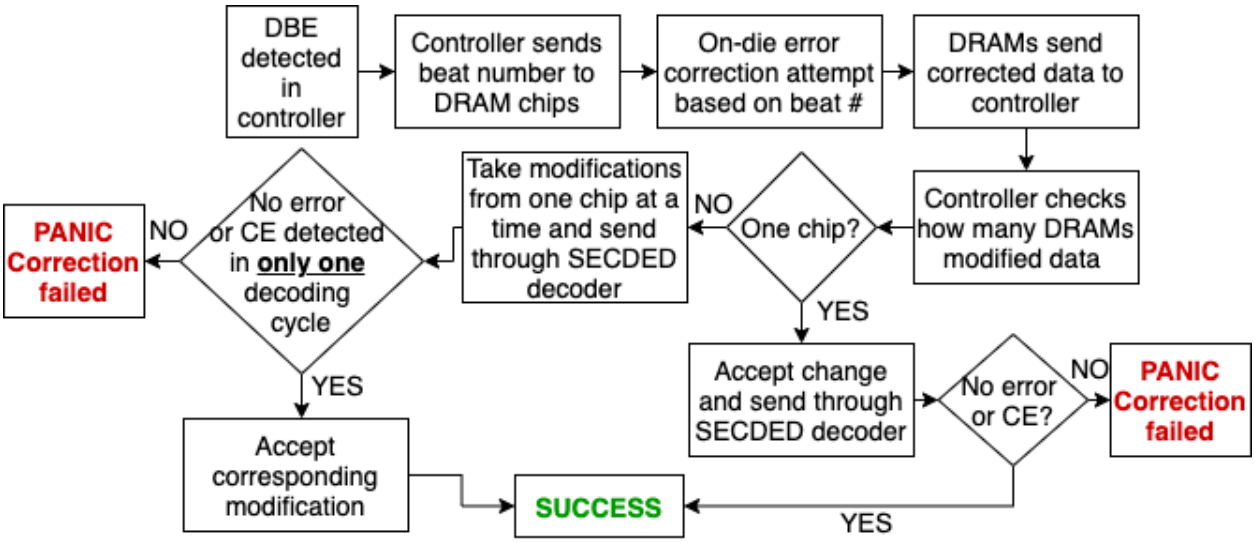


Figure 6.6: Step-by-step COMET double-bit error correction mechanism.

both double-bit errors. The data received by the memory controller is four flips away from the old data, two flips in each 8-bit chunk. When chip 1 correction is considered, the resulting data ends up error-free. The SECDED decoder generates a zero syndrome. However, when chip 8 correction is considered, the resulting data ends up with quad-bit errors. The SECDED decoder, in this case, flags a DUE. The controller rejects this case, accepts the corrections from chip 1 and moves ahead.

From the detailed breakdown of the four scenarios, we see that the correction mechanism is able to successfully correct in three of them. The likelihood of the uncorrectable case is $\sim$1 in 300,000 DBEs (probability of DBE Case 2 in one chip with SBE in another Chip). I.e., COMET achieves 99.9997% double-bit error correction. The step-by-step correction mechanism of DBEs by COMET is shown in Figure 6.6. A similar correction outcome is expected if there is link error instead of single-bit error in the data signals of the other chips. The probability of double-bit error striking two different DRAM chips within the same beat transfer boundary is less than 2×10^{-10} with BER of 10^{-4}. Therefore, we only consider upto single bit error in the other DRAM chips.

Table 6.1: COMET DBE Correction Command Sequence in DDR4 and LPDDR4 protocols

DDR4								
Signals	**Clock Edge**	**Prev. CKE /Pres. CKE**	**CS_n**	**ACT_n**	**RAS_n/A16**	**CAS_n/A15**	**WE_n/A14**	**A[13, 11]**
COMET_special	R1	H	L	H	L	H	H	Valid Signal
Signals		**BG[1:0]**	**BA[1:0]**	**C[2:0]**	**A12/BC_n**	**A10/AP**	**A[2:0]**	**A[9:3]**
COMET_special		BG	BA	Valid Signal	L	L	Target Beat number	Column Address
LPDDR4								
Signals	**Clock Edge**	**CS**	**CA0**	**CA1**	**CA2**	**CA3**	**CA4**	**CA5**
COMET_special-1	R1	H	L	H	L	H	L	BL (L)
	R2	L	BA0	BA1	BA2	C0 (Target Beat)	C9	C1 (Target Beat)
COMET_special-2	R1	H	L	H	L	H	H	C8
	R2	L	C2 (Target Beat)	C3 (Target Beat)	C4 (Target Beat)	C5	C6	C7

6.5.3 Implementation of COMET command

The DBE correction mechanism in COMET requires the controller to send a special correction command to the DRAMs to initiate the on-die correction. This command will need to send the exact beat number during which the DUE was flagged along with the rest of the column address. In DDR4/LPDDR4 standards, there are typically one or more spare command sequences that are reserved for future use (RFU). One such RFU command sequence can be used to support this special command.

In Table 6.1 we have listed a command sequence for DDR4 and LPDDR4 protocols that can be used for COMET DBE correction. In DDR4 it will be a single cycle single command sent on the rising edge of the clock while in LPDDR4 it will be a multi-cycle multi-command sent on successive rising clock edges like their standard read/write operations. In DDR4, address bits A[2:0] determine how the beats would be ordered when sending the data from a particular column address [154, 155] during a read operation. For example, A[2:0] = “010” would send beat number 2 first followed by beats 3, 0, 1, 6, 7, 4, 5, while A[2:0] = “101” would send beat 5 first followed by beats 6, 7, 4, 1, 2, 3, 0. The same address bits can be used in our special command to denote the target beat in which DUE had occurred and the DRAM device would correct and send data accordingly. Similarly, in LPDDR4 protocol [156, 157], C[4:0] of the 10-bit column address (C0 to C9) is used to determine the beat ordering during read operation and can be re-purposed in our special command to send the target beat number. Also, both protocols support burst chop, which allows the DRAM devices to

send reduced number of beats during the memory transaction. Since we need only a single beat post correction from the DRAMs, the special command can enable burst chop. In DDR4, BC_n is set to LOW for a burst size of 4 beats instead of the standard 8 beats. In LPDDR4, the CA5 pin in the first cycle can be set to LOW for the shortest burst length. For DRAM devices that do not guarantee the COMET-SEC-DBC construction, the special command to correct double-bit errors can be turned off in the memory controller.

6.6 Results

6.6.1 Reliability Evaluation

We evaluate the impact of double-bit errors and silent data corruption caused by these errors on system-level reliability through a comprehensive error injection study. While, in most cases, SDCs corrupt the final result or lead to unexpected crashes and hangs during the run of an application, some SDCs might get masked and would eventually have no impact on the final output. Since COMET ensures that none of the double-bit errors result in SDC, our objective is to understand the severity of on-die ECC induced SDCs in the event of a double-bit error without COMET in order to evaluate the usefulness of COMET.

We selected a random implementation of a (136, 128) SEC on-die code that obeys the basic constraints of a Hamming code and only ensures single-bit error correction. For the in-controller ECC, we selected a conventional (72, 64) Hsiao SECDED code [152] that is known to be widely used. Since approximation tolerant applications are expected to mask SDCs and be least impacted by them, we used benchmarks from the AxBench suite [158] for this study. Any standard approximation intolerant application is expected to strictly benefit more from COMET. We built AxBench against GNU/Linux for the open-source 64-bit RISC-V (RV64G) instruction set v2.0 [52] using the official tools [139]. Each benchmark is executed on top of the RISC-V proxy kernel [140] using the Spike simulator [114] that we modified to inject errors. We use our modified version of Spike to run each benchmark to completion 5000 times. During each run, a load operation is randomly chosen and a double-bit error is injected in a 128-bit word. The 128-bit SEC code decodes the erroneous

codeword first, followed by the (72, 64) SECDED decoder. The chosen SEC and SECDED decoder combination has an overall 20.65% probability (average calculated across 100000 random 136-bit codewords) of not flagging a DUE and resulting in a DBE-induced SDC because of miscorrections. We observe the effects on program behavior for the cases where DUE is not flagged and, therefore, corrupted data is sent over to the processor. The results are shown in Figure 6.7.

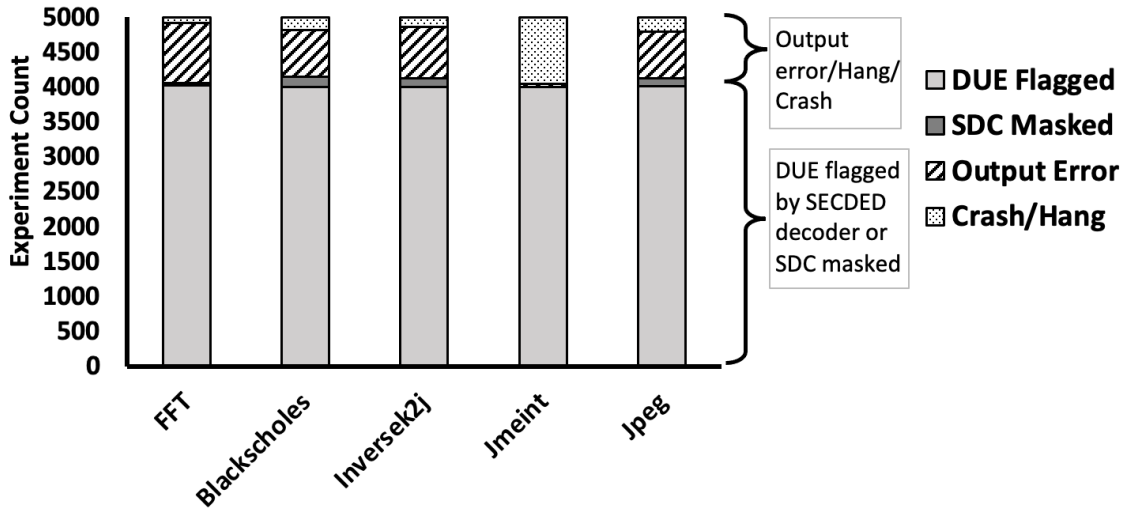


Figure 6.7: The impact of on-die ECC induced SDC in the event of double-bit error on the program behavior when running applications from the AxBench suite.

Overall, on an average, ∼80% of the double-bit errors are flagged as DUE while less than 2% of the times the resulting SDC gets successfully masked by the application. ∼12%, on an average, result in erroneous output with a non-negligible impact on output quality and for the rest of the cases, the program either hangs or crashes.

SEC-COMET or SECDED-COMET code constructions completely eliminate SDCs converting output errors or crashes in the 18% of cases to more acceptable DUEs. SEC-COMET-DBC corrects nearly all of these errors, i.e., 98% point improvement in DBE reliability (no improvement in the 2% cases where the application masks the SDC caused by DBE).

6.6.2 Effectiveness of COMET Double-bit Correction

We evaluate the reliability of a system with 128GB DRAM with three different error correction schemes: no on-die ECC, standard SEC ECC and SEC-COMET-DBC scheme. We used fault simulator MEMRES [159] with real world field data from [160] and [159]. We took into account scaling induced bit error rate of 10^{-4} for this study. Our system has 2 channels, each containing dual ranked DIMM of 64GB capacity with 18 x8 DRAMs. In all three systems we have considered in-controller SECDED protection. We perform Monte Carlo simulations for a 5 year period and consider both undetected as well as detected-but-uncorrectable errors as system failures. For details on each failure mode, we refer the reader to [159]. Overall, we see that adding on-die SEC coding significantly helps in improving device failure by 35% over the system without any on-die coding. The main failure mode that on-die ECC takes care of is single bit permanent fault intersecting with a single-bit transient fault(SBT) in the array or the bus. The SBT in the array is taken care of by the occasional scrubbing that is enabled in the DRAMs. With scrubbing enabled, the DRAM dies, when idle, occasionally activate rows, check for errors in the row using the on-die SEC mechanism, correct (if possible) and write the data back. The intersection with bus faults is taken care by the on-die and in-controller ECCs. With COMET-SEC-DBC, we can achieve a 8.2% reduction in system faults over standard SEC, which translates to more than 150 lesser failures per year. This improvement in memory resiliency comes from double-bit correction which helps to reduce single-row failures and single-word failures.

6.6.3 Impact on Encoder/Decoder Area, Energy and Latency

COMET code constructions do not require additional redundancy bits. But the encoder and decoder circuitry overheads varies based on the exact code implementation. In order to evaluate our proposed SEC code overheads, we synthesized few different SEC implementations along with our construction using a commercial 28nm library. [2] We considered the SEC code with the minimum possible sum of the weight of the columns in the parity check matrix H as the most efficient implementation in

[2]Though DRAM technology is different compared to logic technology, the comparison between different implementations should still hold.

Table 6.2: Synthesis Results for Different x8 SEC Decoder Implementations in Commercial 28nm Library

	SEC-random	**SEC-best case**	**SEC-COMET-DBC (x8)**	**SEC-COMET-DBC (x16)**
Gate Count	168	165	170	170
Area (um2)	331.452	318.168	328.374	332.91
Latency (ps)	512	508	517	520
Power (W)	2.12E-05	1.93E-05	2.09E-05	2.12E-05

terms of gate count. We also compared against a random SEC implementation which satisfies the basic Hamming code constraints required for single error correction.

Based on the results, we see that the difference in area ($<$5%), latency ($<$2.5%) and power ($<$9.7%) among the different SEC decoders is minimal and negligible. Furthermore, on-die ECC consumes a very small fraction of the overall DRAM active power ($\sim$5-7% [24]).

6.6.4 Performance Impact of SEC-COMET-DBC

SEC-COMET has no performance impact. In a system using x8 DDRx protocol based DRAMs with scaling induced bit error rate of 10^{-4} and on-die (136, 128)SEC mechanism, a double-bit error in a 572-bit memory line that causes the (72,64)SECDED decoder to flag a DUE can happen once every $\sim$17000 read operations. This is the probability of DBE occurring within a 136-bit SEC dataword where both error bits are either in the same 8-bit chunk belonging to the 64-bit half that is read from the chip or the mis-corrected bit coincides with one of the two erroneous bits. To evaluate COMET's correction mechanism's impact on performance, we used cycle based simulation of 18 SPEC CPU 2017 benchmarks [161], 8 Parsec benchmarks [162] and 4 applications from the GAP suite [163] on the Gem5 simulator [138]. These are the applications that we could successfully compile and run using Gem5. We used a 2GHz single-core processor with a private 32KB I-cache, 64KB D-cache, shared 512KB L2 cache and shared 2MB L3 cache. For once every

17000 read operations, we doubled the read latency and added worst-case 9 memory cycle penalty for the DBE correction. We evaluated the DDR4-2400-x8 memory configuration with a 64b data channel for 2-billion instructions. The overall performance impact was less than 1.2% compared to an oracular case with no memory errors. This is because one additional memory read every 17k reads is still rare and has negligible impact on queuing delay and overall execution time. Of course, the impact on overall performance reduces with reduction in BER ($<$0.1% for BER of 10^{-8}). *Note that, in absence of SEC-COMET-DBC, these DBEs would require frequent checkpoint-recovery, the performance cost of which is extraordinarily high (30 minutes to restore a checkpoint [164]).*

6.7 Discussion

6.7.1 Independent design of on-die and in-controller codes

All three COMET schemes proposed allow within-DRAM SBE correction that is invisible to the rest of the system. Two of the schemes (SEC-COMET and SEC-COMET-DBC) allow independent code constructions by DRAM and CPU vendors. SEC-COMET and SEC-COMET-DBC require the DRAM vendors to add constraint(s) while constructing the on-die SEC. But the CPU vendors can design any SECDED code independently without requiring any knowledge of the on-die SEC implementation. We proposed in-controller SECDED-COMET for the case where SEC-COMET construction is not guaranteed by the DRAM vendor. SECDED-COMET guarantees protection from DBE-induced SDCs only for those DRAMs that have the on-die SEC implementation used for SECDED-COMET construction.

6.7.2 Using Stronger On-die Codes

SECDED code has the ability to detect double-bit errors, not correct them. Having on-die SECDED would prevent DBE-induced miscorrections. However, as DRAM vendors prefer check bits in multiples of 8 [142], the on-die ECC would be (72, 64)SECDED. This would double the parity storage overhead from 6.25% in (136, 128) SEC to 12.5%. Even after doubling the parity overhead, the code will only be capable of avoiding miscorrections due to DBE, it will not be able to correct

the DBE. Our proposed COMET schemes, explained in detail in the subsequent sections, have the same overhead (6.25%) as today's on-die SEC code while avoiding all DBE-induced SDCs and correcting almost all (99.9997%) DBEs. Therefore, using on-die SECDED only doubles the parity overhead while having weaker protection capabilities as compared to COMET. Similarly, double error correcting also requires twice the number of parity bits per 128-bit of dataword and significantly increases the latency, area and power overhead of the encoder/decoder circuitry.

6.7.3 Using Stronger In-controller ECC

Using a double-error correcting, triple error detecting (DECTED) scheme in the memory controller will require additional storage and data lines to transfer the extra parity bits. For every 64-bits of dataword, DECTED requires 7 extra parity bits as compared to SECDED. In some high performance, high-reliability expensive systems today, single symbol correcting, double symbol detecting (SSCDSD, also known as Chipkill) coding is used to tolerate upto single chip failures. However, the standard 4-bit symbol Chipkill code used today can support only x4 DRAM chips [25]. In order to use x8 DRAM, one data access will have to be split into two, which will have a significant impact on performance. Entire chip failures are very rare and, therefore, Chipkill is considered an overkill in most systems today [25].

6.7.4 Comparison with Past Works

Several past works have proposed stronger memory reliability but most of them either do not improve on-die ECC or incur overheads and require changes to the standard protocol. XED [25] proposes using error detection within each DRAM die and then exposing the detection result to the in-controller code for correction. But they assume that on-die codes implemented in today's DRAM have guaranteed double-error detection capability while in most known cases [24], the on-die code only guarantees single-error correction. Using the same code for multi-bit error detection will not be effective as the code would miscorrect. Besides, it does not support silent SBE correction within DRAMs which is desired by DRAM vendors. Similarly, DUO [165] also gets rid of on-die SBE correction and uses those additional bits for stronger in-controller protection. Thus, DRAM

vendors cannot use DUO to improve yield. Besides, it requires non-negligible changes to the existing memory protocols. A recent work [142] highlights the aliasing problem in SEC codes and provides a construction technique that would result in minimal aliasing. However, their code would still result in SDCs when paired with in-controller SECDED unlike COMET that completely gets rid of SDCs by carefully steering the miscorrected bit. PAIR [166] uses on-die SECDED that requires N on-die ECC decoding cycles for xN DRAM. It ensures that each DQ bit comes from a separate codeword. This incurs a significant latency overhead and is not feasible for larger data width (x16/x32). Besides, it requires an additional signal to transfer the multi-bit error detection information. Other proposals such as Frugal-ECC [167] enhances the reliability of non-ECC DIMMs by adding parity bits to compressed memory lines. Therefore, the maximum achievable reliability is limited by the compressibility of the memory lines. Software Defined Error Correcting Code (SDECC) [168] proposes using software based heuristic recovery from DUEs. However, the correction is prone to miscorrections and is limited by the value locality of the nearby words in the cacheline. Other proposed reliability techniques like Bamboo-ECC [153] uses large ECC symbols and codewords to provide stronger protection while incurring performance overhead. ArchShield [147] provides protection against single-bit scaling induced errors but requires storing of fault maps within the DRAMs that would need to be updated in-field that requires running full array testing using a Built-In Self Test (BIST) engine. CiDRA [169] proposes using on-die ECC to provide protection against multi-bit failures. However, it requires large SRAM overheads that makes its usage prohibitive. COMET requires no additional storage overheads, no additional signals and still allows the DRAM manufacturers to silently correct the SBEs in the memory array without making them visible to the rest of the system.

6.7.5 Accommodating Wider Data Widths

As mentioned previously in Sections 6.4.1 and 6.5.1, with 8-bits of parity for 128-bits of dataword, SEC-COMET (SEC-COMET-DBC) construction works upto per-chip beat width of 64 (16) bits. For wider interfaces, COMET cannot avoid SDCs or correct DBEs. To enable COMET, the 64-bit SECDED dataword has to be formed using multiple 128-bit SEC datawords. Therefore, within the DRAM chip, every 16-bits of the 64-bit data transferred needs to be a part of a different 128-bit

SEC dataword. Thus, a single write or read command would require multiple rounds of on-die SEC encoding and decoding. Typically, during a read/write operation, an entire DRAM row gets activated into the row buffer. The size of a DRAM row is usually few kBs and therefore, contains multiple SEC datawords. Hence, to enable COMET for wider per chip beat widths, the multiple on-chip encoding and decoding can be done in parallel and would not require multiple activations of DRAM rows.

6.8 Conclusion

Aggressive technology scaling in modern DRMs is leading to a rapid increase in single-cell DRAM error rates. As a result, DRAM manufacturers have started adopting on-die error-correcting coding (ECC) mechanism in order to achieve reasonable yields. The commonly used on-die SEC ECC scheme interacts with in memory controller SECDED ECC, to unfortunately cause silent data corruption in $>25\%$ of double-bit-error cases. To prevent silent data corruption from happening, we introduce COllaborative Memory ECC Technique (COMET), a mechanism to efficiently design the on-die SEC ECC or the in-controller SECDED ECC that steers the miscorrection to guarantee that no silent data corruption happens when a DBE occurs inside the DRAM. Further, we develop the SEC-COMET-DBC on-die ECC code and a collaborative correction mechanism between the on-die and in-controller ECC decoders that allow us to correct the majority of the DBEs within the DRAM array without adding any additional redundancy bits to either of the two codes. Overall, COMET can eliminate all double-bit error induced SDCs and correct 99.9997% of all DBEs with negligible area, power and performance impact.

CHAPTER 7

Compression with Multi-ECC: Enhanced Error Resiliency for Magnetic Memories

Emerging non-volatile magnetic memories such as the spin-torque-transfer random access memories (STT-RAMs) provide superior density and energy benefits compared to conventional DRAM or Flash based memories. However, these technologies often suffer from reliability issues and thus strong conventional reliability schemes are required. These schemes have large overhead for storage which, in turn, can potentially eclipse the density and energy benefits these technologies promise. Moreover, the read and write operations in STT-RAMs show asymmetric behaviour i.e., bit-flip probability of $1\rightarrow0$ is significantly higher than $0\rightarrow1$. However, conventional Error Correcting Codes (ECCs) treat both 0 and 1 flips similarly and thus result in unbalanced reliability of these two types of errors. In this chapter, we propose a new ECC protection scheme for STT-RAM based main memories, compression with multi-ECC (CME). First we try to compress every cache line to reduce its size. Based on the amount of compression possible, we use the saved additional bits to increase the protection to strong ECC schemes, if possible. Compression itself reduces the hamming weight of the cache lines, thus reducing the probability of $1\rightarrow0$ bit-flips. Opportunistically using stronger ECC schemes further helps tolerate multiple bit-flips in a cache line. Our results show that for STT-RAM based main memories, CME can reduce the block failure probability by up to 240x (average 7x) over using a standard (72,64) SECDED scheme. The latency and area overheads of CME is minimal with average performance degradation of less than 1.4%.

Collaborators:

- Saptadeep Pal, UCLA
- Prof. Puneet Gupta, UCLA

7.1 Introduction

The criticality of memories in the design and performance of today's computer systems is becoming increasingly prominent. Main memories serve a pivotal role, sitting in between the processor cores and the slow storage devices. With aggressive technology scaling, a large number of processor cores are being integrated in today's systems. As a result, there is an ever increasing demand for main memory capacity in order to be able to exploit the processing power of these multicore and manycore systems and maintain the performance growth. However, DRAM scaling is unfortunately slowing down. Though DRAM is still the main memory workhorse, several application contexts need different properties from the main memory (higher density, non-volatility, higher performance, etc). Hence, it is becoming increasingly important to consider alternative technologies that can potentially avoid the problems faced by DRAM and enable new opportunities.

Several emerging non-volatile memory (NVM) technologies are now being considered as potential replacements for or enhancements to DRAM. Most of these new non-volatile technologies (Phase Change Memory[PCM], STT-RAM, Resistive RAM[ReRAM], etc.) promise better scaling, higher density, and reduced cost-per-bit [30]. However, they come with their own set of challenges. The biggest problem that these emerging technologies face is the high stochastic bit error rate. In fact, the reliability challenges of NVMs can offset the density and energy advantages that they offer. Increase in demand for memory capacity requires aggressive scaling of area-per-bit of storage. At higher density, these non-volatile emerging memory technologies tend to be more susceptible to stochastic bit errors [31]. Due to the random nature of the bit errors, these memory technologies require stronger in-field error-correcting code (ECC) [32].

The stochastic nature of failures in NVMs is similar to the radiation induced soft errors in DRAM and SRAM and occur without any warning. In order to ensure the integrity of the data, an error detection mechanism, followed by correction of the error(s) needs to be incorporated in a system. In conventional systems, ECC schemes are deployed to recover from memory errors. These schemes require adding redundancy bits alongside the original data (or message). For DRAM based memory, the most commonly used ECC schemes to recover from bit error or faulty chip error are the SECDED (Single-Error Correcting, Double-Error Detecting) [127] and Chipkill-Correct [170]

schemes.

This stochastic bit error rate in NVMs, however, is much higher than the single-bit soft error rate in DRAM. For example, in PCM, a two-bit cell may have 10^6-times higher error rate than DRAM and would, therefore, require a much stronger ECC scheme [32, 171]. Also, for most of these emerging NVM technologies, some states show higher error rates than the rest. As a result, the conventional ECC schemes used in DRAM-based memory need to be extended for providing multi-bit asymmetric protection to maintain acceptable limits of yield and performance of systems with these emerging memory subsystems. However, the cost and complexity of stronger error detection and correction circuitry increases exponentially, requiring much larger number of redundancy bits. This adds overhead not just in terms of storage but also power and performance.

Out of the various magnetic NVMs that have been proposed, Spin-Transfer Torque Random Access Memory (STT-RAM) is one of the most promising non-volatile technologies and has been studied extensively as a scalable non-volatile alternative to DRAM [172–174]. While STT-RAM might not have huge density benefits over DRAM like other technologies like PCRAM [175], its read performance is comparable to that of DRAM. The write energy and latency are roughly 5-10X and 1.25-2X respectively worse than that of DRAM [176–178] but much better than most other non-volatile technologies. Also, it has zero leakage power and much better program/erase endurance than the other competing NVM alternatives. In [178], the authors show that with certain optimizations, a STT-RAM main memory can achieve performance comparable to DRAM, while reducing the main memory energy by 60%, thus, making a strong case for STT-RAMs as a potential main memory alternative. STT-RAMs are also being considered as SRAM substitute for on-chip caches and has already been introduced in several commercial products [179, 180]. Though STT-RAMs are not susceptible to radiation induced soft errors, they suffer from a very high Bit Error Rate (BER) [181, 182]. As the NVM technology scales below 45nm, read disturbance error, retention error due to thermal instability, and write error rates are growing, leading to unacceptably high bit error rates (BER). Several circuit level and bit-cell design solutions have been proposed to lower the error rates [159]. Also, a few recent efforts have been made to provide stronger error resiliency [32, 183, 184]. Most of these solutions, however, result in very high energy and area overhead.

In this chapter, we propose CME (Compression with Multi-ECC), a novel scheme to provide strong error correction in Magnetic RAM (MRAMs) based main memory subsystems. Though the proposed techniques would be useful for other types of DDR-based memories, we only consider the characteristics (and error-rates) of STT-RAM in our evaluations. This chapter makes the following contributions:

- We use compression to reduce the size of each cache line so that the saved bits can be used to opportunistically add stronger protection without incurring the storage overhead of the redundancy bits of the stronger ECC codes. The code is chosen such that the final length of every cache line after compression and ECC remains constant in order to make the proposed CME scheme DDR compatible.
- Given the asymmetric nature of errors in STT-RAMs (explained in detail in Section 7.2), compression not only helps to reduce the length of the cache line, it also reduces the number of '1's (hamming weight) in each cache line.
- Since the final compressed cache line length can be anywhere between 13-bits (best case) and 512-bits (no compression), there is a wide variety of code choices available. We propose a dynamic programming solution to choose optimal mix of ECC codes to use, given the weight distribution of cache line words and the final cache line length distribution after compression. We show that optimized CME can achieve upto 240x (avg. 7x) reduction in block failure probability.
- We present two minimal memory architecture changes required to accommodate the tag chip that holds the tag required for our CME scheme per cache line and show that the performance overhead of CME is less than 1.4 % on average.

7.2 Background

This section provides a brief background on two important concepts: the reliability concerns of STT-RAM based memories, and cache line compression techniques. STT-RAM memories suffer from high read/write/thermal error rates. A lot of these errors are asymmetric in nature, i.e., the

probability of an error happening in a particular state is higher than the rest of the states. Cache line compression, on the other hand, has been used extensively in the past mainly to satisfy the rising demand for memory capacity and bandwidth. But in this work compression is used opportunistically for providing stronger error detection and correction to the cache lines.

7.2.1 STT-RAM Basics

In an STT-RAM cell, data is stored in a magnetic tunneling junction (MTJ). As current is passed through a mono-domain ferromagnet, the angular momentum of the electrons flips the direction of magnetization in the ferromagnet. The basic structure of a STT-RAM cell is given in Figure 7.1. MTJ consists of a tunneling oxide (MgO) separating two ferromagnetic layers. One layer (reference layer) has fixed magnetization and the other is a free layer whose direction of magnetization flips depending on the direction of current of sufficient density. The relative alignment of the two layers results either in a high resistance path (when opposite and usually represents '1') or a low resistance path (when parallel and usually represents '0').

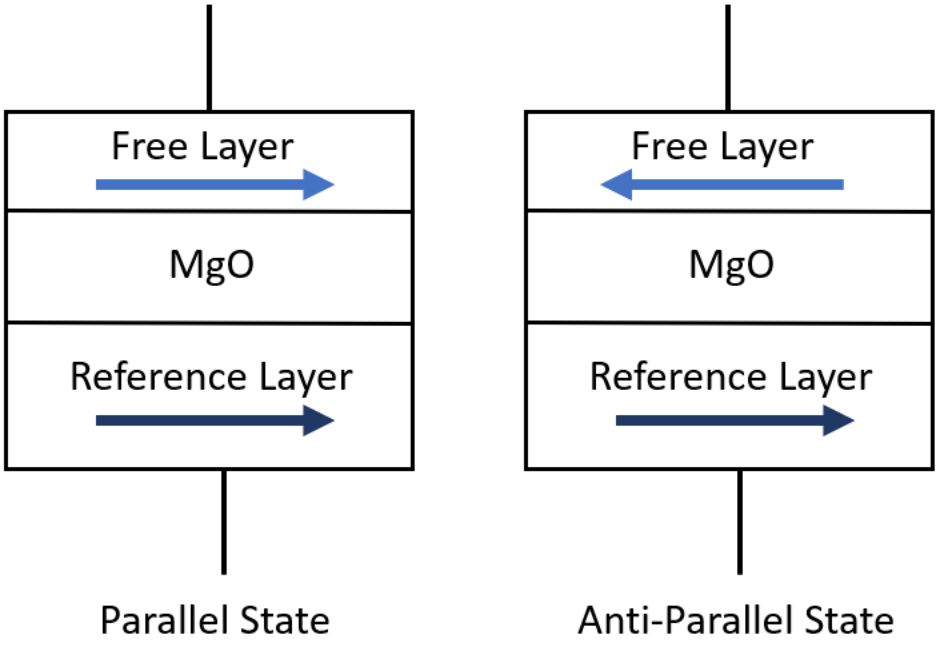


Figure 7.1: Schematic of STT-RAM showing the anti-parallel and parallel states

Errors in STT-RAM can be broadly classified under three categories: read disturb errors, write

errors and retention errors due to thermal instability.

7.2.1.1 Read Errors

The read operation in STT-RAMs is unidirectional, as shown in Figure 7.2. In STT-RAM, feature size scaling has led to a reduction in write current; however, read current has not reduced as much since the correct data may not be sensed when using low-current value. As technology scales below 45nm, read current doesn't reduce significantly beyond 20μA while the write current reduces to around 30μA [182]. Thus, read current is getting closer to the write current such that the read operation now has the potential to alter the stored value. Such an error is called read disturbance error. The data that is read is correct but the stored value becomes erroneous and subsequent reads from this location may contain multiple bit-flips. Since the read current is unidirectional, the unintentional bit flip during read is asymmetric and happens only in one direction ($1 \rightarrow 0$ when reading a '1'). Thus, reducing the number of 1's (or hamming weight) in a cache line will considerably help to reduce the read disturbance errors (RDEs).

7.2.1.2 Write Errors

In STT-RAM, the direction of magnetization in the free layer of the MTJ is flipped based on the direction of the current flowing through the cell. Thus, the value being programmed determines the current direction, as shown in Figure 7.2. In STT-RAM, a write failure happens if the switching current is removed before the MTJ switching completes. The time required for flipping the cell content varies due to the stochastic switching characteristics of the MTJ. However, this failure is also asymmetric [181]. When writing to STT-RAM cells, the MTJ switching from low-resistance state to high-resistance state ($0 \rightarrow 1$) is considered as "unfavorable" switching direction compared to the MTJ switching in the opposite direction: $0 \rightarrow 1$ flipping requires larger switching current than $1 \rightarrow 0$ flipping due to lower spin-transfer efficiency. Also, the variation of MTJ switching time at $0 \rightarrow 1$ flipping is more prominent. Hence, the chances of write error happening are much higher during a $0 \rightarrow 1$ transition than a $1 \rightarrow 0$ transition. As mentioned in [183], the bit error rate of $0 \rightarrow 1$ flipping is $P_{ER,0\rightarrow 1} \sim= 5 \times 10^{-3}$ while that of $1 \rightarrow 0$ flipping is $P_{ER,1\rightarrow 0} \sim= 10^{-7}$. They have also

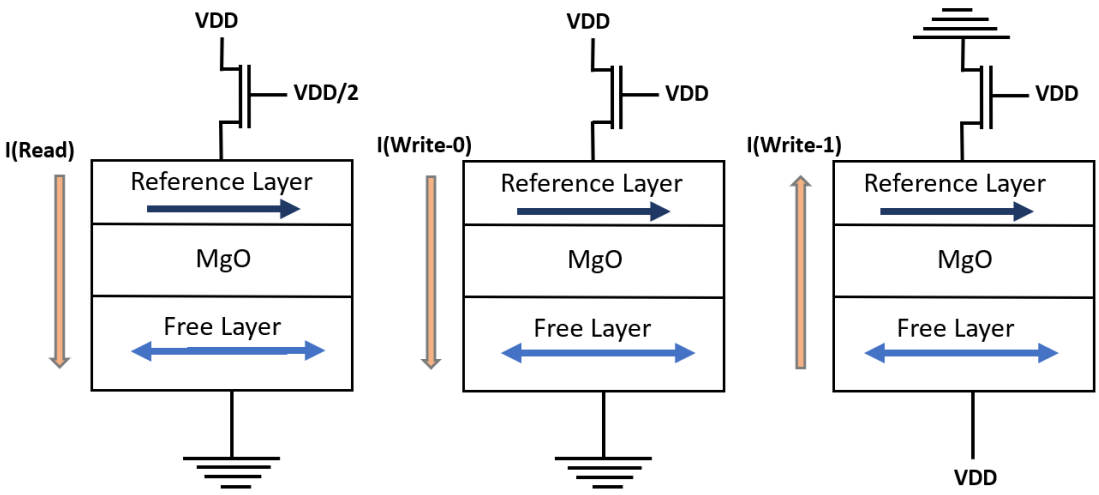


Figure 7.2: Read and write mechanisms for STT-RAM is shown here

analyzed and concluded that the reliability of a word in a cache line decreases exponentially with increase in the hamming weight of cache lines. Thus, just like RDEs, Write Error Rate (WER) can also be reduced by reducing the hamming weight of a cache line.

7.2.1.3 Retention Errors

In STT-RAM, the third major source of errors is retention error where the data stored in the STT-RAM cell flips after a certain period of time. This false switching of data during the standby state is due to the inherent thermal instability of STT-RAMs. Increasing the thermal stability not only reduces retention errors but can also help to reduce read disturb errors [185]. But the critical current or the write current is proportional to the thermal stability of the cell. Higher thermal stability requires a higher write current and/or a longer write pulse. Thus there is a fundamental trade-off between write-ability (write time and/or power) and retention time [186]. Also, thermal stability of STT-RAM cells can be increased by using larger cell sizes, thus, increasing robustness at the cost of area [187].

7.2.2 Previous Work On STT-RAM Reliability

Errors due to read disturbance can be reduced using restore operation, which writes back the data every time there is a read operation [182]. Another work [188] suggests using a pulsed read technique to reduce read disturb errors in STT-RAM cells. However, all these techniques have significant overheads in terms of latency, energy and complexity. One recent work [189] suggests the use of data compression to enable duplication of bits in the memory. If cache lines are duplicated, then a restore operation would be needed only after all the copies have been read. This can potentially decrease the number of restore operations required after every read to deal with read error disturbances. However, if the STT-RAM based memory system uses DDR protocol, the entire cache line (both original and duplicated copies) would get read into the row buffer from the memory array for every read operation. Thus, duplication would technically not reduce the number of restore operations. There are also some bitcell architectures proposed [190, 191] to alleviate the problem of read disturb. However, they also incur significant overheads and only help in dealing with a single type of error. The authors in [192] propose a circuit level technique to detect read disturb errors but detection alone with no correction or reduction in error rates doesn't help to reduce performance degradation that happens due to system crashes when uncorrectable errors occur.

To deal with write errors, [183] suggests reducing the Hamming weight of each cache line. If the number of 1's is reduced in each line, it would considerably reduce the probability of having write errors since a $0 \rightarrow 1$ flip requires longer time and larger current and is thus, more prone to write errors. To reduce the Hamming weight of the cache line, [183] suggests using static/dynamic XOR between words of each cache line exploiting the value locality of stored data. Also, few recent works [32, 193] suggest improvements at the circuit level to improve BER of magnetic memories. Intel, in its recent STT-RAM work [2], proposes using a costly write-verify-write scheme to reduce write errors and a two stage current sensing technique during read to mitigate read disturb error. However, they still have significantly high write bit error rate (can be as high as 10^{-6}).

In [187], the authors proposed using stronger ECC in order to reliably decrease the size of STT-RAM cells and have shown that higher density can be achieved if stronger ECC protection is used. Another work [159] proposed adaptive write-schemes using in-memory variation sensors to

reduce write latency reliably for better application performance. To deal with read margin errors under thermal variation, [194] designed a body-biasing feedback circuit to improve read sensing margin for STT-RAMs. Most of the proposed techniques either target mitigation from one type of error (write or read error) or have very large overheads in terms of circuit complexity, area or power.

7.2.3 Previous Work On Cache Compression

Cache line compression techniques are being widely proposed to satisfy the rising demand for memory storage capacity and memory bandwidth [133, 195, 196]. These techniques exploit spatial and value locality of the data in typical applications. In most cases, there are only a few primitive data types supported by hardware (e.g., integers, floating-point, and addresses), which typically come in multiple widths (e.g., byte, halfword, word, or quadword) and are often laid out in regular fashion (e.g., arrays and structs). The data used in most applications, on the other hand, are low magnitude and are often represented inefficiently, for e.g., 4-byte integer type used to represent values that usually need only 1-byte.

One commonly used compression scheme is Base-Delta-Immediate (BΔI), as proposed in [134]. This scheme exploits the low dynamic range of values present in many cache lines to compress them to smaller sizes. They split up the cache line into multiple equal sized chunks. They take the first chunk as the base and represent all the subsequent chunks as delta with respect to the base. The delta value is smaller than the original value due to the existing value locality in the cache line and hence, can be represented using lesser number of bits. While the authors manage to reduce the compression and decompression latency as compared to the popularly used cache compression techniques such as Frequent Pattern Compression [133], the increase in compression ratio is not significant.

Another recent work on cache compression [135] claims to have better compression ratio than BΔI. This work exploits locality in two layers: within values or words of the cache line data and within bits in the same bit-plane. A bit-plane is a set of bits corresponding to the same bit position within each cache line word in a data array. As a result they manage to achieve higher compression ration than most of the previously proposed cache compression techniques. However, this scheme

has been proposed for 128-byte cache lines and needs to be modified for systems with 64-byte cache lines. Both BΔI and BPC schemes have been compared later in Section 7.6.1 and the pros and cons of each compression scheme in the context of stronger error detection and correction have been discussed.

Most of the past works utilize cache compression to effectively increase the size of the cache. However, our goal is to utilize compression to reduce the hamming weight of the cache lines and also to utilize the additional space to opportunistically add in stronger error correction codes (ECC). Compression with ECC has been proposed previously in the context of DRAM based memory system in FrugalECC [197], COP [198] and Free ECC [199]. In [197] and [199], they used the same protection for every cache line that could be compressed beyond a certain threshold. If uncompressed, the overflow data required additional storage and accesses. In [198], they added error correction only when compression was possible, leaving some cache lines unprotected because of lack of compressibility. Also they use the same protection for every compressed cache line irrespective of how much a particular cache line could be compressed. Another problem with these schemes is that their compression ratio goals are very modest since they focus on metadata. In our case we use different ECC schemes for different cache lines depending on the final compressed size of that particular line. Thus, after encoding with ECC, every cache line is of uniform size and therefore, has no overflow requiring extra storage and accesses. Also we can opportunistically provide much stronger protection to sufficiently compressed cache lines.

7.3 Our scheme - Compression with Multi-ECC (CME)

In this section, we will discuss the details of Compression with Multi-ECC (CME) scheme. Cache line compression is used for two reasons. Firstly, it helps in reducing the hamming weight of each cache line. Secondly, it enables either data duplication (when the compressed cache line is less than half its uncompressed size) or allows to use the available bits to provide stronger error protection. The selection of the stronger ECC scheme depends on the size of the cache line post compression such that the final size of each cache line with the redundant bits remains uniform.

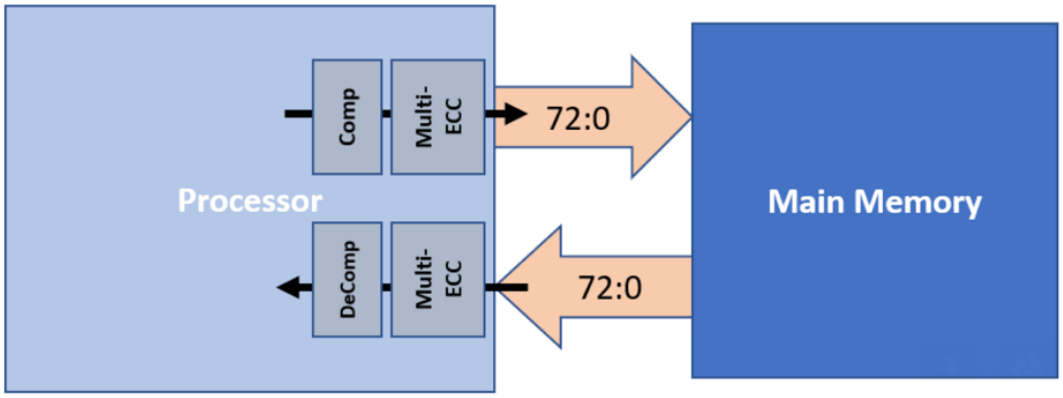


Figure 7.3: Processor Memory system architecture with CME

7.3.1 Overall Architecture

As shown in Figure 7.3, every time a cache line is to be stored in the memory, it is a two-step approach. It first goes through the compression engine and then through the Multi-ECC encoder. In case of a load operation, it first goes through the ECC Decoder and then the de-compression engine. Even though the size of an ECC word is 72-bits, the 73rd bit shown in Figure 7.3 is to signify a tag bit that is sent across the DDR bus in every cycle.

We used a slightly modified version of Bit-Plane Compression (BPC) scheme proposed in [135] which is explained in detail in the following subsection.

7.3.2 Cache Line Compression using modified BPC and an optional Hamming Weight Aware Inversion Coding

Bit Plane Compression (BPC) as described in [135] is a two-step process on a 128B cache line, the first step is where the data is transformed to increase compressibility and the second is to encode the transformed data. We slightly modified the compression scheme for a 64B cache line. We used it for a 64-bit architecture where each cache line word is 64 bits. However, the compression is done on 32-bit words. The modified scheme is shown in Figure 7.4. We split each 64-bit word into two 32-bit sub-words, the higher order bits (bits 63-32) constitute one sub-word and the lower order bits (bits 31-0) form the other sub-word. The first step is cache line manipulation and transformation

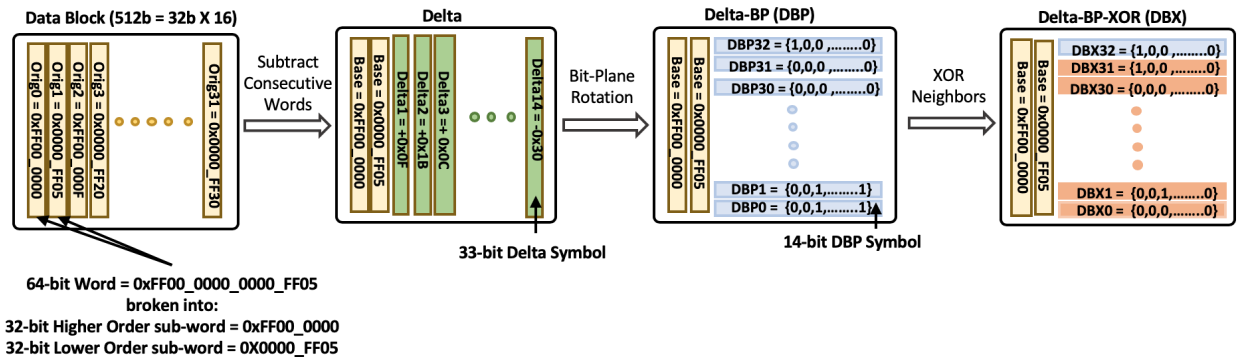


Figure 7.4: An overview of the modified Bit-Plane Transformation scheme

(Delta-BitPlane-XOR [DBX]) to improve compressibility of data and thus reduce the compression hardware complexity.

The next step after data transformation is the compression of the transformed data. BPC combines run-length encoding with a type of frequent pattern encoding to compress the transformed data. As mentioned before, the work in [135] used word-size of 64 bits in a 128-byte cache line, while for our evaluations we use 32-bit words and 64-byte cache line. Hence, our symbol encoding is slightly different from theirs and is shown in table 7.1. This encoding scheme not only helps to reduce the cache line length but also helps to reduce the hamming weight considerably (as seen in Figure 7.11). For instance, in this encoding scheme, a running length of 1's gets encoded to {5'b00000}. The base (first two original) symbols is compressed separately by original symbol encoder as {3'b000}, {3'b001, 4-bit data}, {3'b010, 8-bit data}, or {3'b011, 16-bit data} if its value is 0 or fits into 4/8/16-bit signed integer, respectively. Otherwise, the base symbols are encoded as {1'b1, 32-bit data}.

Since both read and write errors in magnetic memories is asymmetric, reduction of hamming weight is important for resiliency. After each BPC word is encoded, an optional inversion coding [200] can be added to further reduce the hamming weight of the cache line. We check the weight of each BPC encoded word. If it is greater than half the size of the word, we invert the word. In order to facilitate inversion encoding, we add one additional bit in front of each BPC word ('1' if the word is inverted and '0' otherwise). However, note that the additional bit per BPC word required for inversion increases the length of the final compressed line. In Section 7.5.2 we show

that, in many cases, this increase in cache line size due to inversion tag bits leads to weaker ECC scheme and finally, in spite of reducing the hamming weight, inversion ultimately adversely affects the overall block failure probability. One way to reap the benefits of hamming weight reduction using inversion coding without significantly increasing the size of the cache line would be to apply inversion on groups of multiple BPC words instead of a per word basis.

After doing BPC and inversion coding we check the final encoded size of the entire cache line. If the length exceeds 512-bits, the raw cache line is taken and encoded with a (72,64) SECDED code before writing to the main memory. If the compression successfully reduces the size of the cache line, we opportunistically encode the cache line with a stronger ECC code before writing to the memory.

Table 7.1: Frequent Patterns for BPC and DBP/DBX symbol encoding

DBP/DBX Pattern	**Length**	**Code (binary)**
0 (run-length 2∼33)	7-bit	{2'b01, (RunLength-2)[4:0]}
0 (run-length 1)	3-bit	{3'b001}
All 1's	5-bit	{5'b00000}
DBX!=0 and DBP=0	5-bit	{5'b00001}
Consecutive two 1's	9-bit	{5'b00010, StartingOnePosition[3:0]}
Single 1's	9-bit	{5'b00011, OnePosition[3:0]}
Uncompressed	16-bit	{1'b1, UncompressedData[14:0]}

7.3.3 Multi-ECC on Compressed Cache Line

Compression helps to reduce the size of the cache line in most cases. Once the reduction is done, the final size of the cache line determines the ECC scheme to be used. Shorter the length of the compressed cache line, more the available redundant bits for ECC and therefore stronger will be the protection. We wanted to ensure that the STT-RAM based memory subsystem closely matches the standard ECC-DDR protocol. Hence, every fetch will be 72-bit wide.

7.3.3.1 Choice of Codes

The possibility of ECC schemes for a given redundancy is large. Since the final compressed cache line length can be anywhere between 13-bits (best case) and 512-bits (no compression), there is a wide variety of code choices available. Usually, each ECC word in DDR memory is comprised of 72 bits (original message + ECC bits) which is equal to the length of one fetch. However in CME, we also consider ECC code word sizes of 36 (i.e., two ECC words per fetch) and 144 (i.e., one ECC word per two fetches). We therefore restricted our code space to the options provided in Table 7.2.

Table 7.2: Choice of Error Correcting Codes for CME

ECC scheme	Length of ECC word (original message + ECC bits)
SECDED (Single Error Correcting, Double Error Detecting) [127]	36 (29 + 7) 72 (64 + 8) 144 (135 + 9)
DECTED (Double Error Correcting, Triple Error Detecting) [201]	36 (23 + 13) 72 (57 + 15) 144 (127 + 17)
3EC4ED (3 Error Correcting, 4 Error Detecting) [202]	36 (17 + 19) 72 (50 + 22)

The reason for using code word sizes of 36-bits and 144-bits alongside 72-bits is to increase the opportunity for stronger protection. As an example, let's assume a compressed message is 58 bits long, therefore splitting the message to two 29-bit messages can enable SECDED protection (29+7) on each 29-bit message and two code words can constitute a 72-bit fetch. If we had restricted our code space to 72-bit words, the 58 bits of the two messages will only get SECDED protection since DECTED protection will require the message to be of length 57 bits. It can be noted from Table 7.2 that, no code stronger than 3EC4ED (3 error correcting, 4 error detecting) has been used even for cases where stronger protection is possible (for eg. 4EC5ED can be used in cases where the compressed cache line size is less than 368 bits). This is because stronger ECC not only adds greater hardware complexity and overheads, the redundant bits added to each word in the

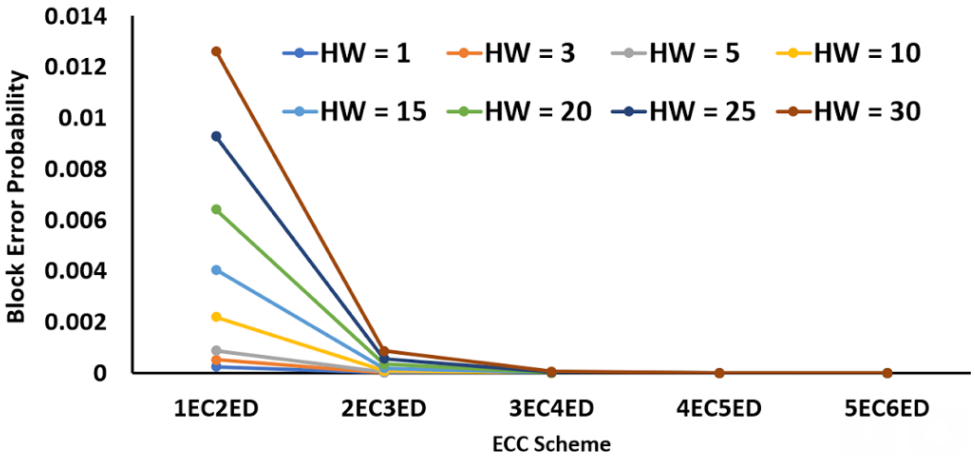


Figure 7.5: Block failure probability is shown for blocks with different Hamming weight (HW) and ECC schemes. The probability of 1→0 bit-flip is considered to be 10^{-5}

cache line often increases the Hamming weight of the overall word considerably, thus increasing chances of read disturb/write errors. As seen in Figure 7.5, with stronger ECC, the block failure probability decreases rapidly till 3EC4ED, beyond which the benefit of stronger ECC saturates. 144-bit 3EC4ED was not evaluated because of the large size of the encoder/decoder alongside the large number of redundancy bits.

We also wanted to limit our tag overhead per cache line to 8. If the tag is protected by SECDED, then 4 bits of redundancy would be required. Thus the actual tag can be at most 4 bits. The first bit would be used to denote if the cache line is compressed or not. With 3 bits of tag left to denote the ECC scheme used on a compressed cache line, there can be eight distinct codes that can be used. We decided to use 2x and 4x duplication for any 512-bit cache line that gets compressed beyond 256-bits or 128-bits respectively. Therefore, cache lines with compressed length higher than 256 are left with six choices of codes.

Unlike [189] which uses duplication to avoid reading the entire line (which in standard DDR memories actually does not help with read disturb as explained earlier), we use it to minimize uncorrectable errors. As observed from our SPEC2006 benchmark traces, the average number of read operations between two write operations is less than two. Thus, when the data is duplicated once, a cache line word would fail only when both copies of the word have un-correctable errors (note that all used ECC codes detect more errors than they can correct). Therefore, in our scheme,

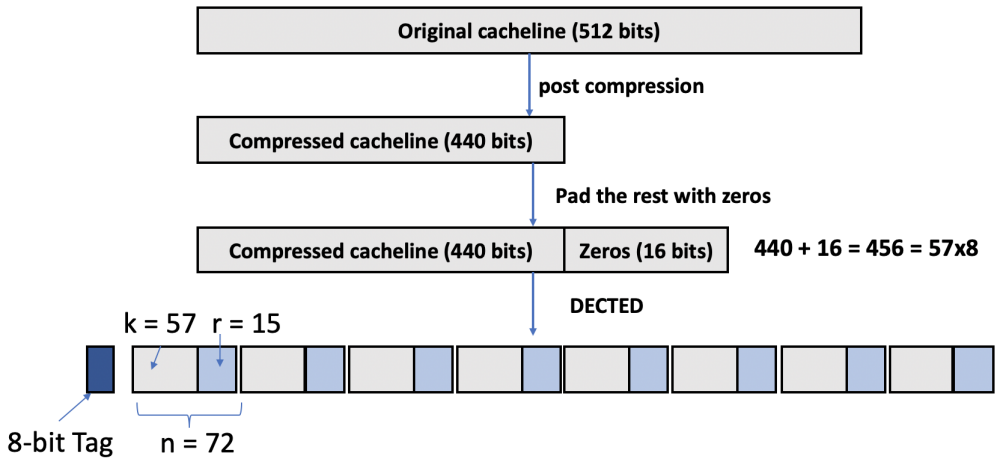


Figure 7.6: An example of CME scheme where the compressed cache line size is 440 bits

we read the entire cache line (with all the copies) and only use the subsequent copies if there is an error in the first copy. Reduction in errors means scrubbing (refresh with ECC) operations will be needed less frequently, which saves both time and energy. If the compressed size is less than $1/4^{th}$ of the original size, we propose a 4x duplication for even stronger protection against errors.

Since all the cache lines within a certain range of compressed length get the same type of protection, majority of those cache lines would have to be padded with zeros at the end, before adding ECC, to increase the final message length to a certain value required by that particular code type. For example (shown in Figure 7.6), if all the cache lines whose final compressed length lies between 416 and 456 get (72,57) DECTED protection, all cache lines within that range whose length is not exactly equal to 456 would need to be padded with zeros to increase the final length to 456 before splitting up the cache line into eight 57-bit words and adding DECTED protection on each one of them.

Based on this we realized that for any range, most of the cache lines would have the last few words being all 0s. This can be seen in Figure 7.7 where we show the hamming weight of each 32-bit word of compressed cache lines. Based on the compressed cache line size, we bin the cache lines with post-compression size in between 256 to 512 bits in to four buckets. Each plot in the figure corresponds to one bucket and shows the distribution of average hamming weight of the 32-bit words across six benchmarks from the SPEC2006 suite. Each of these 6 benchmarks had

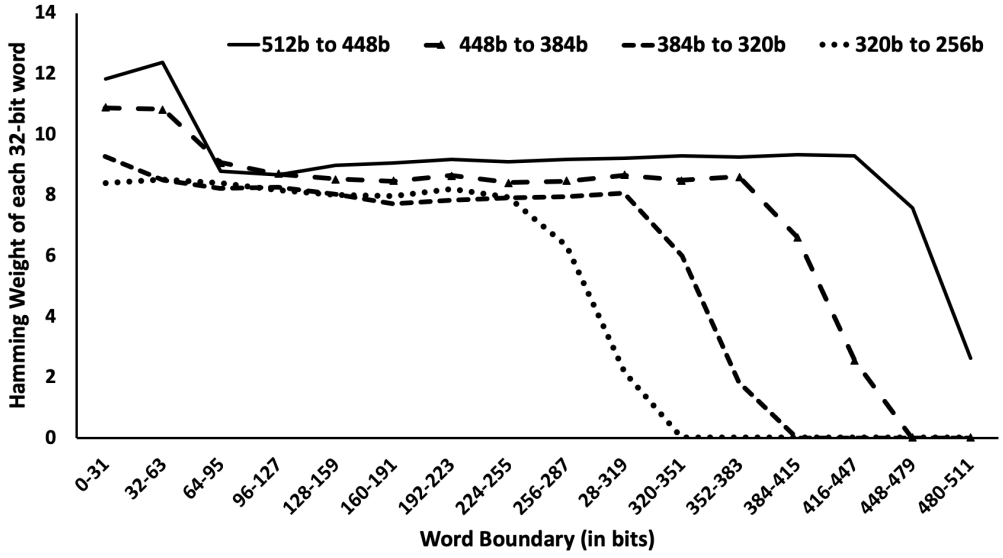


Figure 7.7: Average hamming weight of each 32-bit word of all cache lines within each bucket. Uniform bucket size of 64 bits were used for all cache lines whose final size lies between 512 bits and 256 bits.

widely different average compression ratio (original-size/compressed-size) and hamming weight. Hence it can be argued that these six benchmarks are good representations of the entire suite.

It can be seen that in most cases, the last few words have much less average hamming weight as compared to the rest. This is because majority of those words have all 0s. Thus, we decided to add split codes to our code space. In split codes, the same cache line would have two different types of protection. The first few words would have stronger protection than the last few. In order to limit the code space, we decided to evaluate combinations of at most two codes from Table 7.2 and limit the use of weaker protection up till the fourth word from the end. To further limit the code space, after listing down all the choices, we chose the strongest code within a redundancy range of 4 bits. This means that if by adding 4 extra bits we would get a strictly stronger code, we would remove the weaker one from our final evaluation. Overall, after all these manipulations, we were left with 28 codes (from the possible 64 code pairs) from which we would have to select the best six codes. Next, we discuss how we select these best six codes which constitutes CME.

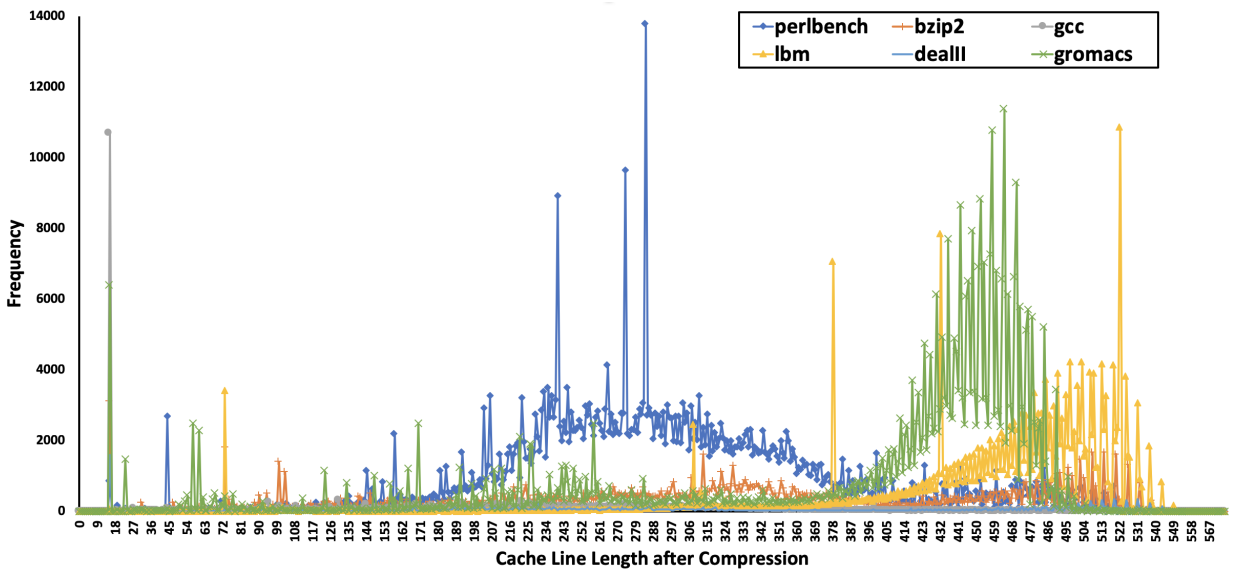


Figure 7.8: Distribution of cache line length after compression of six benchmarks from the SPEC2006 suite

7.3.3.2 Dynamic Programming to choose the final set of codes

To select the best six codes, we took into account the final distribution of cache line length after compression and the average hamming weight of each word for all cache lines within a certain range of compressed length across the SPEC-2006 benchmark suite. The same six benchmarks mentioned previously were also used for this. The cache line length distribution of the six benchmarks can be seen in Figure 7.8.

Based on these distributions, we used dynamic programming to choose the six optimal codes.

For each code, the maximum compressed length of a cache line for being protected by that code is fixed. For example, for each of the 8 words in a cache line to have DECTED protection, the compressed cache line can be at most 456 bits. This maximum length is fixed for each type of code.

For every code choice, we first find the weight distribution of all cache lines whose compressed length is within the range (`max_length`, 256), where `max_length` is the maximum allowable length for that particular code. We use this weight distribution to calculate the block failure probability if this code was selected for these cache lines. The failure probability is then weighted by the fraction of cache lines that fall within this range to evaluate the effectiveness of adding the code.

For the dynamic programming, we start with the code (say codeA) which has the smallest max_length and calculate the block failure probability over the cache lines of compressed size (max_lengthA, 256). Next, we add the second code (say codeB) with the next smallest (max_length). We then evaluate two cases, (1) When codeA and codeB are both used together to protect cache lines between (max_lengthA, 256) and (max_lengthB, max_lengthA) respectively; calculating this joint probability can leverage the block probability for (max_lengthA, 256) that was calculated in the last step and (2) When only codeB is used.

In subsequent iterations, we add new code types and calculate the combined block failure probability by leveraging the already calculated block failure probabilities for smaller max_length codes. From the seventh iteration onwards, we only calculate the probabilities of code combinations which has six codes in total. Since most of the code combinations except the newly added code has been evaluated in previous iterations, the dynamic programming approach helps us minimize the time to calculate the block failure for a set of six codes. After all the six code combinations are iterated through, we choose the one with the smallest aggregate block failure probability. The set of codes that are finally chosen are given in the Table 7.3.

Note that this dynamic programming based code space search is done just once to find the best six codes which constitutes CME. A fixed CME scheme would be built in to the hardware and all the applications would use the same scheme.

7.3.4 Additional Tag Bits and Memory Organization

Every cache line now needs additional tag bits to denote if the cache line is compressed and what protection scheme is used.

As shown in Table 7.4, we use 8 additional bits of tag to each cache line to denote the transformation operation that was done for that particular cache line.

- *Bit0:* Denotes if the stored cache line has been compressed or not. If compressed then the first bit of the tag is ‘1’; else ‘0’.
- *Bits1-3:* When the cache line is compressed, these three additional bits denote the ECC/duplication

Table 7.3: ECC scheme to be used depending on the compressed cache line size

Length of compressed cache line (in bits)	**ECC scheme to be used**
>512	No compression (Use Raw Cache line + (72,64)SECDED)
≤512 and >508	(72,64)SECDED on each 64-bit cache line word
≤508 and >495	(144,127)DECTED on each 127-bit cache line word
≤495 and >482	(72,57)DECTED on the first 2 57-bit cache line words and (144,127)DECTED on the last 3 127-bit cache line words
≤482 and >469	(72,57)DECTED on the first 4 57-bit cache line words and (144,127)DECTED on the last 2 127-bit cache line words
≤469 and >427	(72,57)DECTED on the first 6 57-bit cache line words and (144,127)DECTED on the last 127-bit cache line word
≤427 and >256	(72,50)3EC4ED on the first 6 50-bit cache line words and (144,127)DECTED on the last 127-bit cache line word
≤256 and >128	Duplicate the cache line (2 copies) and (72,64) SECDED on each word
≤128	Duplicate the cache line (4 copies) and (72,64) SECDED on each word

scheme used for that cache line as given in Table 7.4, else the field is populated with '000'. Note that the tag bits for the duplication cases (last two) are the highest weighted since they are the least frequently occurring.

- *Bits4-7:* These 4-bits are ECC bits used to provide a (8,4) SECDED protection on the first 4-bits of tag to correct a single-bit error and detect any double-bit error.

7.3.4.1 DDR4 Primer

In today's system with DRAM based main memory system, a processor accesses the main memory through the memory controller. Memory controller buffers memory access requests from the processor, schedules the requests, converts them into DRAM commands complying with the specific DDR protocol and sends them over a DDR bus to the dual in-line memory module (DIMM). One

Table 7.4: 8-bit Tag per Cache Line for CME

Tag Bits	**When Compression is possible**		**No Compression**
Bit-0	-	‘1’	‘0’
Bits1-3	BPC + (72,64)SECDED	‘000’	‘000’
	BPC + (144,127) DECTED	‘001’	
	BPC + (72,57) DECTED on first two words and (144,127) DECTED on the rest	‘010’	
	BPC + (72,57) DECTED on first four words and (144,127) DECTED on the rest	‘011’	
	BPC + (72,57) DECTED on first six words and (144,127) DECTED on the rest	‘100’	
	BPC + (72,50) 3EC4ED on first six words and (144,127) DECTED on the rest	‘101’	
	BPC + duplication (2copies) + (72,64)SECDED	‘110’	
	BPC + duplication (4copies) + (72,64)SECDED	‘111’	
Bits4-7	(8,4)SECDED redundancy for the first 4 Tag bits	ECC for Bits 0-3	‘0000’

or more DIMMs is supported on a memory bus. Each DIMM has a DIMM controller along with 9/18/36 x4/x8 DRAM memory chips. The DIMM controller acts as the interface to the DDR bus and manages the DIMM. In our example we use a DIMM with 9 x8 memory chips as the baseline to explain the modifications required to fit an additional STT-RAM tag chip along with 9 x8 STT-RAM chips on a STT-RAM DIMM connected to a DDR4 bus.

In a conventional DRAM based main memory system with x8 DRAM DIMM, for reading from or writing a cache line to the memory, each memory chip from the same rank in the DIMM sends 64-bits over 8 cycles to form a 512-bit (576-bits with ECC) cache line. First, the memory controller sends an ACTIVATE request to the DIMM along with the rank, bank and row addresses. Based on the addresses the DIMM controller activates the row in the bank of all the chips in that rank. The row is read from the array into the respective row buffers. Each row buffer now holds one row, i.e.,

an entire DRAM page. If it is DDR4 type memory one DRAM page size of an x8 memory chip is equal to 1KB. From this 1KB page per DRAM chip, only 64-bits need to be accessed sequentially in 8 bursts where each burst consists of 8-bits. The beginning of this 64-bit chunk in the 1KB page is determined by the column address sent to the DIMM by the memory controller next along with the READ/WRITE command. For a READ operation the last three bits of the column address determines the burst order, i.e., the order in which the cache line words will be read. This is to enable "priority word first" [203] for improving performance where the payload word gets read and decoded first and sent to the processor while the rest of the words are brought to the cache. The rest of the bits in the column address determine the beginning of the chunk being accessed.

Assuming the STT-RAM based main memory system uses DDR4 protocol, we explore two ways of storing the additional bits of tag per cache line.

7.3.4.2 Scheme 1

The first option is to store the tag bits separately in the memory. We require an extra tag chip to store the 8 bits of tag per cache line and an additional data signal in the DIMM. This extra tag chip will be a x1 memory chip with one data signal pin. The size of the memory page will be one-eighth that of the other memory chips in the DIMM (128B) and in each burst only one bit will be read instead of the conventional 8-bit burst. Thus, when the DIMM controller sends the READ/WRITE request to the tag chip, it will shift the column address bits by 3 (divide by 8). The data signal pin of this x1 tag chip will be connected to the extra data signal in the DIMM. The minor changes required to accommodate this extra tag chip are depicted in Figure 7.9.

Though implementing this only requires subtle changes to the DIMM and memory architecture, this scheme incurs latency overhead. In today's systems where the main memory has (72,64) SECDED protection, each 72-bit word read from the memory in a burst has to go through the ECC decoder to get the original 64-bit message and to check for any single-bit/double-bit errors. Since a 576-bit cache line (512-bits of message + 64 bits of ECC) is sent over 8 bursts, each of length 72-bits, and each burst is a separate ECC word, the ECC decoding can begin as soon as the first burst or ECC word arrives. However, in our case we have to wait for the tag bits before we can start

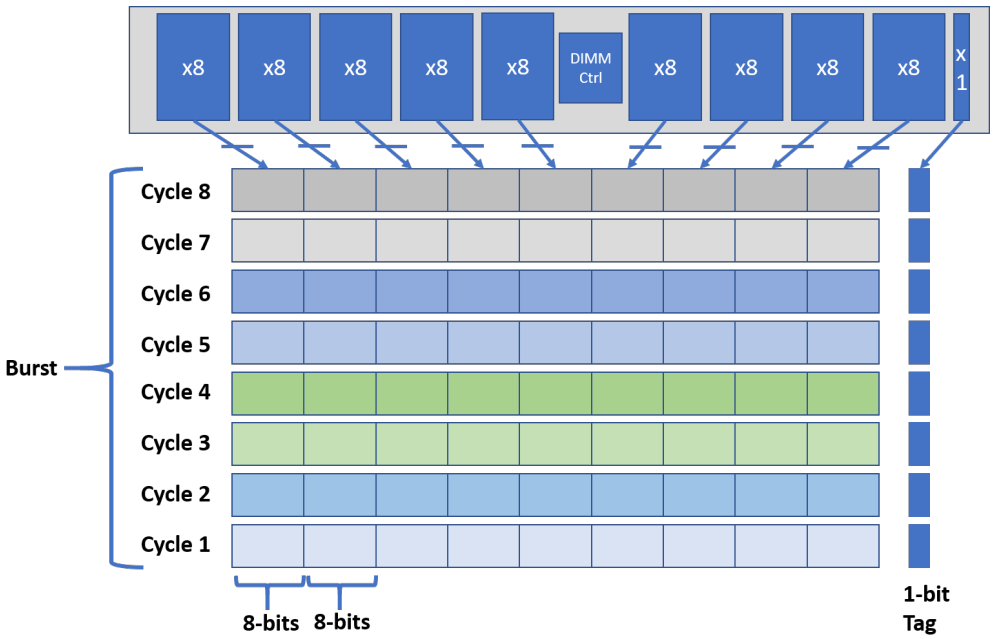


Figure 7.9: CME-Scheme 1 is shown where tag bits are stored in an x1 DRAM chip. One tag bit is read every cycle in burst. Different colors represent different 72-bit ECC words in a 512-bit cache line.

with the ECC decoding and in every burst only one bit of tag is sent. This latency overhead will vary from cache line to cache line. For an uncompressed cache line, as soon as the first bit of tag arrives, the ECC decoding can start since all un-compressed cache lines are protected by (72,64)SECDED code. However, if the cache line is compressed, i.e., if the first tag bit is '1', the ECC decoding has to wait for at least 4 cycles. Since the tag is protected by a linear (8,4) SECDED code, the first 4 bits of tag are the original tag message and the last 4 bits are the redundancy bits. So, as long as there is no error in the tag bits, the first 4 bits of tag should be sufficient to tell us the ECC scheme used for that particular cache line. Thus, the ECC decoding can start after 4 cycles. If after 8 cycles it turns out that there was an error in the first 4 bits of tag then the ECC decoding has to be done again, causing a 8-cycle latency overhead, however probability of that happening is very small and will not impact the performance of a system.

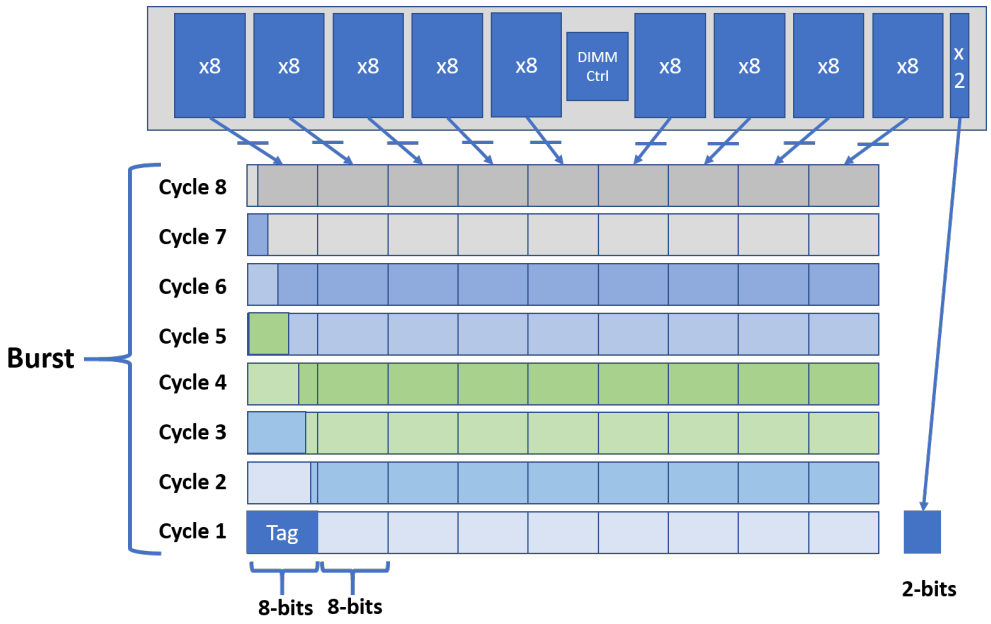


Figure 7.10: CME-Scheme 2 where tag bits for ECC scheme used are stored in an x8 DRAM. The tag bit and it's parity representing compression are stored in an x2 DRAM chip and are brought in the same burst. Different colors represent different 72-bit ECC words in a 512-bit cache line.

7.3.4.3 Scheme 2

To avoid this latency overhead, we propose an alternative option of embedding the 8 bits of tag into the compressed cache line itself. This means that after encoding the compressed cache line with ECC, the final length has to be 568 bits so that adding 8 bits of tag to the cache line increases it to 576 bits (as shown in Figure 7.10). If the cache line cannot be compressed at all no tag bits will be needed as the standard (72,64) SECDED code will be used for all uncompressed cache lines. However, overall one bit of tag is still required separately to denote if the cache line has been compressed. The one extra bit of tag can either be stored in the memory controller for a small sized main memory system or a separate x2 memory chip in the DIMM can be used. If stored in the memory then an additional bit of parity is required to protect that one tag bit (1 bit is enough since errros are asymmetric). This tag bit can be fetched using a x2 memory chip in one burst and will have a page size of 4B. For the x2 memory chip, the least-significant 5-bits of column address select need to be ignored because of the reduced page size and no requirement of bursts. Reduction of the burst size will be minimal modifications of the circuitry that is present in today's DDR4 DIMM

which supports variable burst sizes of 4 and 8. The DIMM will require 2 additional data signal pins and the memory controller needs to know that valid data will be sent over those 2 data signal pins only in the first burst. All these changes should be minimal and easy to implement given the current existing architectures. This scheme will have latency overhead of only 1 cycle. However, having this option will require our ECC schemes to change since now the final size of the ECC encoded message will be different. We re-ran our code selector to select the six optimal schemes and the selected codes are provided in Table 7.5. We call this CME Scheme-2 when we evaluate it in Section 7.5.2.

Table 7.5: ECC scheme to be used depending on the compressed cache line size when the tag is embedded in the cache line (CME-Scheme 2)

Length of compressed cache line (in bits)	**ECC scheme to be used**
>504	No compression (Use Raw Cache line + (72,64)SECDED)
≤ 504 and >500	(71,63)SECDED on each 63-bit cache line word
≤ 500 and >487	(142,125)DECTED on each 125-bit cache line word
≤ 487 and >474	(71,56)DECTED on the first 2 56-bit cache line words and (142,125)DECTED on the last 3 125-bit cache line words
≤ 474 and >461	(71,56)DECTED on the first 4 56-bit cache line words and (142,125)DECTED on the last 2 125-bit cache line words
≤ 461 and >419	(71,56)DECTED on the first 6 56-bit cache line words and (142,125)DECTED on the last 125-bit cache line word
≤ 419 and >252	(71,49)3EC4ED on the first 6 49-bit cache line words and (142,125)DECTED on the last 125-bit cache line word
≤ 252 and >126	Duplicate the cache line (2 copies) and (71,63) SECDED on each word
≤ 126	Duplicate the cache line (4 copies) and (71,63) SECDED on each word

If the STT-RAM based memory is embedded and on-chip (as seen in some commercial STT-RAM based memory systems [204]), the tag placement will be much simpler and will not incur any additional latency overhead since all 8 bits can be fetched in one cycle.

7.4 Evaluation Methodology

While compression and error correcting codes are used individually in caches and main memory, we focus on combining the two and opportunistically providing stronger protection for STT-RAM (or any other magnetic or high error rate memory) based main memory systems in this chapter. We first compare the hamming weight reduction achieved after doing Bit-Plane Compression with and without hamming-weight-aware inversion against another scheme [183] proposed earlier to reduce hamming weight and an alternative compression scheme (BΔI [134]). We then evaluate the effectiveness of using stronger codes for STT-RAM based memories with error rates provided in Table 7.6. We use two design points, one from Intel [2] and the other from Samsung [3], for our analysis. For the Intel design point, the read/write/retention error rates are as provided. For the Samsung design point, the write error rate is provided along with the thermal stability factor (Δ). Based on the technology node, the read disturb rate is calculated from [174]. The retention error rate is calculated based on the Δ using Equation 7.1.

$$P_{ret} = 1 - e^{-\frac{t_r}{\tau_0} e^{-\Delta}} \tag{7.1}$$

where τ_0 is reversal attempt period and is on the order of a nanosecond [187], and t_r is the interval for which the evaluation is conducted.

For STT-RAM based memories, refresh is not the same as DRAM [174, 187]. In DRAM, refresh is used to prevent deterministic errors cause by charge leakage over time. But in STT-MRAM, errors are stochastic in nature. Therefore, if DRAM like refresh is performed in STT-RAM, where the cell contents are simply read and written back without any error correction, already flipped bits in the memory would be read and written back, as is, without any correction. This would not be effective in lowering the error rate. In STT-RAM, refresh needs to be accompanied by error correction and is similar to the scrubbing operations performed in today's systems. For all our analysis, unless otherwise mentioned, we consider a scrubbing interval of one second. We finally evaluate the hardware overhead of compression with multi-ecc and its impact on performance.

To evaluate our protection scheme for STT-RAMs, we extracted memory traces of 18 benchmarks from the SPEC CPU2006 benchmark suite using Gem5 [138]. These applications are a mix of

Table 7.6: Evaluation setup

Cache Line Size	**512-bits (64-Byte)**	
Design Point - I [2]	Write Bit Error Rate	1×10^{-6}
	Read Bit Error Rate	1×10^{-12}
	Retention Error Rate	Negligible (200C 10 years)
Design Point - II [3]	Write Bit Error Rate	1×10^{-6}
	Read Bit Error Rate	1×10^{-10}
	Retention Error Rate	Calculated based on thermal stability factor Δ = 40 using equation 7.1

integer and floating point benchmarks. Next, each application was subjected to CME. The average block failure probability (average failure probability of each word in the cache line) was computed for each design point based on the final set of cache lines obtained after applying compression to the obtained memory traces.

The probability of a cache block/word of hamming weight W not failing under a certain write/read/retention bit error rate P_{ER} protected by a t-error correction ECC is given by the following equation:

$$P_{block} = \sum_{i=0}^{t} \binom{W}{i} (1-P_{ER})^{W-i} (P_{ER})^{i} \tag{7.2}$$

For overall block error rate, we calculated the probability of failure using the knowledge obtained from the memory traces about the number of reads between two consecutive write/scrubbing instructions to a a particular memory address. For example, when a cache line protected using (72,57) DECTED code is read twice consecutively before a write operation to the same address, the probability of no fault is calculated by considering all the following cases: (a) When two or less faults occur during the same operation (either during write, any of the two reads or because of retention error). (b) One fault occurs during one operation and another fault occurs during another operation. When a cache line gets duplicated twice, we consider a block to be failing only when both copies have un-correctable errors. Based on our memory trace statistics, we saw that most of the cache lines are read once. However, there are still 2-5% of cache lines that are read more than

twice (some even more than 10 times).

To evaluate the area and latency overheads of stronger ECC codes, we synthesized both (72,64) SECDED and (144,127) DECTED decoding engines using an industrial 45nm library. The (144,127)DECTED decoder is expected to have the largest overheads compared to the baseline (72,64) SECDED. To evaluate the performance impact of CME due to the overheads of compression and multi-ECC (discussed in detail in Section 7.5.4), we ran performance simulation using Gem5 [138]. Two micro-architectural configurations were evaluated as provided in Table 7.7. The first set of evaluations were done for a system with 8 in-order (InO) cores sharing only 2MB of unified L2 cache and no prefetch. So that there is minimal memory access latency hiding techniques. Also the shared L2 cache is not statically partitioned and the different processes compete for cache space. These are expected to exaggerate the effect of CME overheads on performance. For these experiments, each thread runs the same application in separate processes. The second set of evaluation was done for a single out-of-order (OoO) core having a unified 2MB L2 cache and with prefetching enabled. The performance evaluations on Gem5 were done for the benchmarks in the SPEC2006 suite, fast forwarding for 1 billion instructions and executing for 2 billion instructions. The latency overheads considered for the simulations are discussed in detail in Section 7.5.4.

7.5 Results

In this section we demonstrate that CME provides considerable benefit in terms of block error reduction as compared to a normal (72,64) SECDED code. The evaluations also show the minimal impact of the hardware overheads of CME on performance.

7.5.1 Reduction in Hamming Weight

We evaluate the hamming weight reduction when using Bit Plane Compression with and without hamming-weight-aware inversion scheme and compare it against a previously proposed Dynamic-XOR scheme [183] where the goal was to solely minimize the weight of each cache line. We also included another popularly used cache line compression scheme (BΔI) [134] for this hamming

Table 7.7: Core Micro-architectural Parameters

	Config-1	**Config-2**
Cores	8, InO (@ 2GHz)	1, OoO (@ 2GHz)
ISA	ALPHA	x86
L1 Cache per core	32KB I$ 32KB D$ 4-way	32KB I$ 32KB D$ 4-way
L2 Cache	2MB (shared, unified) 8-way	2MB (unified) 8-way
Cache Line Size	64B	64B
Memory Configuration	32GB of single-channel x4 DDR4-2400	32GB of single-channel x4 DDR4-2400
Nominal Voltage	1V	1V

weight analysis. From Figure 7.11 it can be seen that for all applications both the compression schemes and the Dynamic-XOR scheme reduce hamming weight of cache line as compared to the original weight. BPC without inversion reduces hamming weight by upto 67.3% (avg. 21.3%) compared to the original weight. Adding inversion reduces the hamming weight further (on an avg. 30% compared to the original weight). For most applications, cache line after BPC with or without inversion ends up with a lower hamming weight than Dynamic XOR. On an average BPC with inversion has 16.8% lower weight than the Dynamic-XOR scheme. Thus, BPC not only has the advantage of reducing cache line size over Dynamic-XOR which, in turn, allows for stronger ECC, it also reduces hamming weight of the entire cache line, thus reducing chances of unwanted bit flips during write and read operations in STT-RAMs. On top of BPC, hamming weight aware inversion coding further reduces hamming weight by an additional $\sim$8%. BΔI, on an average, performs better than BPC with inversion in the matter of reducing hamming weight of cache lines of most applications. BPC, however, outperforms BΔI in reducing block failure probability and has been discussed in detail in Section 7.6.1. In a recent work [205], the authors propose to use stronger ECC for cache lines whose hamming weight is above a certain threshold and use weaker ECC otherwise.

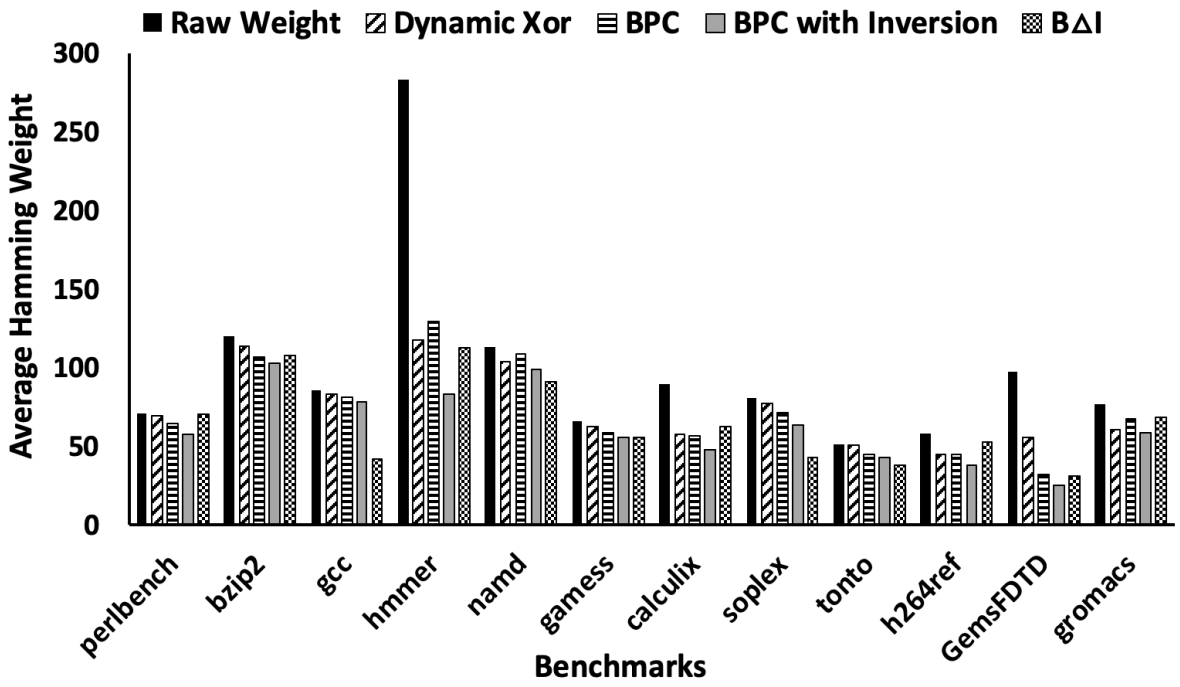


Figure 7.11: Comparison of average Hamming weight of original cache line, BPC, BΔI and DBX schemes

For this work, the stronger ECC scheme is (72,64) SECDED on each 64-bit cache line word while the weaker ECC is (523,512)SECDED on the entire cache line. In our baseline case, every cache line, irrespective of its hamming weight, gets the stronger protection used in [205] and with CME, the cache lines gets even stronger protection.

7.5.2 Reduction in block failure probability

To evaluate the reduction in block errors, block failure probability is computed per application for all words in all cache lines retrieved from the memory traces for the following three cases:

- *Baseline:* No Compression and each 64-bit cache line word gets (72,64)SECDED protection.
- *Scheme-1:* Compression (with and without hamming-weight-aware inversion) with Multi-ECC protection scheme where the tag bits are separate.
- *Scheme-2:* Compression (with and without hamming-weight-aware inversion) with Multi-

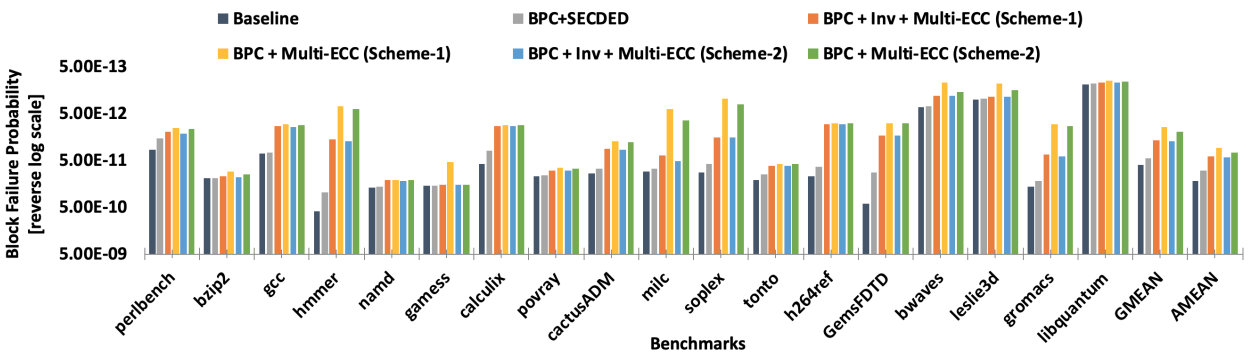


Figure 7.12: Reduction in block failure probability induced due to write/read/retention errors for the first design point [2] is shown. The y-axis is in logarithmic scale (reverse order). The geometric mean and arithmetic mean of the improvement of CME Schemes over baseline is shown in plot.

ECC protection scheme where the tag bits are embedded into the cache line.

. As mentioned before, both schemes were compared for two design points with different read disturb/write/retention error rates. The results are shown in Figures 7.12 and 7.13. Please note that the y-axes are in logarithmic scale (reverse order). This means that the taller the bar, the smaller is the block failure probability (better it is). Also in both CME schemes, every in-compressible cache line gets the baseline (72,64) SECDED protection per word.

The Compression (with and without inversion) with Multi-ECC (CME) protection scheme is compared against the baseline case where each 64-bit cache line word gets (72,64) SECDED protection. Compared to the baseline, Scheme-1 has 8 extra bits per cache line and Scheme-2 has 2 extra bits. The compression scheme used is the modified version of BPC explained in Section 7.3.2. We first analyzed the improvement in block failure probability that comes only because of hamming weight reduction by BPC. To do that we computed the reduction in block failure probability when there is BPC alone. The compressed cache lines were padded with zero to increase the final size to 512 and then every 64-bit chunk was provided with (72,64) SECDED protection. For the first design point, BPC with (72,64) SECDED alone (without inversion and multi-bit error correction) reduced block failure probability by at most 4x (average 1.38x). Similarly, for the second design point, the maximum improvement was 4.61x (average 1.35x). These improvements are negligible.

The improvements in block failure probability come mostly from the opportunistic stronger

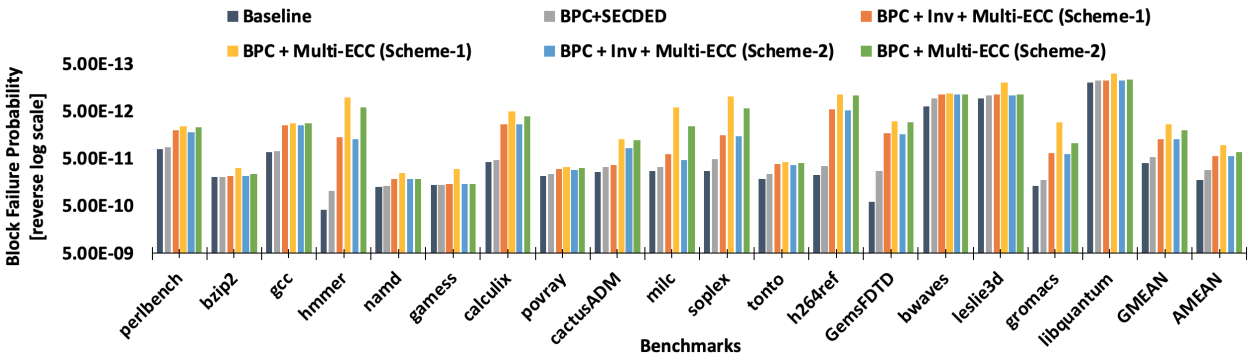


Figure 7.13: Reduction in block failure probability induced due to write/read/retention errors for the second design point [3] is shown. The y-axis is in logarithmic scale (reverse order). The geometric mean and arithmetic mean of the improvement of CME Schemes over baseline is shown in plot.

protection that is added to the cache lines. For the first design point, BPC with Multi-ECC (without inversion) reduces block failure probability by as much as 176x (average 6.18x) and 150x (average 5.11x) for Scheme-1 and Scheme-2 respectively when compared to the baseline. For the second design point, BPC with Multi-ECC (without inversion) reduces block failure probability by as much as 240x (average 6.81x) and 148x (average 5x) for Scheme-1 and Scheme-2 respectively. In fact, successful compression and, therefore, stronger protection is possible for 70.5% cache lines across the benchmarks used. As a result, CME performs dramatically better than the baseline case as well as the case with only compression and SECDED protection. For the baseline case, the benchmark with the worst block failure probability is hmmer. If the entire suite is considered, the reliability of the system would be limited by this benchmark for both design points. With Compression and Multi-ECC, this benchmark, however, has the highest compression ratio and the maximum reduction in the average hamming weight of cache lines. Therefore, it ends up with the maximum reduction in block failure probability. The two benchmarks with the lowest reductions are namd and libquantum. This is because these two benchmarks are the least compressible and hence, most of the cache lines end up with the same (72,64) SECDED protection as the baseline. Across the entire SPEC suite, the maximum block failure probability with the baseline protection scheme is reduced by 5x and 6x when CME Scheme-1 is used for the two design points respectively.

An unexpected result is observed when hamming-weight-aware inversion is used along with BPC. Since inversion helps to reduce the hamming weight of the cache lines, it is expected to reduce the block failure probability beyond what compression without inversion and multi-ecc can achieve. However, for both design points and both schemes it can be seen from the figures that across majority of the benchmarks, the block failure probability is higher with inversion than what it is without inversion. With inversion, CME reduces block failure probability by at most 35x (average 3.2x) compared to the baseline for the first design point (using Multi-ECC Scheme-1). This is significantly lower than the corresponding 6.18x average improvement that was achieved without inversion. Though inversion reduces average hamming weight across all benchmarks (see Figure 7.11), it adds extra bits to the cache lines because each BPC word requires one extra bit to know if the word has been inverted. This can add as many as 35 extra bits to a cache line. As a result, a lot of cache lines end up getting weaker protection. To better utilize the reduction in hamming weight without increasing the cache line size significantly, hamming-weight-aware inversion can be done on groups of multiple BPC words instead of doing inversion on each BPC word. We analyze one such case for design point 2 using Multi-ECC Scheme-1. It is seen that when groups of 3 BPC words are used, BPC with inversion and Multi-ECC (average reduction in block failure probability is 6.14x) outperforms the case without inversion for multiple benchmarks.

To compare against single strong code scheme such as FrugalECC [197], we further evaluate the case of having one stronger code (72,57) DECTED for all cache lines that could be compressed, irrespective of their final size, along with (72,64) SECDED for the uncompressed ones and compared this against CME Schemes. It is seen in Figure 7.14 that FrugalECC like scheme can achieve at most 34x reduction in block failure probability compared to the baseline while CME Schemes can achieve at much as 240x reduction. For all benchmarks, CME Schemes-1 and 2 perform better than the single strong code scheme.

7.5.3 Hardware Overhead of Multi-ECC Scheme

The CME scheme requires support for multiple ECC engines (SECDED, DECTED and 3EC4ED). Having multiple ECC encoders and decoders on a memory controller on chip can be costly in terms

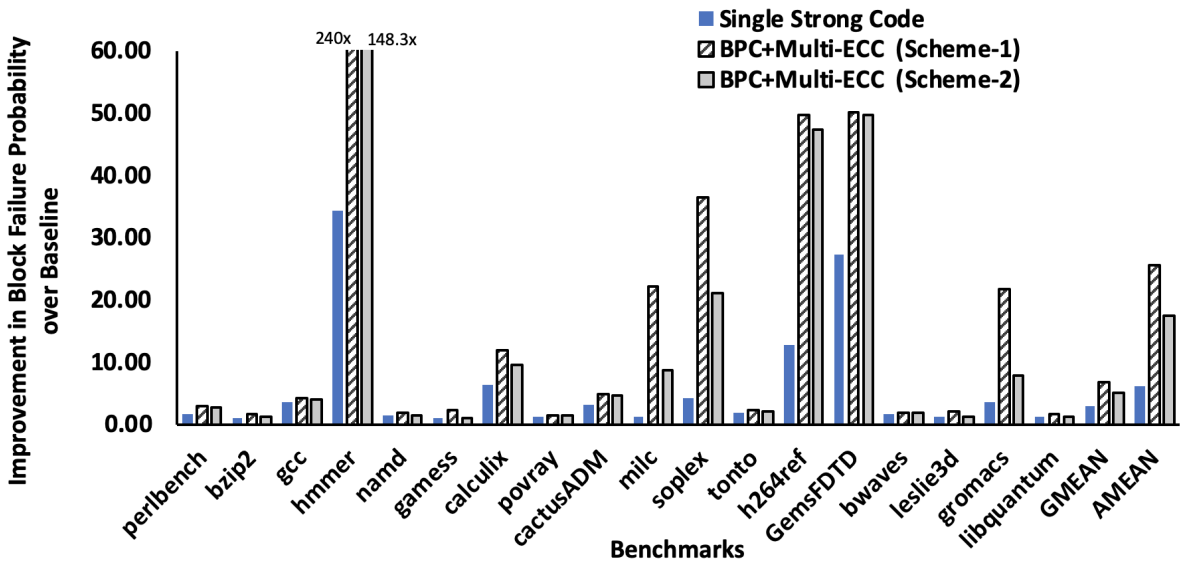


Figure 7.14: Improvement of CME Schemes 1 and 2 over a scheme that provides uniform (72,57) DECTED for all compressible cache lines and (72,64) SECDED if in-compressible.

of both area as well as power. However, if asymmetric quantum BCH coding [206] is used, G and H matrices for a smaller ECC (for e.g., DECTED) can be composed out of sub-matrices of G and H matrices of a stronger ECC scheme (for eg. 3EC4ED), therefore the same hardware can be reused. Since in our case, the total codeword length is same for all the cases (n=72 bits), eliminating rows from the bottom of the H-matrix of a stronger code (for e.g., 3EC4ED) would generate the H-matrix of a weaker code (for e.g., DECTED) with the same codeword length. However, for encoding using G-matrix, the rows (=k) of the G-matrix decrease as we move towards a stronger ECC code for a given constant codeword length (=n). Also, (72,57) DECTED H-matrix can re-use a (144,127) DECTED H-matrix. Thus, the largest decoder would be required for the (144,127) DECTED and the other BCH based codes ((72,50) 3EC4ED and (72,57) DECTED) can re-use most of the decoder. (72,64) SECDED code would require a separate decoder. However, even the baseline (72,64) SECDED protection scheme would require the same parity check engine. Synthesizing the parity check engines of (144, 127) DECTED using an industrial 45nm library results in about only about 8700 μm^2 of additional area overhead compared to a only SECDED implementation as in the baseline. Moreover, since only one word is decoded at a time, reuse isn't expected to have any performance degradation.

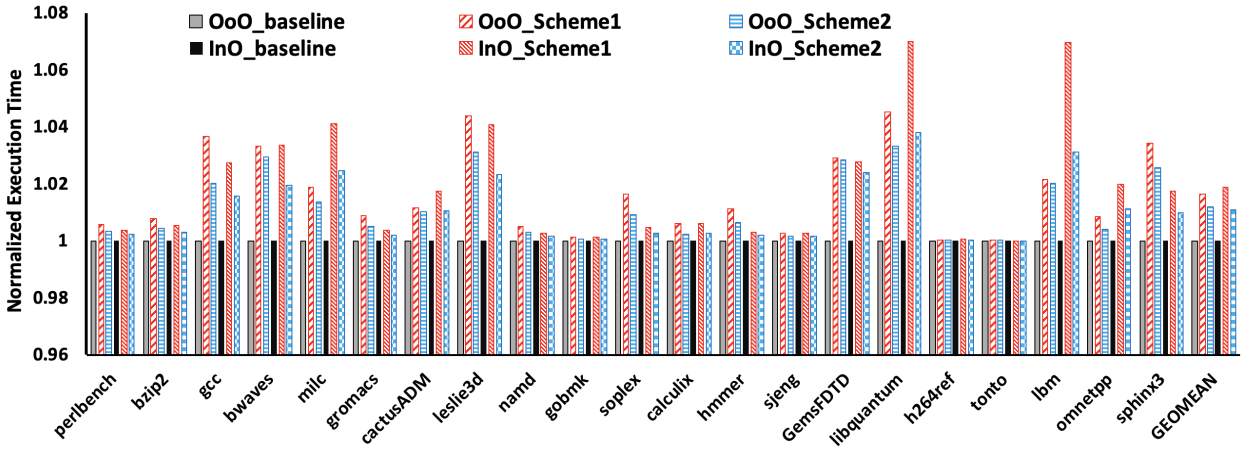


Figure 7.15: Comparing Normalized Execution Time of two systems (one with 8 InO cores and another with a single OoO core), both having three protection schemes: baseline (72,64)SECDED, CME Scheme-1 and CME Scheme-2. InO and OoO results are normalized to their respective baselines.

As reported in [135], the area overhead of BPC compression engine is about 48000 μm^2 when synthesized using 40nm TSMC standard cells. The total area overhead of BPC and ECC encoding/decoding engines is not significant when compared to the currently available large sizes of processors. The per-bit energy overhead of BPC is less than 1% of per-bit STT-RAM read energy [207]. The latency overhead of CME and the possible impact of system performance is evaluated next.

7.5.4 System Performance Evaluation

As mentioned in [135], Bit-Plane compression takes 7 cycles for compression/decompression. Since in our case each DBX word is less than half their size, we should be able to fit in twice the number of DBX encoders in the same area. As a result, we would be able to complete the encoding step (last step in BPC) in 2 cycles instead of 4. Thus, we would require 5 cycles for BPC compression/decompression. The latency overhead of fetching the tag bits from the memory would be different for the two CME Schemes.

In Scheme-1, each bit of tag is fetched in each cycle. As discussed earlier ECC decoding can

begin after 1 cycle if the cache line is not compressed and after 4 cycles when compressed (only after all 4 bits of tag are read)[1]. In our performance simulations we have conservatively assumed that every cache line is compressed and thus, would incur a latency overhead of 4 cycles every time a cache line is read from the memory when using CME Scheme 1 as compared to the baseline. From our synthesis results we see that even the largest ECC decoding gets done in one cycle. Thus, for stronger protection there is no additional latency overhead. However, in many of today's memory systems, the payload word gets read from the memory in the first burst even when it is not the first cache line word and the rest of the cache line words get read in the successive bursts. This is called priority word first [203] and is done to improve performance. In our case, we have to wait for the entire cache line to arrive before we can start with the decompression. Thus, priority word first cannot be implemented here. To account for this we have taken an average of 4 cycle overhead. Thus, overall CME Scheme 1 has 13 cycle latency overhead as compared to baseline.

For Scheme 2 the BPC overhead remains the same as Scheme-1. In this scheme, both bits of tag are read in one burst. Therefore, the additional latency overhead for fetching tag is 1 cycle. Here also the priority word first cannot be implemented and thus, additional 4 cycles are taken. Overall accessing a cache line in the memory in Scheme-2 will take 10 more cycles compared to the baseline. The performance results are shown in Figure 7.15. For the system with 8 InO cores, the performance degradation for Scheme-1 is, at most, $\sim 6.9\%$ (avg. 1.8%) and for Scheme-2 is, at most $\sim 3\%$ (avg. 1.05%). This 8 core system had only 2MB shared Last Level Cache (LLC) and had minimal memory access latency hiding techniques like prefetching enabled. As a result, this exaggerates the latency overhead of CME. For the system with a single OoO core and 2MB LLC, the performance degradation when using Scheme-1 is, at most, $\sim 4.5\%$ (avg. 1.6%) and, for Scheme-2 is, at most, $\sim 3.3\%$ (avg. 1.1%). As expected, for OoO core with a larger cache and no competition among cores for cache space and better memory access latency hiding techniques like prefetching, the performance impact when using CME over the baseline is lesser than the previous system.

[1]Since the tag bits are encoded using systematic SECDED code, the original message, i.e., the 4 bits of tag remain unchanged and the remaining 4 bits of redundancy protecting the tag bits are appended at the end.

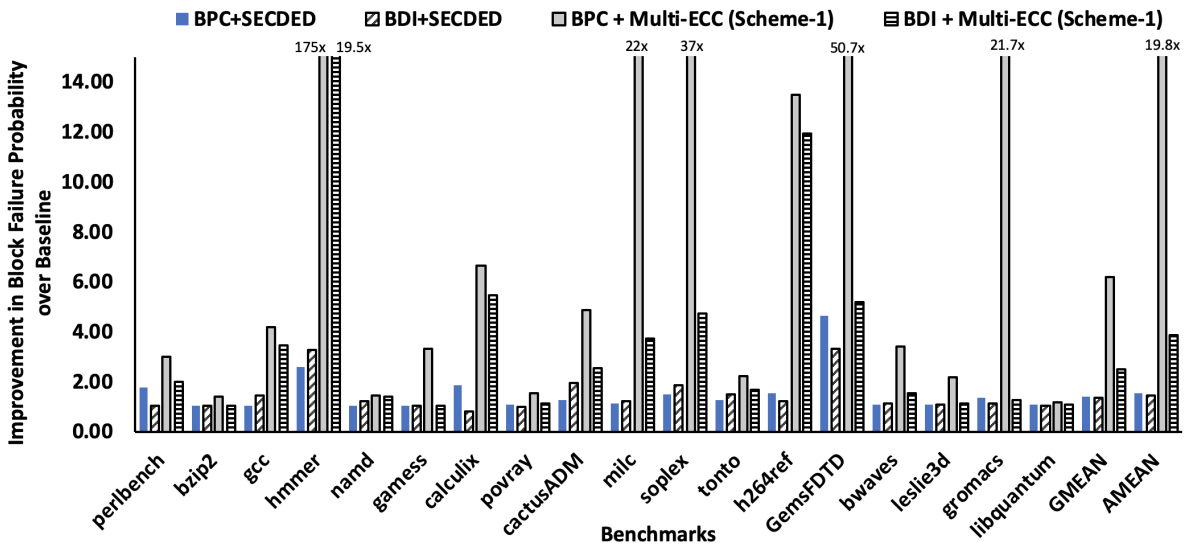


Figure 7.16: Improvement in Block Failure Probability of BΔI and BPC over Baseline [no compression and (72,64)SECDED].

7.6 Discussion

7.6.1 Using an Alternative Compression Scheme

The results presented in Section 7.5.2 were generated using the modified BPC compression scheme discussed in detail in Section 7.3.2. However, we also analyzed another commonly used cache line compression scheme - BΔI. The results are shown in Figure 7.16.

The observation was that BΔI with (72,64)SECDED protection on each block performs better than BPC with (72,64) SECDED for majority of benchmarks. This is because, for most of the benchmarks, cache lines have lower hamming weight when compressed with BΔI as compared to BPC. Since the write and read disturb errors in STT-RAM are asymmetric in nature and all cache lines get the same baseline protection irrespective of how much they are compressed, the reduction in block failure probability comes solely from the lowering of hamming weight. Hence, BΔI performs better than BPC. However, when the extra space is opportunistically used for stronger ECC protection, BPC outperforms BΔI for all 18 benchmarks. With BPC and Multi-ECC (Scheme-1), reduction in block failure probability is upto 175x (geomean of 6.18x). But with BΔI and Multi-ECC (Scheme-1), reduction in block failure probability is upto 19.55X (geomean of 2.49x).

This is because BPC has better compression ratio than BΔI. This allows most cache lines to get stronger ECC protection. Thus, the compression scheme with the highest average compression ratio needs to be chosen for CME even if the scheme results in higher average hamming weight of cache lines.

7.6.2 Variable Scrubbing Interval

Scrubbing is done in today's systems [208] to reduce the probability of multi-bit errors. As mentioned previously, a refresh operation for STT-RAM would need to be accompanied by an ECC check, which resembles scrubbing where a cache line need to be read in to the memory controller which contains the ECC engine and then written back. Therefore, beyond unavailability of the bank/array, a scrub operation would also consume memory bandwidth. Thus, it is also important to minimize the bandwidth consumption overhead of scrubbing. Based on our analysis we see that for most applications, CME allows to relax the scrubbing interval by as much as 50x as compared to the baseline (72,64)SECDED protection scheme. Another observation is that the scrubbing interval required to achieve a target block failure probability varies among applications. It depends on the compressibility of the cache lines. If the compression ratio of the cache lines is high, the scrubbing interval can be relaxed. For CME, the memory controller needs to check the final compressed cache line size to determine the ECC scheme to be used. This information can be used to provide support for variable scrubbing interval. The system initially starts with the lowest scrubbing interval (maximum scrubbing frequency). A counter keeps track of how many times a cache line gets the strongest ECC protection. If it is beyond a certain threshold, the scrubbing interval can is increased. The counter is reset after a certain period of time or every time the scrubbing interval increases.

7.6.3 Using STT-RAM as non-ECC DRAM Alternative - Reliability Point of View

In this work we considered the STT-RAM based memory subsystem as main memory with DDR protocol. As a result we compared the MTTF of STT-RAM devices with baseline protection as well as CME protection against that of non-ECC DRAM. Most of the mobile and low power devices using DRAM do not have ECC protection. STT-RAM, because of its power, density and

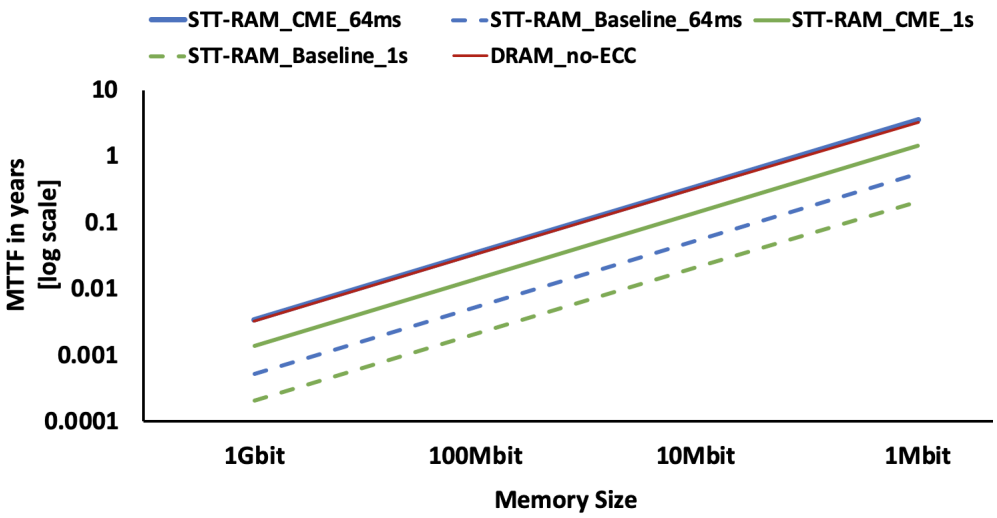


Figure 7.17: MTTF of STT-RAM devices (with different protection schemes and scrubbing intervals) and non-ECC DRAM devices of different sizes. Note that the y-axis is in log scale.

non-volatility benefits over DRAM [178], is considered as a possible DRAM alternative in these devices. However, using STT-RAM memory as a DRAM alternative in such devices would require the STT-RAM device FIT rates to be comparable to what is seen in today's commodity non-ECC DRAM devices. While STT-RAM is not susceptible to radiation induced soft errors [159], the transient read disturb/write/retention error rates can be much worse than the transient bit error rates of DRAM. We use the Samsung design point for our analysis since it has higher bit error rates and used the geometric mean of the block failure probabilities across all benchmarks. We analyze using STT-RAM scrubbing (refresh with ECC) intervals of 64ms (same as the DRAM refresh interval) as well as one second. The DRAM error FIT rate was obtained from [209]. We only considered DRAM transient FIT for our analysis. Note that the FIT rates are for DDR2 DRAM technology. With scaling, the FIT rates for the later DRAM technologies, DDR3/4, are expected to be worse. The results are shown in Figure 7.17.

With baseline protection, the MTTF of STT-RAM device is much lower than a same sized DRAM device, even with 64ms scrubbing interval. In fact, CME with one second scrubbing interval has 2.65x higher MTTF than SECDED based baseline protection with 64ms scrubbing interval. However, it is still not as good as DRAM. With scrubbing interval of 64ms, STT-RAM with CME protection achieves almost similar (*sim*1.07x higher) MTTF as DRAM. Without scrubbing, the

MTTF per Mbit, with CME, drops from a few years to less than an hour, making it almost unusable. Thus, CME protection scheme, with scrubbing, allows STT-RAM to be as reliable an alternative as non-ECC DRAM. The baseline (72,64)SECDED protection scheme requires much lower scrubbing interval (less than 6ms) as compared to 64ms with CME, to achieve similar MTTF and thus, makes it almost infeasible energy and performance-wise.

7.7 Conclusion

In this work, we proposed a new ECC protection scheme for STT-RAM based main memories, compression with multi-ECC (CME). First we try to compress every cache line to reduce its size and then apply hamming weight aware inversion coding to reduce the hamming weight of each block. Based on the amount of compression possible, we use the saved additional bits to increase the protection using stronger ECC codes if possible. Compression with inversion itself reduces the hamming weight of the cache lines, thus reducing the probability of $1 \rightarrow 0$ bit-flips. Opportunistically using stronger ECC codes further helps tolerate multiple bit-flips in a cache line. Our results show that for STT-RAM based main memories, CME can reduce the block failure probability by up to 240x (average 7x) over using a (72,64) SECDED for each cache line word when using two different CME schemes proposed in this chapter. The latency and area overheads of CME is minimal with average performance degradation of less than 1.4%.

CHAPTER 8

PCM-Duplicate: Achieving DRAM-like PCM By Trading Off Capacity For Latency

Phase Change Memory (PCM) is considered one of the most promising scalable non-volatile main memory alternatives to DRAM. It provides ∼4x-5x cost per bit advantage over DRAM, thus enabling a cost-effective dense main memory solution. However, PCM accesses are slower than DRAM, which leads to significantly poorer overall system performance (up to 80% higher execution time for memory-intensive applications based on our analysis). To use PCM as a viable DRAM replacement, the performance gap between the two memory technologies must be bridged, primarily by improving PCM read latency.

In this chapter, we propose an optimized PCM architecture, PCM-Duplicate, that trades off capacity to improve PCM read latency. In PCM-Duplicate, every row in the PCM subarray has a duplicate row. During a memory read, both the rows are activated simultaneously. As a result, the bitline discharges through two PCM cells. This reduces the discharge time significantly, bringing down the overall sensing latency by >3x compared to baseline PCM. While the overall PCM density benefit over DRAM halves, it still provides 2x more capacity than DRAM while having almost comparable read latency. PCM-Duplicate can either be used as a low-cost DRAM main memory alternative or it can be used to replace the DRAM-based last level cache used in today's hybrid main memory systems for the slower PCM memories. Both these system options not only improve main memory capacity but also allow main memory-based persistence by replacing DRAM and making the entire main memory non-volatile.

- Michael Zixing Wang, UCLA
- Prof. Puneet Gupta, UCLA

8.1 Introduction

In today's computing systems, main memories serve a pivotal role, sitting in between the processor cores and the slow storage devices. As a result, there is an ever-increasing demand for main memory capacity to fully extract and exploit the processing power of today's high-performance multicore and manycore systems. Though DRAM is still the main memory workhorse, DRAM scaling is, unfortunately, slowing down. Besides, several application contexts need different properties from the main memory such as higher density, lower cost-per-bit, non-volatility, etc [210]. Hence, it is becoming increasingly important to consider alternative technologies that can potentially avoid the problems faced by DRAM and enable new opportunities.

Several emerging non-volatile memory (NVM) technologies are now being considered as potential replacements for or enhancements to DRAM. Most of these new non-volatile technologies (Phase Change Memory[PCM], STT-RAM, Resistive RAM[ReRAM], etc.) promise better scaling, higher density, and reduced cost-per-bit [30]. 3D-XPoint [211, 212] is one such commercially available PCM-based non-volatile main memory that has gained a lot of attention. Unfortunately, NVMs have their own set of challenges and cannot simply replace DRAM in their current form. Compared to DRAM, NVMs have higher read and write latencies and often consume more energy. Besides, most NVM technologies have limited write endurance and also suffer from high stochastic bit error rates [31, 174, 213]. Hence, most current NVM-based main memory systems are hybrid in nature comprising of both DRAM and NVM [212, 214, 215]. A hybrid memory system helps to increase capacity while reducing the impact on performance.

A hybrid memory system can be configured in two ways [212, 214, 215]. In one configuration (Memory Mode), the smaller but faster DRAM is used as a hardware-managed cache for the denser but slower PCM. However, the DRAM cache is transparent to the operating system (OS) and hence, the total main memory capacity is equal to the total PCM capacity. In the other configuration, the DRAM and PCM are configured as a flat address space, where the OS is aware of both memories for page allocation. Hence, the entire memory capacity can be fully utilized. However, the placement of data between the two memories has to be efficiently managed [215]. The first configuration has the advantage that it can be easily deployed, with DRAM acting as an additional level of caching

between the CPU caches and main memory. However, this mode does not ensure non-volatility as data stored in the DRAM will be lost when power is lost. Besides, the DRAM is transparent to the OS. This has a non-negligible impact on overall memory capacity. DRAM has 4-5x higher cost-per-bit as compared to PCM [216–219] and hence, using DRAM as a transparent cache to mitigate the loss in performance because of the slower PCM increases the overall system cost.

Overall, based on our analysis and past works [33], it seems that improving PCM read latency can provide significant performance gains. If PCM read latency becomes closer to that of DRAM, the use of large DRAM-based caches for performance improvement can be avoided. That would help to increase main memory capacity, reduce memory cost and make the main memory non-volatile. In this chapter, we propose an optimized PCM architecture (PCM-Duplicate) that helps to lower the sensing time in a PCM array by activating two wordlines in a PCM array simultaneously. In this architecture, data is duplicated across two rows in a PCM array. During a read operation, both rows are activated together. This helps to reduce the overall sensing latency since the bitline voltage now discharges through two cells instead of one, thus, increasing the rate of discharge. The overall read latency becomes almost comparable to that of DRAM while the overall PCM capacity becomes half. Thus, there is a capacity-latency tradeoff. We provide two possible ways of using this reduced capacity faster PCM in today's systems. The first option is to use it as a low-cost alternative to today's DRAM-based main memory subsystem. It provides 2x higher capacity/lower cost at $\sim$6% higher average execution time compared to a system using DRAM memory. The other option is to use it in a hybrid main memory setup where the PCM-Duplicate acts as the faster cache (lower cost/higher capacity compared to DRAM cache) for the slower PCM main memory. In this setup, the increased cache capacity provides up to 38% (average 5.7%) higher performance than today's baseline hybrid system with DRAM and PCM. Most importantly, in both system options, the entire main memory system is non-volatile and hence, allows easy main memory-based persistence and checkpointing. This reduces a significant overhead that is incurred in today's systems where the application state is stored in much slower non-volatile storage devices and each checkpoint restoration can take as long as 30 minutes [164].

8.2 Background

This section provides a brief background on two important concepts: details and working of a PCM cell, and the pros and cons of using PCM-based memories as DRAM replacements.

8.2.1 PCM Basics

Phase change memory (PCM) is a type of non-volatile memory. PCM exploits the unique ability of chalcogenide glass of switching between high resistance amorphous and low resistance crystalline states. Figure 8.1 shows the structure of a PCM cell, typically comprising of Germanium-Antimony-Tellurium or GST. The state of the cell can be changed by heating and the two different states represent the stored data; high resistance RESET represents '0' and low resistance SET represents '1'. Switching from one state to the other requires two different heat-time profiles. As shown in Figure 8.1, to program a '0', a high power short pulse quickly raises the temperature of the PCM element above the melting point. The pulse is abruptly terminated, and the small region of melted material rapidly cools through thermal conduction, locking the material in the amorphous state. To program a '1', the amorphous element needs to be converted into a polycrystalline state. To do that, a long electric pulse is used to raise the temperature of the PCM element above the crystallization but below the melting point. The temperature needs to be sustained for a lengthy period so that most of the material crystallizes and the target cell resistance is achieved. Thus, programming a '0' (RESET) requires much lower latency than programming a '1' (SET). To read the data stored in the cell, the cell resistance is sensed without changing the state of the cell.

A PCM memory module is similar to today's DRAM array and is split up into small independent banks. As shown in Figure 8.2, each bank consists of multiple subarrays, where each subarray is made of horizontal wordlines and vertical bitlines. Each bitline is attached to a sense-amplifier circuitry that senses the bitline voltage and outputs a digital value of '0' or '1' based on that. The bitline sense amplifiers drive then drive the global amplifiers shared across all sub-arrays in a bank. The global sense amplifiers boost the voltage and drive the data out of the PCM chips to the processor over a DDRx-based bus.

During a read operation, the read address selects the target bank and activates the corresponding

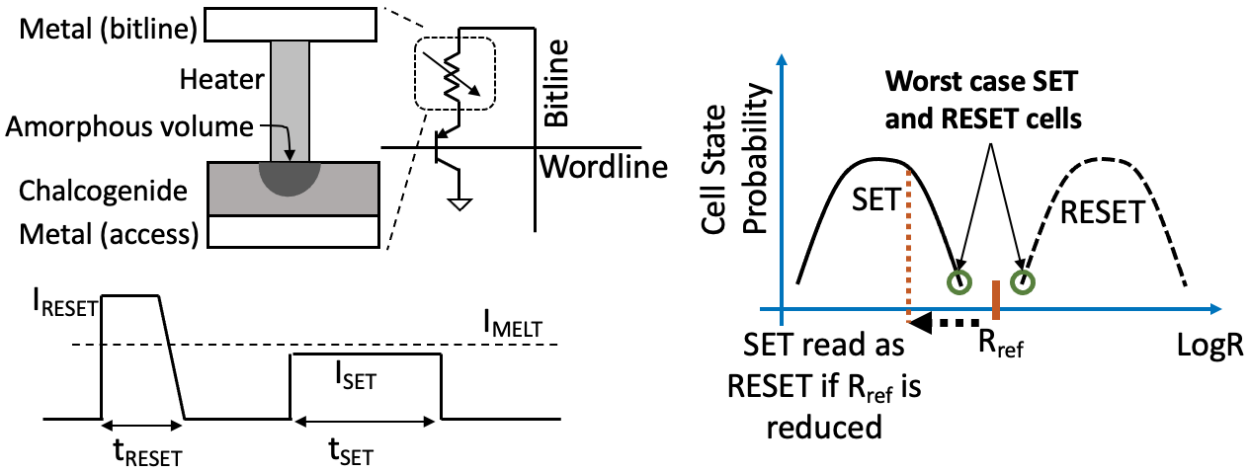


Figure 8.1: Structure of PCM cell, overview of SET and RESET current pulses and variation in cell resistance for SET and RESET states.

wordline. In voltage-based sensing, which has been commonly used in many industry chips [220, 221], the bitline is precharged to V_{rd}. Once the target wordline is turned on, the bitline starts discharging. The rate of discharge depends on the cell resistance. When in SET state, the discharge is faster and takes lesser time for the bitline voltage to drop below the reference voltage (V_{ref}) as compared to that when the cell is in RESET state. After a pre-determined amount of time (T_{sense}), the sense amplifiers compare the bitline voltage with V_{ref}. If it is higher than V_{ref}, then the cell is considered to be in RESET state, if not, then in SET state. The combination of V_{ref} and T_{sense} is decided based on the cell characteristics or the resistance distribution of the cell at each state and the V_{rd} voltage used. The sensing time is determined conservatively to account for device variations and drift. For target SET and RESET resistance values, the wait time before sensing bitline voltage should be such that the voltage separation is at least 300mV between the worst-case cells of the two states (shown in Figure 8.1). The sensing latency (T_{sense}) constitutes the largest fraction of the read time in a PCM array [33]. Hence, in this chapter, we try to reduce the sensing latency using two different optimization schemes.

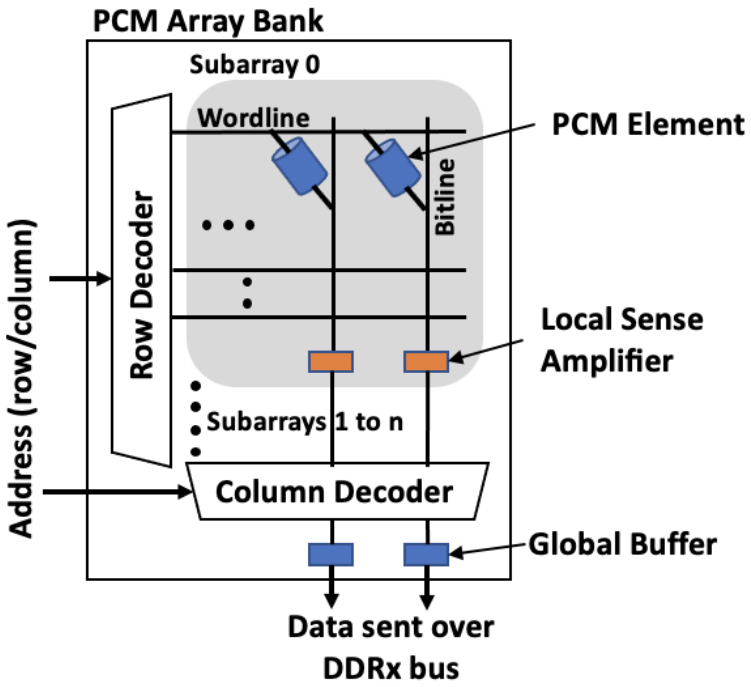


Figure 8.2: Organization of a PCM bank

8.2.2 DRAM vs. PCM

DRAM scaling is, unfortunately, slowing down [222, 223] while the demand for main memory capacity is increasing at a very fast rate. Besides, increasing capacity and aggressive technology scaling in modern DRAM chips is significantly impacting manufacturing yield and reliability [24]. With increasing scaling induced error rates, DRAM manufacturers are resorting to increased row/column sparing and within-dram error-correcting codes to maintain acceptable DRAM yields [24, 26, 29]. These techniques add significant area, latency, and energy overheads. As a result, DRAMs have overall become a large contributor to operational cost. The other major disadvantage with DRAM is that it is volatile in nature. Several application contexts today, such as persistent database management, not only demand higher main memory density, but also the ability to store persistent data in main memories instead of using heavyweight filesystems in today's slower persistent storage devices. Phase Change Memory (PCM) has become one of the most promising scalable byte-addressable main memory alternatives to DRAM. PCM provides a significant 4x-5x [216, 219] cost-per-bit advantage over DRAM while being non-volatile and amenable to technology scaling. As a result, it has been studied extensively and manufactured commercially as DRAM replacement towards

building a low cost, higher capacity main memory system [216].

However, PCM has its own set of disadvantages when compared to DRAM, the most important being higher read and write latencies, write power, and limited endurance [213, 222]. As a result, in today's systems, PCM is typically used in a hybrid main memory setup comprising of both DRAM and PCM memory modules. DRAM modules have limited capacity but lower read and write latencies as compared to the denser but slower PCM. Thus, while PCM helps to achieve higher capacity main memory at a lower cost, DRAM helps to reduce the performance impact by servicing the more frequent/recent accesses at lower latency and energy overheads. This hybrid memory can be used in two modes: (1) Memory mode, and (2) App Direct mode. In the memory mode, DRAM acts as a hardware-managed cache for the slower and denser PCM. This DRAM-based cache, sitting in between the last level cache (LLC) and the PCM main memory, is transparent to the OS [212, 214, 215] and, therefore, the overall main memory capacity is equal to only that of the total PCM. Besides, the main memory is not persistent since data sitting in the volatile DRAM will be lost if the system loses power. In the app direct mode, the DRAM and PCM modules are configured as a flat address space and the OS uses both memories for page allocation. This mode has the advantage of providing higher memory capacity, but faces the challenge of efficient data placement and swapping of data between the two memories. The frequently accessed hot pages would need to be identified in the PCM and swapped with the cold data in the DRAM. Since this data management requires additional hardware/software support, this mode is less frequently used as compared to the memory mode in today's systems.

8.3 Motivation and Past Work

There has been significant research in improving latency and limited endurance of PCM [224–228]. However, achieving read latency similar to that of DRAM has not been talked about much. One of the past works from Nair et. al. [33] focused on improving read latency by either reducing the reference sensing resistance and using ECC to tolerate the uni-directional sensing errors or by increasing the read voltage and using ECC to tolerate the read disturb errors. We tried to combine the two techniques with stronger ECC to achieve the best possible read latency. We describe it in

detail under PCM-ECC Scheme.

8.3.1 PCM-ECC Overview: Combination of Previously Proposed Improvements

In PCM-ECC, we combine the previously proposed techniques [33] to reduce the read sensing time by reducing the reference sensing resistance (R_{sense}) and increasing the read voltage (V_{rd}). PCM cells have a variation in cell resistance in both SET and RESET states as shown in Figure 8.1. The final resistance of the cell depends on the amount of amorphous volume in the cell and how easily the cell crystallizes in either of the two states. The sensing reference resistance is determined by the worst-case SET cell and the read voltage is determined by the worst-case RESET cell. Reducing the sensing resistance leads to lower sensing latency, but also increases sensing circuitry errors. However, this sensing error is unidirectional, where SET gets classified as RESET. On the other hand, increasing the read voltage that the bitlines are pre-charged to before it gets discharged through the cell to the sense amplifier also helps to improve read latency. However, doing so increases error rates due to read disturb. The read current flowing through the cell can accidentally flip the state of the cell. In PCM, read disturb errors are also typically unidirectional and result in RESET switching to SET. Based on prior studies [229], every 30mV increase in sensing voltage increases the read disturb error rate by 3 orders of magnitude.

From past works and PCM device characterization results [230], using a R_{sense} of 10kΩ and V_{rd} of 0.70V results in a sensing time of 69ns and a bit error rate (BER) of 10^{-16}. As proposed in [33], in PCM-ECC, we reduce R_{sense} from 10kΩ to 7kΩ and increase V_{rd} to 0.82V. The combined effect of reduced reference resistance and increased read voltage leads to a significant reduction in sensing time from 69ns to 34ns. But the BER increases exponentially to 10^{-5}, primarily coming from the reduction in reference sensing resistance. This increase in error rate can, however, be mitigated using error correcting codes (ECC). We use a default BER target of 10^{-16}, similar to that in [33]. We show that with a 4-bit rank-level error correcting code (ECC-4) for every 64-bits of data, the probability that a 512-bit memory line would have 5 errors is 3.9×10^{-18}, well below the desired target. The parity storage overhead of ECC-4 is 25-bits per 64-bits of data [231, 232] and incurs a decoding latency of 4 cycles. We assume that the PCM-based memory system will use

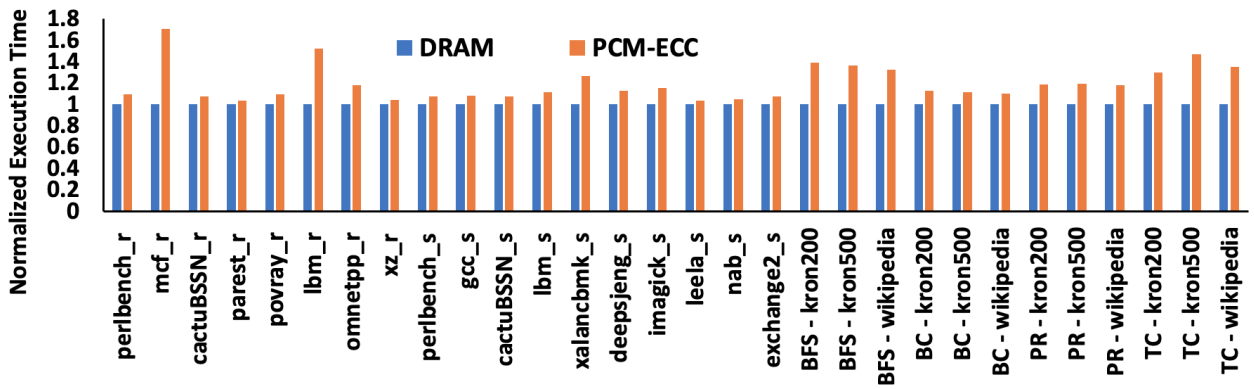


Figure 8.3: Normalized Execution Time of SPEC-2017 and GAP workloads comparing DRAM and PCM-ECC based main memory systems. The execution times are normalized against the system using DRAM.

today's standard DDRx protocol [9]. Thus, the total size of a memory line that is accessed during each memory READ/WRITE operation is 576-bits (512-bits of data and 64-bits of parity bits for rank-level within controller ECC). For ECC-4, with 25-bits of parity per 64-bits of data, the size of the memory line increases to 712-bits. Thus, accessing all the data and parity bits would require two consecutive READ/WRITE operations when using today's DDRx protocol. This is expected to significantly degrade performance. To reduce the impact on performance, we divide the ECC-4 code into two parts: (1) A Single-bit Error Correcting, Multi-bit Error Detecting code (2) A 4-bit Error Correcting Code. The first part of the code requires 8-bits per 64-bits of data and a single cycle of decoding. Thus, to achieve single-error correcting, multi-error detecting property, only one memory access is required. If the decoder detects an uncorrectable error, then the remaining parity bits are fetched using an additional READ and the decoding with error correction takes 4 cycles. For a 512-bit memory line and a BER of 10^{-5}, once every 60,000 reads would require additional access to fetch the extra parity bits.

8.3.2 Performance Analysis: PCM-ECC vs. DRAM

With the modifications in reference sensing resistance and read voltage, the PCM sensing time decreases by half. While this improvement is significant, a DRAM-based memory system still achieves higher performance. To understand the difference in performance we simulated a single-

core system using cycle accurate Gem5 simulator [138]. We ran several workloads from the CPU SPEC 2017 [161] and GAP [163] suites, where we fast-forwarded 1 billion instructions and ran the simulation for a total of 3 billion instructions. We used a 2GHz single-core processor with a private 32KB I-cache, 64KB D-cache, 512KB L2 cache, and 2MB L3 cache. On average we see that the DRAM-based main memory system outperforms the system with the PCM-ECC by an average of 19%. While the sensing latency improvements in the PCM array provide significant benefit (6.7% average improvement in execution time over baseline PCM-based main memory system), it still lags behind DRAM-based main memory system. This is primarily because the PCM-ECC read latency is still more than twice that of DRAM. Thus, PCM-ECC provides $\sim$3x higher capacity/lower cost compared to DRAM, but has up to 60% higher execution time for memory intensive workloads.

8.3.3 Motivation to achieve near-DRAM latency

From the performance results we see that PCM-ECC improves system performance as compared to baseline PCM but it still significantly lags behind DRAM. One way to improve the performance is to have a hybrid main memory system with a DRAM-based cache for slower PCM. But, as mentioned before, the two biggest challenges with that are the limited capacity of the DRAM cache and the lack of non-volatility. Even though the DRAM cache is typically much larger than the LLC, capacity limited applications do not benefit much because of the frequent misses in the DRAM cache. As seen in the evaluations in [33], with a DRAM cache that is 16x larger than the total L3 capacity, the memory-intensive workloads have an average of 20.1 misses per thousand instructions (MPKI). This significantly impacts overall performance. Besides, users cannot utilize the non-volatility advantage of PCM since the DRAM cache is volatile and transparent to the OS. So, even though the OS thinks that it is writing to the non-volatile PCM, the data actually gets written to the DRAM first. If the system loses power before the data in the DRAM cache could be written back to the PCM, the data is lost. Hence, persistence is not guaranteed in such a hybrid memory system. Thus, it is necessary to have a purely PCM-based memory with higher capacity than DRAM while achieving near-DRAM read latency.

8.4 Bridging the Performance Gap Between PCM and DRAM

While PCM-ECC provides larger main memory capacity than DRAM, workloads that do not require the larger capacity and fit within the DRAM suffer because of the larger read/write latencies of the PCM. As a result, we aim to bridge this performance gap by further trading off PCM capacity to achieve near-DRAM read latency. Our proposed reduced capacity PCM-Duplicate architecture has similar read performance as that of DRAM while having $\sim$2x larger capacity at the same cost compared to DRAM.

8.4.1 PCM-Duplicate Overview: PCM with DRAM-like read latency

We propose a PCM-Duplicate scheme that helps to significantly improve PCM read latency by trading off capacity. In this scheme, every row in a PCM subarray will have a duplicate row. The original row and the duplicate row are activated simultaneously when reading from a particular address. As a result, the overall resistance of the PCM cells being sensed is halved as they are connected in parallel. This allows us to reduce the reference resistance (R_{sense}) from 10kΩ to 5kΩ without impacting the bit error rate (BER). We can further reduce R_{sense} and increase read voltage (V_{rd}) to get additional improvements in sensing latency. With $R_{sense} = 4k\Omega$ and $V_{rd} = 0.8V$, we can achieve a sensing latency of 21ns and BER of 10^{-8}. With 2-bit error correction (ECC-2) per 512-bits of data, the probability of having 3 errors is 2.2×10^{-17}, which is well below the desired target of 10^{-16}. The exact ECC protection used is described later in Section 8.4.3. PCM-Duplicate provides an overall sensing latency reduction of more than 3x compared to the baseline PCM memory and brings the overall read latency closer to that of DRAM (1.3x of DRAM). However, it halves the memory capacity. But PCM has more than 4x cost-per-bit benefit over DRAM [216, 218, 219]. Even after halving the capacity, the cost-per-bit benefit of PCM-Duplicate stands at a significant 2x over DRAM. Keeping the trade-offs of PCM-Duplicate in mind, this reduced capacity PCM with near-DRAM latency is also a good fit as the last level cache for slower PCM main memories. Using PCM-Duplicate as the last level cache for the slower PCM main memory instead of DRAM would provide 2x more cache capacity at the same cost, as well as overall main memory persistence since both last-level cache and main memory are PCM-based and hence, non-volatile.

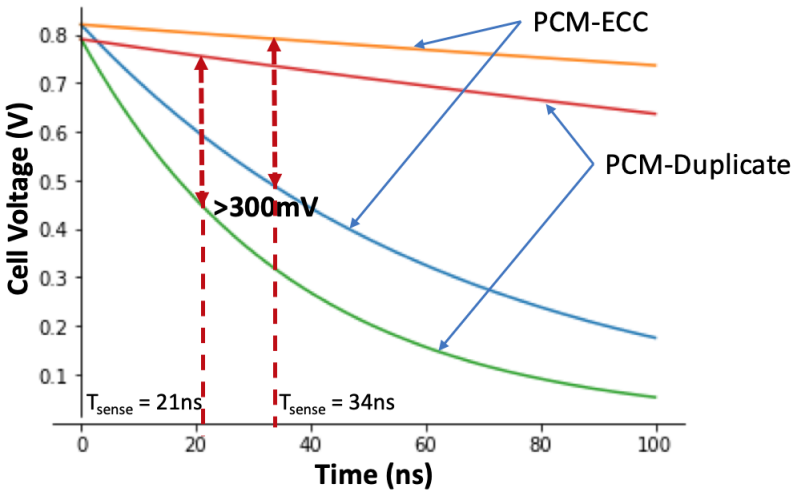


Figure 8.4: Sensing latencies of PCM-ECC vs PCM-Duplicate

8.4.2 PCM-Duplicate Implementation

To activate two rows simultaneously, the row decoder is modified to activate two wordlines using a single address. So, if row address A is sent, rows A and A' are activated. Since two PCM cells in the same state are connected in parallel (as shown in Figure 8.5), the total cell resistance becomes half. As a result, the 300mV separation between the worst-case SET and RESET states during discharge can be achieved much faster than in the baseline PCM cell. The PCM can operate in both normal and duplicate modes. The operation mode can be set by the memory controller during boot time by setting a PCM mode register. In normal mode, the row decoder decodes using all row address bits and activates a single row at a time. In duplicate mode, each PCM subarray is halved in capacity and each row has a corresponding duplicate neighboring row. The row decoder masks the least significant bit of the row address and activates two neighboring rows that have the same A[MSB:LSB-1] bits. The original and duplicate rows must be in the same subarray since each subarray has its own set of local sense amplifiers.

8.4.3 Reducing Write Time and Energy using ECC and Infrequent Refresh

In PCM-Duplicate, the overall write current increases as two PCM cells are being programmed simultaneously. The increase in total write current translates to an overall increase in write energy. One way to reduce the write energy consumption is by reducing the write latency. As provided

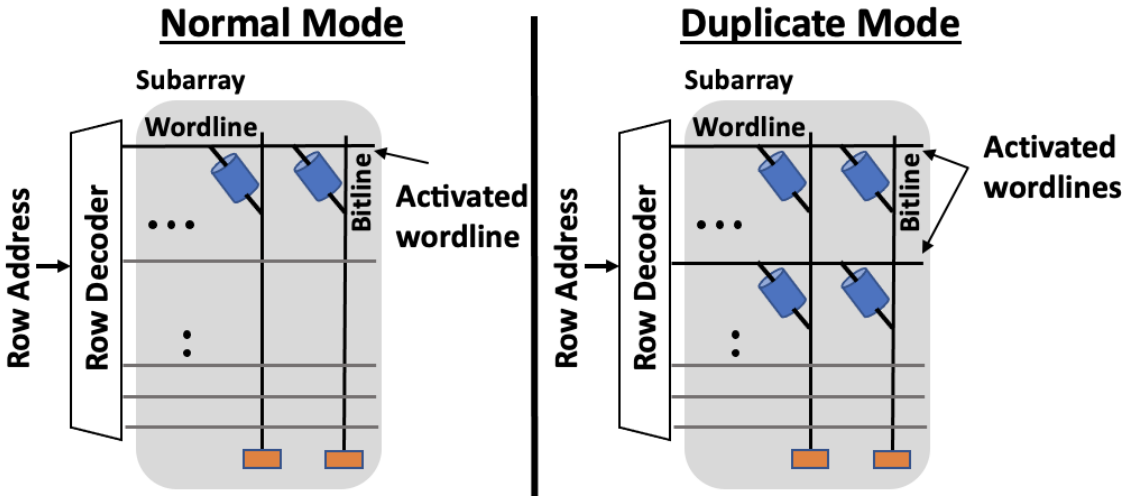


Figure 8.5: The two operation modes in PCM-Duplicate

in [222], the SET latency is ∼4x that of RESET latency. Reducing the SET pulse width results in a smaller volume of crystalline chalcogenide, which increases the resistance of the cell. Besides, the resistance of these partially SET cells increases further over time [233], eventually resulting in retention error when the SET cells are sensed as RESET. Based on the characterization results presented in [228], the resistance drift follows a power-law model [228] and the SET cells begin to lose data after 4 seconds. If the PCM cells are refreshed every 4 seconds, the retention error rate has been estimated to be less than 10^{-8}. The overall BER, considering sensing error, read disturb error, and retention error is $< 2 \times 10^{-8}$. With (144,128) Double Error Correction, Triple Error Detection (DECTED) code per 128-bits of data, the probability of a triple-bit error is 2.7×10^{-18}, which is well below the desired target of 10^{-16}. Hence, to facilitate lower write time for better performance and energy, we use DECTED code per 128-bits of data and refresh memory lines every 4 seconds. This allows us to bring down the SET latency of PCM-Duplicate from $1\mu s$ to 250ns.

8.4.4 Sneak Current in Crossbar Architecture

Typically, in most non-volatile resistive main memory modules, the wordlines and bitlines are organized in a crossbar array to achieve maximum density. In such a setup, each PCM element does not have an individual access transistor. This is because having an access transistor per cell significantly increases the cell size, thereby, negating most of the density benefits of crossbar

architecture. Instead, each PCM element is placed at every wordline and bitline intersection. The bitlines are first precharged to V_{rd}.

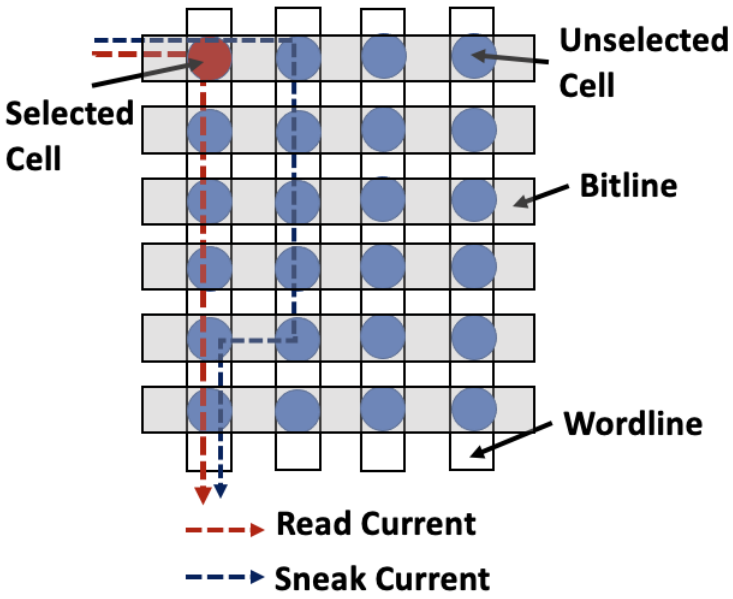


Figure 8.6: Crossbar array structure showing read current and sneak current

In an ideal scenario, based on the row address, during activation, the wordline voltages should be set such that voltage V_{rd} is applied across the target cells and zero voltage is applied across the unselected cells. However, in a crossbar array, while applying the desired voltage across the cells to be read, adjacent cells get partially selected. This creates parallel 'sneak' discharge paths through the adjacent junctions, as shown in Figure 8.6. The parallel paths usually have higher resistance since the sneak currents have to pass through multiple resistive cells in series (three cells in Figure 8.6). But, these parallel circuits can alter the measured output, resulting in a read error. The impact of sneak current depends on the ratio of the current flowing through the target cells to the sneak current. Higher the ratio, the smaller the impact. With two wordlines activated at once in PCM-Duplicate, the number of sneak paths increases. However, the total current flowing through the target cells being read also increase in PCM-Duplicate due to lower total cell resistance (duplicate data cells connected in parallel). Thus, the ratio of actual to sneak current does not increase significantly to worsen the read error. Besides, in crossbar arrays, to tackle the problem of sneak currents, each PCM cell is placed in series with either a diode or a switching mechanism such as Ovonic Threshold Switch (OTS) [234] that is much smaller than an access transistor. The

OTS conducts only when there is enough voltage applied across the cell. The sneak paths consist of multiple PCM/OTS cells in series. Hence, most of the cells in these paths do not see enough voltage across their terminals and hence, remain in the off state.

8.5 Evaluation Methodology

We modified cycle-accurate Gem5 simulator to evaluate the performance of PCM-Duplicate. The details of the simulated systems are provided in Table 8.1. System-1 is a DRAM-based main memory system. System-2 replaces the DRAM main memory with the baseline PCM memory that has 4x higher capacity than DRAM, but has 4.6x larger read time. System-3 uses our proposed PCM-Duplicate memory that provides 2x capacity at the same cost as compared to DRAM while having comparable read latency. We also simulated hybrid memory systems where the faster memory (DRAM/PCM-Duplicate) acts as a hardware-managed cache for the slower PCM main memory. This last level cache is transparent to the OS and, therefore, does not add to the main memory capacity. In System-4, DRAM acts as the last level cache for the baseline PCM main memory. In System-5, we replaced the DRAM cache with our proposed 2x higher capacity PCM-Duplicate cache and the baseline PCM main memory with the PCM-ECC main memory. We simulated several workloads from SPEC CPU 2017 [161], Parsec [162] and GAP [163] benchmark suites. We fast-forwarded 1 billion instructions and ran the simulations for a total of 3 billion instructions. Since most of the larger Parsec and GAP benchmarks either suffer from a timeout or a kernel panic issue when simulating a multi-core system on Gem5 (similar observations reported in [235]), we had to be limited to a single core system and scale down the size of the caches. We used a 2GHz out-of-order single-core processor with a private 32KB I-cache, 64KB D-cache, 256KB L2 cache, and 512KB L3 cache in all six systems. The DRAM cache in System 4 is 8MB and the PCM-Duplicate cache is 16MB. The three graphs used when running the GAP workloads are 200MB and 500MB sized synthetic kron [163] and 600MB sized wikipedia graphs. Using graphs larger than this results in unreasonably long runtimes of more than a day. We also had to scale down the L4 sizes because of the limited-sized SPEC workloads that we used for this analysis. For an L4 size larger than 16MB, the L4 miss rate would be exceptionally low. This means that the number of

Table 8.1: Details of the different memory systems evaluated

	System-1	System-2	System-3	System-4	System-5
Processor					
Details	OoO, single-core, 2GHz				
L1-I/L1-D/L2/L3	32KB/64KB(2-way)/256KB/512KB				
LLC transparent to OS					
Technology	-	-	-	DRAM	PCM-Duplicate
Size	-	-	-	8MB	16MB
Latency (Read/Write)	-	-	-	15ns	21ns/250ns
Main Memory					
Technology	DRAM	Baseline PCM	PCM-Duplicate	Baseline PCM	PCM-ECC
Capacity	8GB	32GB	16GB	32GB	24GB
Read Latency	15ns	69ns	21ns	69ns	34ns
Write Latency	15ns	1us	250ns	1us	1us
Bus per Channel	DDR4				
Ranks per Channel	1				
Number of channels	2				

memory accesses would be too less for gathering any meaningful main memory performance results. Many past works have faced similar issues [33, 226] and hence, could only evaluate a limited set of workloads.

8.6 Results

We evaluate the performance improvements achieved by using our proposed optimized PCM substrates in two different main memory system configurations: (1) Replacing DRAM-based main memory system with PCM-Duplicate instead of baseline PCM, (2) Replacing the DRAM cache in hybrid main memory system with PCM-Duplicate cache for the slower PCM main memory.

8.6.1 Using PCM-Duplicate as Main Memory

PCM-Duplicate has read latency comparable to that of DRAM while providing 2x higher capacity at the same cost. In Figure 8.7, we compared PCM-Duplicate based main memory system (System-3) with DRAM and baseline PCM-based main memory systems (System-1 and System-2, respectively). Overall, for SPEC workloads, we see an average of 12.27% improvement of System-3 with PCM-Duplicate main memory over System-2 using baseline PCM-based main memory. For memory intensive workloads such as mcf and lbm, we see more than 30% speedup with PCM-Duplicate. The Parsec and GAP workloads are more memory intensive and also require larger memory capacity. For these workloads, the PCM-Duplicate provides a significant 23.54% average speedup. The improvement in performance is due to more than 69.5% reduction in read latency. Since the working set sizes of the workloads fit within the main memory, the reduction in main memory capacity in System-3 compared to System-2 did not impact the overall performance of System-3. PCM-Duplicate's read latency is comparable to that of DRAM. When compared against DRAM main memory (System-1), System-3 has, on an average, 4.6% higher execution time for SPEC workloads. This is primarily due to the longer write time of PCM. For larger GAP and Parsec workloads, this increases to 5.9% higher execution time. However, with only $\sim$5% slowdown, on an average, PCM-Duplicate provides 2x more capacity at the same cost or 2x lower cost for the same main memory capacity. Besides, PCM is non-volatile and, therefore, PCM-Duplicate can be easily used for main memory based checkpointing instead of having to use much slower storage.

8.6.2 Using PCM-Duplicate as Last Level Cache instead of DRAM

As mentioned before, today's systems using PCM typically implement a hybrid main memory system where each memory channel consists of both DRAM and PCM DIMMs. The DRAM becomes a hardware managed cache that is transparent to the OS. As a result, the total memory capacity reduces by a non-negligible amount as compared to a full PCM system. Also, the main memory no longer provides the benefit of persistence that comes with NVM-based main memories. In order to deal with both these problems, we propose to using optimized PCM-Duplicate as a smaller but faster cache for the slower PCM main memory. PCM-Duplicate would provide 2x

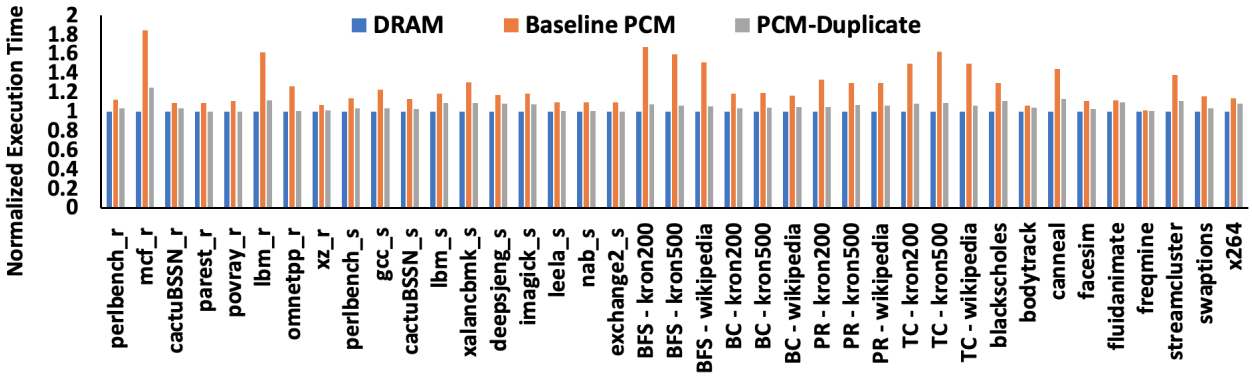


Figure 8.7: Normalized Execution Time of SPEC-2017, GAP and Parsec workloads comparing DRAM (System-1), Baseline-PCM (System-2) and PCM-ECC (System-3) based main memory systems. The execution times are normalized against the System-1.

more cache capacity as compared to DRAM and the entire cache-main memory system would be non-volatile. We evaluated this proposed System-5 and compared its performance with today's standard hybrid main memory system using DRAM based cache and baseline PCM-based main memory (System-4). The results are shown in Figure 8.8.

For less memory intensive SPEC 2017 workloads, we found that our proposed system provides an average of 4.67% speedup (upto 18% speedup for memory intensive workloads like mcf_r) as compared to the baseline hybrid system. For larger, more memory intensive Parsec and GAP workloads, the improvement in performance is upto 38.7% (average 9.43%). The performance improvement is due to the increase in the size of the last level cache. Even though PCM-Duplicate has higher write time than DRAM, the read latency is almost comparable. Since writes do not mostly fall in the critical path, the decrease in misses per thousand instruction (average 15.8%) translates to the overall speedup. For example, canneal has a 14.2% speedup while X264 has a 5.07% speedup. Both applications have similar read to write ratio (2.17:1 vs 2.89:1). But the difference in performance stems from the fact that canneal's misses per thousand instruction reduces by 102% while in x264, the reduction is by 29%. Overall, we see that using PCM-Duplicate as last level cache instead of DRAM helps in improving the overall system performance. This speedup comes from the larger sized last level cache that has similar read performance as that of the DRAM cache.

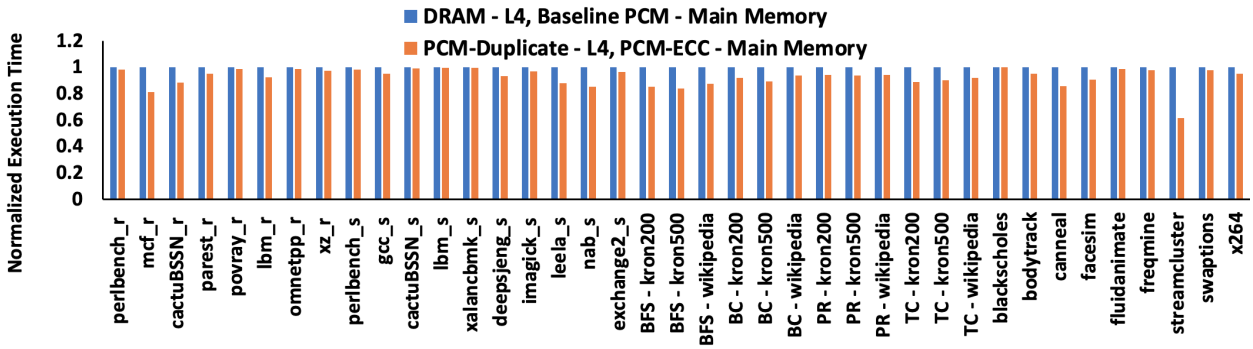


Figure 8.8: Normalized Execution Time of SPEC-2017, Parsec and GAP workloads comparing DRAM and PCM-Duplicate as last level caches (System 4 vs. System 5) for slower PCM main memories. The execution times are normalized against the system using DRAM-based cache (system 4).

8.6.3 Enabling Lightweight Main Memory Based Persistence

In both main memory system configurations using PCM-Duplicate (System-3 and System-5), the main memory is non-volatile. Using DRAM as either the main memory (System-1) or the last level cache for the slower PCM (System-4) in hybrid memory systems makes the overall main memory system volatile. As a result, to ensure data persistence and enable checkpointing, much slower storage devices such as solid state drives(SSDs) or hard disk drives (HDDs) are used to flush and store the application state. During checkpoint recovery, the data is read back from the storage devices into the main memory before the program resumes. This overhead can be upto 30 minutes [164]. Using PCM-Duplicate as main memory or last level cache for slower PCM makes the entire main memory system non-volatile. Thus, expensive slow storage based checkpointing can now be easily replaced by lightweight main memory based checkpointing. Since, the main memory now falls within the persistence domain, the data in the CPU caches and the write queues in the memory controller need to be flushed to the main memory. This dramatically reduces the checkpointing overhead.

8.7 Conclusion

Phase Change Memory (PCM) is considered one of the most promising scalable non-volatile main memory alternatives to DRAM. It provides ∼4x-5x cost per bit advantage over DRAM. However, the PCM has more than4x higher read latency, which leads to significantly poorer overall system performance (up to 80% for memory-intensive applications based on our analysis). To use PCM as a viable DRAM replacement, the performance gap between the two memory technologies has to be bridged, primarily by improving PCM read latency. In this chapter we propose an optimized PCM architecture, PCM-Duplicate, that trades off capacity to improve PCM read latency. In PCM-Duplicate, every row in the PCM subarray has a duplicate row. During a memory read, both the rows are activated simultaneously. As a result, the bitline discharges through two PCM cells. This reduces the discharge time significantly, bringing down the overall sensing latency by >3x compared to baseline PCM. While the overall PCM density benefit over DRAM halves, it still provides 2x more capacity than DRAM while having almost comparable read latency. PCM-Duplicate can either be used as a low-cost DRAM main memory alternative or can be used to replace DRAM-based last level cache for slower PCM main memory. Both these system options allow main memory-based persistence by replacing DRAM and we evaluate both options in this work. The first system using PCM-Duplicate-based main memory has 2x more memory capacity than DRAM, provides non-volatility while having only 6% worse overall system performance (average). The second system provides a fully non-volatile hybrid system with a PCM-Duplicate cache for slower PCM-ECC main memory. This system has 2x more cache capacity and 75% main memory capacity as compared to today's hybrid system with DRAM cache and baseline PCM. Our proposed hybrid system can provide up to 38.7% (average 7.87%) better overall performance than the baseline hybrid system. Thus, PCM-Duplicate-based memory systems provide two significant benefits over DRAM - (1) Higher capacity at lower cost (2) Lightweight main memory-based persistence.

CHAPTER 9

Conclusion

This chapter reviews the key contributions of this dissertation and outlines directions for future work.

9.1 Overview of Contributions

A series of techniques were proposed to cope with hardware variability and errors in memory systems. With increase in memory capacity and decrease in physical dimensions of cells, memory reliability is becoming a growing concern. The challenge with memory resiliency techniques is that the fault tolerance mechanisms need to effective but with minimal overhead.

9.1.1 FaultLink and SAME-Infer

FaultLink and SAME-Infer provided a holistic virtualization-free fault tolerance methodology to deal with hard faults in software managed embedded memories in IoT devices. Hardware design in most of these IoT devices is driven by the need for low cost and low power. One way to reduce power consumption is to lower the supply voltage. But as the VDD is lowered, some of the weak SRAM memory cells begin to fail. Hence, low cost protection against hard faults in memory is required if these devices have to be run at low voltage. FaultLink does exactly that with almost no hardware overhead. In software managed memories, data placement in memory is orchestrated by the software. Thus, application programmers, with the help of tools like compiler and linker, explicitly partition data into physical memory regions that are distinct in the address space. FaultLink utilizes exactly that property of software managed memories and makes loading application in faulty memory plausible. It takes a pre-compiled binary of an application and links

it to the memory in such a way that the application would not access the bad locations. Thus, the application is compiled once but the final linked binary image is unique for every chip. SAME-Infer is an extension of FaultLink for approximation tolerant deep learning inference applications where some of the program sections can tolerate faults. These program sections are placed in partially faulty memory segments based on their fault tolerance capability. Doing so allows us to tolerate higher hard fault rate, which, in turn, allows us to further reduce the operating voltage compared to FaultLink.

9.1.2 SDELC

SDELC helps to recover from unpredictable single bit flips in the memory that occur during runtime. It helps to localize the error to a smaller chunk in a 32/64-bit message and then tries to heuristically recover from it using software defined policies that leverage on the available side information about memory contents to choose the most likely candidate codeword. Overall, SDELC opportunistically copes with memory errors in low-cost IoT devices and helps in correcting majority of the single-bit errors.

9.1.3 Parity++

Parity++, like SDELC, is another lightweight error recovery scheme. But Parity++ tackles the problem of miscorrections that might occur with SDELC during the heuristic recovery. Instead of trying to recover from errors heuristically, Parity++ preferentially provides stronger protection to certain "special messages". While it provides stronger protection than a basic Single Error Detection parity, it has lower overhead than a full single error correcting Hamming code. With just two additional bits of redundancy per message, this code is ideal for last level caches. We also propose a memory speculation scheme that can be used to further hide the decoding latency that comes with using any error correcting code. Parity++ can be extended to embedded scratchpad memories as it has much lower area overhead and can be opportunistically used to reduce the chip area.

9.1.4 COMET

DRAM manufacturers are adopting on-die error correction coding (ECC) schemes, along with within memory controller ECC, to correct SBEs in the memory. Unfortunately, the on-die SEC can miscorrect double-bit errors (which would have been safely detected but uncorrected errors in conventional in-controller ECC) resulting in triple bit errors more than 45% of the time which are then undetectable or miscorrected in the memory controller >55% of the time resulting in silent data corruption. COMET is a collaborative on-die and in-controller error correction scheme that completely eliminates all double-bit error induced silent data corruption and corrects 99.9997% double-bit errors at absolutely no additional storage, latency and area overheads.

9.1.5 Compression with Multi-ECC

Emerging non volatile memories (NVM) have been suggested as potential replacements for DRAM based main memory systems. However, reliability is the current biggest concern facing these NVMs that can potentially eclipse the density and energy benefits these technologies promise. One of the most well researched NVM technologies is magnetic memories, primarily STT-RAMs. Compression with Multi-ECC (CME) aims to provide stronger protection to magnetic memories at the same cost as today's standard ECC solutions. We observed that reading or writing a '1' in these memories is more error-prone that '0'. We use compression to reduce the size of the memory lines and the number of 1's in them. Then, based on the total compression, we opportunistically use the saved space to pack in ECC bits for much stronger protection. Overall, CME reduces memory block failure probability by upto 240x in magnetic main memory-based systems.

9.1.6 PCM-Duplicate

Phase Change Memory (PCM) is another NVM technology that has become one of the most promising scalable byte-addressable main memory alternative to DRAM. PCM provides a significant 4x-5x cost-per-bit advantage over DRAM while being non-volatile and amenable to technology scaling. As a result, it has been studied extensively and manufactured commercially as DRAM replacement towards building a low cost, higher capacity main memory system. However, PCM

also has >4x higher read latency than DRAM, significantly impact overall system performance. Our proposed PCM-Duplicate architecture helps to bring the PCM read latency almost close to that of DRAM by halving the capacity advantage. PCM-Duplicate, if used as DRAM main memory replacement provides 2x capacity/cost benefit while having only 6% (average) poorer overall system performance than DRAM. PCM-Duplicate, if used as replacement for DRAM caches in hybrid main memory system can provide upto 38% (average 7.87%) performance benefit while making the entire main memory system non-volatile. Thus, PCM-Duplicate provides significant cost/density benefit over DRAM with similar or better performance while also allowing lightweight main memory based persistence.

Overall, the dissertation advocates for adopting strongest memory resiliency scheme possible within the overhead limitations. The schemes should be tailored to the type of memory and its position in the memory hierarchy. This dissertation shows how appropriate fault models, software data value behavior, and the memory architecture itself can be intelligently and opportunistically exploited to provide higher reliability while incurring minimal area, power, and performance overheads compared to the conventional reliability schemes used in today's systems.

9.2 Directions for Future Work

While this dissertation investigates error mechanisms, device characteristics, software behavior, and memory architecture to enable lightweight opportunistic memory resilience, there are potential directions and avenues for extending the ideas on memory reliability and improved memory performance. Some of them are summarized below.

9.2.1 Extensions Of Techniques Proposed In This Dissertation

Firstly a FaultLink-compatible remote software update mechanism for IoT devices in the field need to be designed and new failure modes with SDELC need to be supported. Also for FaultLink, the stack and heap in applications were not split. A split stack and heap would lead to smaller program sections, which, in turn, would allow the user to tolerate more faults and thus run at ever lower

supply voltages. Parity++ can be extended to server class systems with large sized last level caches where the chip area savings would be considerable and can be utilized to increase the number of cores or the size of the memory to improve overall system performance. Also, a stronger version of Parity++, which can provide double error correcting to certain "special messages" with just one extra bit as compared to the commonly used SECDED code can be used in off-chip dense main memories with high bit error rate. While we proposed Compression with Multi-ECC (CME) for magnetic memories, the technique adheres to today's DRAM bus protocols. Hence, this technique can be extended and evaluated for DRAM memories, given the increasing rate of scaling induced DRAM errors at advanced technology nodes. PCM-Duplicate can also be further improved by using dynamic sizing of PCM-Duplicate and PCM-ECC during runtime based on memory access patterns. During runtime, based on parameters like cache miss rate, page faults, etc., portions of PCM-Duplicate or PCM-ECC memory can be dynamically converted to PCM-Baseline to get higher main memory capacity instead of better read performance. Dynamic switching between different modes might help in further improving overall system performance.

9.2.2 Asymmetric Error Correction In Non-Volatile Memories

Our analysis as well as past works have shown that the fault models in most non-volatile memories are asymmetric in nature. Programming or modifying when reading the cell content is often tougher in one state than the other. Thus, the probability of an error happening in a cell storing '1' is often much higher or lower than a cell storing '0'. Today's standard error correcting codes usually provide uniform protection from errors. The binary codes think that errors in cells storing 0 and 1 are equally likely, i.e., a $1 \rightarrow 0$ flip is protected the same way as a $0 \rightarrow 1$ flip. But if the probability of a flip happening in one direction is much higher than the other, then providing uniform protection results in unnecessary waste of parity bits. Instead, those parity bits can be efficiently used to provide stronger asymmetric protection to only the cells storing data which has much higher likelihood of errors.

9.2.3 Making Neuromorphic Computing Robust

Alongside that, these emerging non-volatile memory technologies are also being widely used for providing hardware solutions for brain inspired neuromorphic computing. Considering the large number of neurons and synapses required to perform efficient learning and classification, the reliability of these memory technologies is becoming a major concern in the design of hybrid, analog-digital-non-volatile heterogeneous architectures. There is scope in developing novel error detection and correction codes, that, when coupled with the asymmetric error pattern seen in most NVMs, will enable robust neuromorphic computing at low overheads. One such example is arithmetic codes (AN codes) [236] that were previously used in computer processors to ensure the accuracy of its arithmetic operations. Unlike Hamming codes that use hamming weight and hamming distance, AN codes use arithmetic weight to maximize the arithmetic distance between codewords.

9.2.4 Improving Reliability and Endurance in Hybrid Main Memory Systems

With rising demand for using hybrid persistent main memory, non-volatile memory (NVM) specific reliability techniques and opportunistic NVM systems that exploit and cope with variability are becoming increasingly critical. However, the two major problems in these hybrid persistent main memory systems are: (1) difference in error rates between DRAM and the NVM, and (2) low endurance of the non-volatile memory technology. One can look into techniques for coming up with a single solution that would provide stronger protection for the NVM while providing enough information to enable intelligent swapping between the DRAM and the NVM that would reduce the number of NVM program/erase cycles.

When swapping pages between DRAM and NVM, the dirty bit of the top eight candidates that can be swapped out of DRAM are first read. If any of them is clean it is chosen for the swap since that would not require any write on the NVM side. If none of them are clean, the ECC bits of those dirty candidate pages are read. These dirty ECC bits are compared against the old ECC bits of the corresponding pages in the NVM. The ECC would be designed in such a way that this comparison would give a fair idea of the number of bit flips between the updated dirty copy sitting in the DRAM

and the old copy sitting in the NVM. While this can be achieved using a hash, the goal here is to use a special type of error correcting code so that this code can serve two purposes: (1) provide stronger protection to the messages stored in the NVM, and (2) help in figuring out the number of bit flips between the updated and the old copies. The page that would result in the least number of bit flips would be chosen for eviction from DRAM since that would eventually lead to the smallest number of bits that would need to be re-programmed in the NVM.

9.2.5 Enabling Shared-Bus Read/Write in Memories for Performance and Energy Efficiency

As we have seen in this dissertation, workloads from several prevailing and emerging application domains often exhibit significant value locality where same or almost same values get written to or read from different memory locations within a short interval of time. Thus, consecutive read or write operations to different memory locations, often load or store very similar values. One such example can be today's deep learning workloads where nearby inputs or activations tend to be of similar values and are accessed in consecutive cycles. Most of the past research has only leveraged the value locality within the same memory line and did not exploit the value locality across different memory operations and locations. The concept of shared bus memory operations aims to exploit this value locality to significantly reduce redundant load/store operations. If two successive load/store operations are reading/writing almost same values from/to two different locations, the operations can be done simultaneously, providing dramatic performance and energy gains.

Some early analysis based on workloads from SPEC benchmark suite shows that more than 40% memory operations can be shared with 85-90% of the data bits (avg.) being identical. However, efficiently leveraging value locality for shared read/write is not possible in most of today's memory technologies. This is because, if the two values are not exactly the same, the bits that are different would have to be accessed separately using partial read/write. Either today's memory technologies or peripheral circuitries do not allow partial accesses, or these partial accesses take up additional operations (over multiple extra cycles), thus providing no performance or energy benefits. One can aim to design appropriate structures such as modified sense amplifiers that will sense more than two values. For example, if two wordlines are read together, then the sense amplifier will have to

differentiate ‘00’, ‘11’, ‘01’, and ‘10’. This will enable combination of multiple distinct read and write operations in order to tune performance and/or energy consumption.

9.2.6 Combining Memory Reliability with Security

Memory system security and reliability are both concerned with the confidentiality, integrity, and availability of systems. Techniques to achieve either of the two typically involve adding extra information (parity bits or cryptography keys) to the original data that is being protected. However, security and reliability research have mostly been pursued independently. A reliability problem can often lead to a security issue and techniques to enhance memory system reliable can also help in making it secure. For instance, row hammer attacks in DRAMs use the fact that activating one row multiple times in a bank often lead to bit flips in its neighboring victim rows. The modified data in the victim rows is then exploited by the attacker to access privileged data. If on-chip or rank-level error correcting codes are present, these bit flips induced by the attacker will get corrected before the data is sent out of the memory controller. Thus, simple reliability enhancing techniques like ECC can prevent security concerning row hammer attacks. One possible scope of extending Lightweight Opportunistic Memory Resilience framework is by combining memory security and reliability. One can develop a framework that would take into account attributes used in both reliability and security research communities and propose solutions that take care of both threats in today’s memory systems.

REFERENCES

[1] M. Gottscho, I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Low-cost memory fault tolerance for iot devices," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 128:1–128:25, Sept. 2017.

[2] L. Wei, J. G. Alzate, U. Arslan, J. Brockman, N. Das, K. Fischer, T. Ghani, O. Golonzka, P. Hentges, R. Jahan, P. Jain, B. Lin, M. Meterelliyoz, J. O'Donnell, C. Puls, P. Quintero, T. Sahu, M. Sekhar, A. Vangapaty, C. Wiegand, and F. Hamzaoglu, "13.3 a 7mb stt-mram in 22ffl finfet technology with 4ns read sensing time at 0.9v using write-verify-write scheme and offset-cancellation sensing technique," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 214–216, Feb 2019.

[3] J. H. Kim, W. C. Lim, U. H. Pi, J. M. Lee, W. K. Kim, J. H. Kim, K. W. Kim, Y. S. Park, S. H. Park, M. A. Kang, Y. H. Kim, W. J. Kim, S. Y. Kim, J. H. Park, S. C. Lee, Y. J. Lee, J. M. Yoon, S. C. Oh, S. O. Park, S. Jeong, S. W. Nam, H. K. Kang, and E. S. Jung, "Verification on the extreme scalability of stt-mram without loss of thermal stability below 15 nm mtj cell," in *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pp. 1–2, June 2014.

[4] I. Alam, L. Dolecek, and P. Gupta, "Lightweight software-defined error correction for memories," in *Dependable Embedded Systems*, pp. 207–232, Springer, Cham, 2021.

[5] I. Alam, *Lightweight Fault Tolerance in SRAM Based On-Chip Memories*. University of California, Los Angeles, 2018.

[6] I. Alam and P. Gupta, "Same-infer: Software assisted memory resilience for efficient inference at the edge," in *The International Symposium on Memory Systems*, MEMSYS 2020, (New York, NY, USA), p. 10–22, Association for Computing Machinery, 2020.

[7] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-aware resiliency: Unequal message protection for random-access memories," in *2017 IEEE Information Theory Workshop (ITW)*, pp. 166–170, Nov 2017.

[8] I. Alam, C. Schoeny, L. Dolecek, and P. Gupta, "Parity++: Lightweight error correction for last level caches," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 114–120, 2018.

[9] I. Alam, S. Pal, and P. Gupta, "Compression with multi-ecc: Enhanced error resiliency for magnetic memories," in *Proceedings of the International Symposium on Memory Systems*, pp. 85–100, 2019.

[10] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," *Communications of the ACM*, vol. 54, pp. 100–107, Feb. 2011.

[11] S. Mittal, "A Survey of Architectural Techniques for Managing Process Variation," *ACM Computing Surveys*, vol. 48, no. 4, 2016.

[12] A. Rahimi, L. Benini, and R. K. Gupta, "Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques From Circuits to Software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410–1448, 2016.

[13] B. Schroeder and G. A. Gibson, "Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You?," in *Proc. 5th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2007.

[14] 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 *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015.

[15] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014.

[16] V. Narayanan and Y. Xie, "Reliability Concerns in Embedded System Designs," *Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[17] NSF and SRC, "Report to the National Science Foundation on the Workshop for Energy Efficient Computing," tech. rep., 2015.

[18] J. Wang and B. H. Calhoun, "Minimum Supply Voltage and Yield Estimation for Large SRAMs Under Parametric Variations," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 19, no. 11, pp. 2120–2125, 2011.

[19] S. E. Schuster, "Multiple Word/Bit Line Redundancy for Semiconductor Memories," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 13, no. 5, pp. 698–703, 1978.

[20] P. Gupta, Y. Agarwal, L. Dolecek, N. Dutt, R. K. Gupta, R. Kumar, S. Mitra, A. Nicolau, T. S. Rosing, M. B. Srivastava, S. Swanson, and D. Sylvester, "Underdesigned and Opportunistic Computing in Presence of Hardware Variability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 32, no. 1, pp. 8–23, 2013.

[21] M. Jung, C. Weis, N. Wehn, M. Sadri, and L. Benini, "Optimized active and power-down mode refresh control in 3d-drams," in *2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–6, 2014.

[22] Sanghyuk Kwon, Young Hoon Son, and Jung Ho Ahn, "Understanding ddr4 in pursuit of in-dram ecc," in *2014 International SoC Design Conference (ISOCC)*, pp. 276–277, 2014.

[23] 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 *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), p. 60–71, Association for Computing Machinery, 2013.

[24] "ECC Brings Reliability and Power Efficiency to Mobile Devices," tech. rep., Micron technology, Inc., 2017.

[25] P. J. Nair, V. Sridharan, and M. K. Qureshi, "Xed: Exposing on-die error detection information for strong memory reliability," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 341–353, 2016.

[26] M. Patel, J. S. Kim, T. Shahroodi, H. Hassan, and O. Mutlu, "Bit-exact ecc recovery (beer): Determining dram on-die ecc functions by exploiting dram data retention characteristics," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 282–297, 2020.

[27] S.-L. Gong, J. Kim, and M. Erez, "Dram scaling error evaluation model using various retention time," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 177–183, 2017.

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

[29] M. Horiguchi and K. Itoh, *Nanoscale Memory Repair*. New York: Springer SBM, 2011.

[30] S. Mittal and J. S. Vetter, "A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 1537–1550, May 2016.

[31] S. Sills, S. Yasuda, A. Calderoni, C. Cardon, J. Strand, K. Aratani, and N. Ramaswamy, "Challenges for High-Density 16Gb ReRAM with 27nm Technology," in *Symposium on VLSI Circuits (VLSI Circuits)*, pp. T106–T107, June 2015.

[32] Y. Emre, C. Yang, K. Sutaria, Y. Cao, and C. Chakrabarti, "Enhancing the Reliability of STT-RAM through Circuit and System Level Techniques," in *2012 IEEE Workshop on Signal Processing Systems*, pp. 125–130, Oct 2012.

[33] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi, "Reducing read latency of phase change memory via early read and turbo read," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 309–319, 2015.

[34] S. Hamdioui, G. Gaydadjiev, and A. J. van de Goor, "The State-of-art and Future Trends in Testing Embedded Memories," in *International Workshop on Memory Technology, Design and Testing (MTDT)*, 2004.

[35] S.-L. Lu, Q. Cai, and P. Stolt, "Memory Resiliency," *Intel Technology Journal*, vol. 17, no. 1, 2013.

[36] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. 1999.

[37] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: A Design Alternative for Cache On-Chip Memory in Embedded Systems," in *Proceedings of the ACM/IEEE International Symposium on Hardware/Software Codesign (CODES)*, 2002.

[38] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge, "Circuit and Microarchitectural Techniques for Reducing Cache Leakage Power," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 12, no. 2, pp. 167–184, 2004.

[39] L. Wanner, C. Apte, R. Balani, P. Gupta, and M. Srivastava, "Hardware Variability-Aware Duty Cycling for Embedded Sensors," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 21, no. 6, pp. 1000–1012, 2013.

[40] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification (Version 1.2)," 1995.

[41] N. Dutt, P. Gupta, A. Nicolau, A. BanaiyanMofrad, M. Gottscho, and M. Shoushtari, "Multi-Layer Memory Resiliency," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2014.

[42] R. C. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, 2005.

[43] L. Wanner, L. Lai, A. Rahimi, M. Gottscho, P. Mercati, C.-H. Huang, F. Sala, Y. Agarwal, L. Dolecek, N. Dutt, P. Gupta, R. Gupta, R. Jhala, R. Kumar, S. Lerner, S. Mitra, A. Nicolau, T. Simunic Rosing, M. B. Srivastava, S. Swanson, D. Sylvester, and Y. Zhou, "NSF Expedition on Variability-Aware Software: Recent Results and Contributions," *De Gruyter Information Technology (it)*, vol. 57, no. 3, 2015.

[44] L. Lai, *Cross-Layer Approaches for Monitoring, Margining and Mitigation of Circuit Variability*. Ph.d. dissertation, University of California, Los Angeles (UCLA), 2015.

[45] Y. Agarwal, A. Bishop, T.-B. Chan, M. Fotjik, P. Gupta, A. B. Kahng, L. Lai, P. Martin, M. Srivastava, D. Sylvester, L. Wanner, and B. Zhang, "RedCooper: Hardware Sensor Enabled Variability Software Testbed for Lifetime Energy Constrained Application," tech. rep., University of California, Los Angeles (UCLA), 2014.

[46] S. Lamikhov-Center, "ELFIO: C++ Library for Reading and Generating ELF Files." = http://elfio.sourceforge.net/, 2016.

[47] "CPLEX Optimizer." `https://www.ibm.com/analytics/cplex-optimizer`.

[48] M. F. Schilling, "The Surprising Predictability of Long Runs," *Mathematics Magazine*, vol. 85, no. 2, pp. 141–149, 2012.

[49] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the IEEE International Workshop on Workload Characterization (IWWC)*, 2001.

[50] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March SS: A Test for All Static Simple RAM Faults," in *International Workshop on Memory Technology, Design, and Testing (MTDT)*, 2002.

[51] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design and Test*, vol. 34, no. 2, pp. 60–68, 2017.

[52] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.0." `https://riscv.org`, 2014.

[53] P. P. Shirvani and E. J. McCluskey, "PADded Cache: A New Fault-Tolerance Technique for Cache Memories," in *Proceedings of the VLSI Test Symposium*, 1999.

[54] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories," in *Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.

[55] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 27–38, 2005.

[56] M. Mutyam and V. Narayanan, "Working with Process Variation Aware Caches," in *Design, Automation, and Test in Europe (DATE)*, 2007.

[57] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2008.

[58] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "ZerehCache: Armoring Cache Architectures in High Defect Density Technologies," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2009.

[59] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Archipelago: A Polymorphic Cache Design for Enabling Robust Near-Threshold Operation," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[60] A. BanaiyanMofrad, H. Homayoun, and N. Dutt, "FFT-Cache: A Flexible Fault-Tolerant Cache Architecture for Ultra Low Voltage Operation," in *Proceedings of the ACM/IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011.

[61] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.

[62] M. Manoochehri, M. Annavaram, and M. Dubois, "CPPC: Correctable Parity Protected Cache," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.

[63] M. K. Qureshi and Z. Chishti, "Operating SECDED-Based Caches at Ultra-Low Voltage with FLAIR," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[64] T. Mahmood, S. Hong, and S. Kim, "Ensuring Cache Reliability and Energy Scaling at Near-Threshold Voltage with Macho," *IEEE Transactions on Computers (TC)*, vol. 64, no. 6, pp. 1694–1706, 2015.

[65] M. Gottscho, A. BanaiyanMofrad, N. Dutt, A. Nicolau, and P. Gupta, "DPCS: Dynamic Power/Capacity Scaling for SRAM Caches in the Nanoscale Era," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, p. 26, 2015.

[66] M. Mavropoulos, G. Keramidas, and D. Nikolos, "A Defect-Aware Reconfigurable Cache Architecture for Low-Vccmin DVFS-Enabled Systems," in *Design, Automation, and Test in Europe (DATE)*, 2015.

[67] S. Mittal, "A Survey of Architectural Techniques for Improving Cache Power Efficiency," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 1, pp. 33–43, 2014.

[68] F. J. Aichelmann, "Fault-Tolerant Design Techniques for Semiconductor Memory Applications," *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 177–183, 1984.

[69] L. A. D. Bathen and N. D. Dutt, "E-RoC: Embedded RAIDs-on-Chip for Low Power Distributed Dynamically Managed Reliable Memories," in *Design, Automation, and Test in Europe (DATE)*, 2011.

[70] R. van Rein, "BadRAM: Linux Kernel Support for Broken RAM Modules," 2016.

[71] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM Refresh-Power Through Critical Data Partitioning," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[72] L. A. D. Bathen, N. D. Dutt, A. Nicolau, and P. Gupta, "VaMV: Variability-Aware Memory Virtualization," in *Design, Automation, and Test in Europe (DATE)*, 2012.

[73] M. Gottscho, L. A. D. Bathen, N. Dutt, A. Nicolau, and P. Gupta, "ViPZonE: Hardware Power Variability-Aware Memory Management for Energy Savings," *IEEE Transactions on Computers (TC)*, vol. 64, no. 5, pp. 1483–1496, 2015.

[74] M. M. Sabry, D. Atienza, and F. Catthoor, "OCEAN: An Optimized HW/SW Reliability Mitigation Approach for Scratchpad Memories in Real-Time SoCs," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, 2014.

[75] H. Sayadi, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, "A Data Recomputation Approach for Reliability Improvement of Scratchpad Memory in Embedded Systems," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2014.

[76] A. M. H. Monazzah, H. Farbeh, S. G. Miremadi, M. Fazeli, and H. Asadi, "FTSPM: A Fault-Tolerant ScratchPad Memory," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[77] F. Li, G. Chen, M. Kandemir, and I. Kolcu, "Improving Scratch-Pad Memory Reliability Through Compiler-Guided Data Block Duplication," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2005.

[78] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate Storage in Solid-State Memories," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.

[79] 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 *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[80] M. Shoushtari, A. BanaiyanMofrad, and N. Dutt, "Exploiting Partially-Forgetful Memories for Approximate Computing," *IEEE Embedded Systems Letters (ESL)*, vol. 7, no. 1, pp. 19–22, 2015.

[81] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate Storage for Energy Efficient Spintronic Memories," in *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, 2015.

[82] C. Yan and R. Joseph, "Enabling Deep Voltage Scaling in Delay Sensitive L1 Caches," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[83] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture Support for Defending Against Buffer Overflow Attacks," in *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[84] M. Ahmad, "Reliability models for the internet of things: A paradigm shift," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pp. 52–59, Nov 2014.

[85] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, (New York, NY, USA), pp. 8:1–8:12, ACM, 2017.

[86] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[87] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[88] S. Koppula, L. Orosa, A. G. Yağlçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, "Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 166–181, ACM, 2019.

[89] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *CoRR*, vol. abs/1506.02626, 2015.

[90] C. Sakr and N. Shanbhag, "An analytical method to determine minimum per-layer precision of deep neural networks," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1090–1094, April 2018.

[91] "Multiple Knapsack Problem." `http://www.or.deis.unibo.it/kp/Chapter6.pdf`.

[92] "THE CIFAR-10 DATASET." `https://www.cs.toronto.edu/~kriz/cifar.html`.

[93] "THE MNIST DATABASE." `http://yann.lecun.com/exdb/mnist/`.

[94] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *CoRR*, vol. abs/1804.03209, 2018.

[95] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: efficient neural network kernels for arm cortex-m cpus," *CoRR*, vol. abs/1801.06601, 2018.

[96] T. D. Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, 2016.

[97] S. Kim, P. Howe, T. Moreau, A. Alaghi, L. Ceze, and V. Sathe, "Matic: Learning around errors for efficient low-voltage neural network accelerators," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, 2018.

[98] C. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 331–344, 2019.

[99] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, E. Elsen, J. H. Engel, L. Fan, C. Fougner, T. Han, A. Y. Hannun, B. Jun, P. LeGresley, L. Lin, S. Narang, A. Y. Ng, S. Ozair, R. Prenger, J. Raiman, S. Satheesh, D. Seetapun, S. Sengupta, Y. Wang, Z. Wang, C. Wang, B. Xiao, D. Yogatama, J. Zhan, and Z. Zhu, "Deep speech 2: End-to-end speech recognition in english and mandarin," *CoRR*, vol. abs/1512.02595, 2015.

[100] N. Seifert, P. Slankard, M. Kirsch, B. Narasimham, V. Zia, C. Brookreson, A. Vo, S. Mitra, B. Gill, and J. Maiz, "Radiation-induced soft error rates of advanced cmos bulk devices," in *2006 IEEE International Reliability Physics Symposium Proceedings*, pp. 217–225, 2006.

[101] V. Degalahal, Lin Li, V. Narayanan, M. Kandemir, and M. J. Irwin, "Soft errors issues in low-power caches," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 10, pp. 1157–1166, 2005.

[102] "Stringent Tests from ICs to Modules Ensure DRAM Reliability." `https://www.atpinc.com/blog/dram-testing-module-chips-ic-burn-in-quality-characteristics`.

[103] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–14, 2016.

[104] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," tech. rep., IBM Microelectronics Division, 1997.

[105] J. K. Wolf and B. Elspas, "Error-Locating Codes – A New Concept in Error Control," *IEEE Transactions on Information Theory*, vol. 9, no. 2, pp. 113–117, 1963.

[106] J. K. Wolf, "On an Extended Class of Error-Locating Codes," *Information and Control*, vol. 8, no. 2, pp. 163–169, 1965.

[107] E. Fujiwara and M. Kitakami, "A class of Error Locating Codes for Byte-Organized Memory Systems," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1993.

[108] J. Song, G. Bloom, and G. Palmer, "SuperGlue: IDL-Based, System-Level Fault Tolerance for Embedded Systems," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[109] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[110] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pp. 258–265, 2000.

[111] A. Alameldeen and D. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," tech. rep., University of Wisconsin, Madison, 2004.

[112] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[113] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[114] A. Waterman and Y. Lee, "Spike, a RISC-V ISA Simulator – git commit 3bfc00e." = https://github.com/riscv/riscv-isa-sim.

[115] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[116] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt, "Cache scrubbing in microprocessors: myth or necessity?," in *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings.*, pp. 37–42, March 2004.

[117] J. Yan and W. Zhang, "Evaluating instruction cache vulnerability to transient errors," in *Proceedings of the 2006 Workshop on MEmory Performance: DEaling with Applications, Systems and Architectures*, MEDEA '06, (New York, NY, USA), pp. 21–28, ACM, 2006.

[118] H. Duwe, X. Jian, and R. Kumar, "Correction prediction: Reducing error correction latency for on-chip memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 463–475, Feb 2015.

[119] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. Elsevier, 1977.

[120] S. Lin and D. J. Costello, *Error control coding*, vol. 2. Prentice Hall Englewood Cliffs, 2004.

[121] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced srams," in *IEEE International Electron Devices Meeting 2003*, pp. 21.4.1–21.4.4, Dec 2003.

[122] C. W. Slayman, "Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations," *IEEE Transactions on Device and Materials Reliability*, vol. 5, pp. 397–404, Sept 2005.

[123] "Qualcomm Centriq 2400 Processor." `https://www.qualcomm.com/products/qualcomm-centriq-2400-processor`.

[124] J. Huynh, "White Paper: The AMD Athlon MP Processor with 512KB L2 Cache," tech. rep., May 2003.

[125] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, pp. 66–76, March 2003.

[126] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, pp. 5–25, Jan 2002.

[127] M. Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, pp. 395–401, July 1970.

[128] D. H. Yoon and M. Erez, "Memory mapped ecc: Low-cost error protection for last level caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 116–127, ACM, 2009.

[129] S. Kim and A. K. Somani, "Area efficient architectures for information integrity in cache memories," *SIGARCH Comput. Archit. News*, vol. 27, pp. 246–255, May 1999.

[130] N. N. Sadler and D. J. Sorin, "Choosing an error protection scheme for a microprocessor's l1 data cache," in *2006 International Conference on Computer Design*, pp. 499–505, Oct 2006.

[131] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined ECC: Heuristic recovery from uncorrectable memory errors," tech. rep., University of California, Los Angeles, Oct. 2017.

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

[133] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," 2004.

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

[135] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (Piscataway, NJ, USA), pp. 329–340, IEEE Press, 2016.

[136] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-aware resiliency: Unequal message protection for random-access memories," in *Proc. IEEE Information Theory Workshop*, (Kaohsiung, Taiwan), pp. 166–170, Nov. 2017.

[137] R. W. Hamming, "Error detecting and error correcting codes," *Bell Labs Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[138] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[139] Q. Nguyen, "RISC-V Tools (GNU Toolchain, ISA Simulator, Tests) – git commit 816a252." `https://github.com/riscv/riscv-tools`.

[140] A. Waterman, "RISC-V Proxy Kernel – git commit 85ae17a." `https://github.com/riscv/riscv-pk/commit/85ae17a`.

[141] 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 *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 13–25, 2019.

[142] S.-I. Pae, V. Kozhikkottu, D. Somasekar, W. Wu, S. G. Ramasubramanian, M. Dadual, H. Cho, and K.-W. Kwon, "Minimal aliasing single-error-correction codes for dram reliability improvement," *IEEE Access*, vol. 9, pp. 29862–29869, 2021.

[143] T.-Y. Oh, H. Chung, Y.-C. Cho, J.-W. Ryu, K. Lee, C. Lee, J.-I. Lee, H.-J. Kim, M. S. Jang, G.-H. Han, K. Kim, D. Moon, S. Bae, J.-Y. Park, K.-S. Ha, J. Lee, S.-Y. Doo, J.-B. Shin, C.-H. Shin, K. Oh, D. Hwang, T. Jang, C. Park, K. Park, J.-B. Lee, and J. S. Choi, "25.1 a 3.2gb/s/pin 8gb 1.0v lpddr4 sdram with integrated ecc engine for sub-1v dram core operation," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 430–431, 2014.

[144] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[145] S. Cha, O. Seongil, 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 *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 61–72, 2017.

[146] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[147] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting dram scaling by tolerating high error rates," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), p. 72–83, Association for Computing Machinery, 2013.

[148] K. K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, (New York, NY, USA), p. 323–336, Association for Computing Machinery, 2016.

[149] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu, "Improving dram performance by parallelizing refreshes with accesses," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 356–367, 2014.

[150] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking dram design and organization for energy-constrained multi-cores," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), p. 175–186, Association for Computing Machinery, 2010.

[151] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, "A case for exploiting subarray-level parallelism (salp) in dram," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 368–379, 2012.

[152] M. Y. Hsiao, "A Class of Optimal Minimum Odd-Weight-Column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[153] J. Kim, M. Sullivan, and M. Erez, "Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 101–112, 2015.

[154] JEDEC, "DDR4 SDRAM Specication," 2012.

[155] I. Micron Technology, "8Gb: x4, x8, x16 DDR4 SDRAM."

[156] JEDEC, "Low Power Double Data Rate 4 (LPDDR4) SDRAM Specication," 2014.

[157] S. Electronics, "Mobile DRAM Stack Specification."

[158] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2017.

[159] S. Wang, H. C. Hu, H. Zheng, and P. Gupta, "MEMRES: A Fast Memory System Reliability Simulator," *IEEE Transactions on Reliability*, vol. 65, pp. 1783–1797, Dec 2016.

[160] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2012.

[161] "SPEC releases major new CPU benchmark suite." `https://www.spec.org/cpu2017/press/release.html`.

[162] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[163] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[164] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[165] S.-L. Gong, J. Kim, S. Lym, M. Sullivan, H. David, and M. Erez, "Duo: Exposing on-chip redundancy to rank-level ecc for high reliability," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 683–695, 2018.

[166] S. Jeong, S. Kang, and J.-S. Yang, "Pair: Pin-aligned in-dram ecc architecture using expandability of reed-solomon code," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

[167] J. Kim, M. Sullivan, S. Gong, and M. Erez, "Frugal ecc: efficient and versatile memory error protection through fine-grained compression," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2015.

[168] M. Gottscho, C. Schoeny, L. Dolecek, and P. Gupta, "Software-defined error-correcting codes," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 276–282, 2016.

[169] Y. H. Son, S. Lee, O. Seongil, S. Kwon, N. S. Kim, and J. H. Ahn, "Cidra: A cache-inspired dram resilience architecture," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 502–513, 2015.

[170] S. Kaneda and E. Fujiwara, "Single Byte Error Correcting Double Byte Error Detecting Codes for Memory Systems," *IEEE Transactions on Computers*, vol. 31, pp. 596–602, July 1982.

[171] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell Phase Change Memory: Toward an Efficient and Reliable Memory System," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pp. 440–451, 2013.

[172] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. A. Wolf, A. W. Ghosh, J. W. Lu, S. J. Poon, M. Stan, W. H. Butler, S. Gupta, C. K. A. Mewes, T. Mewes, and P. B. Visscher, "Advances and future prospects of spin-transfer torque random access memory," *IEEE Transactions on Magnetics*, vol. 46, pp. 1873–1878, June 2010.

[173] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai, "Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory," *J. Phys.: Condens. Matter*, vol. 19, pp. 165209–13, 04 2007.

[174] Intel, "Memory resiliency," *Technology Journal*, vol. 17, May 2013.

[175] R. A. Bheda, J. A. Poovey, J. G. Beu, and T. M. Conte, "Energy efficient phase change memory based main memory for future high performance systems," in *2011 International Green Computing Conference and Workshops*, pp. 1–8, July 2011.

[176] Suock Chung, K. Rho, S. Kim, H. Suh, D. Kim, H. Kim, S. Lee, J. Park, H. Hwang, S. Hwang, J. Lee, Y. An, J. Yi, Y. Seo, D. Jung, M. Lee, S. Cho, J. Kim, G. Park, Gyuan Jin, A. Driskill-Smith, V. Nikitin, A. Ong, X. Tang, Yongki Kim, J. Rho, S. Park, S. Chung, J. Jeong, and S. Hong, "Fully integrated 54nm stt-ram with the smallest bit cell dimension for high density memory application," in *2010 International Electron Devices Meeting*, pp. 12.7.1–12.7.4, Dec 2010.

[177] H. Li, X. Wang, Z. Ong, W. Wong, Y. Zhang, P. Wang, and Y. Chen, "Performance, power, and reliability tradeoffs of stt-ram cell subject to architecture-level requirement," *IEEE Transactions on Magnetics*, vol. 47, pp. 2356–2359, Oct 2011.

[178] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," pp. 256–267, 04 2013.

[179] H. Noguchi, K. Kushida, K. Ikegami, K. Abe, E. Kitagawa, S. Kashiwada, C. Kamata, A. Kawasumi, H. Hara, and S. Fujita, "A 250-MHz 256b-I/O 1-Mb STT-MRAM with advanced perpendicular MTJ based dual cell for nonvolatile magnetic caches to reduce active power of processors," in *2013 Symposium on VLSI Technology*, pp. C108–C109, June 2013.

[180] D. Shum, D. Houssameddine, S. T. Woo, Y. S. You, J. Wong, K. W. Wong, C. C. Wang, K. H. Lee, K. Yamane, V. B. Naik, C. S. Seet, T. Tahmasebi, C. Hai, H. W. Yang, N. Thiyagarajah, R. Chao, J. W. Ting, N. L. Chung, T. Ling, T. H. Chan, S. Y. Siah, R. Nair, S. Deshpande, R. Whig, K. Nagel, S. Aggarwal, M. DeHerrera, J. Janesky, M. Lin, H. J. Chia, M. Hossain, H. Lu, S. Ikegawa, F. B. Mancoff, G. Shimon, J. M. Slaughter, J. J. Sun, M. Tran, S. M. Alam, and T. Andre, "CMOS-embedded STT-MRAM arrays in 2x nm nodes for GP-MCU applications," in *2017 Symposium on VLSI Technology*, pp. T208–T209, June 2017.

[181] Y. Zhang, X. Wang, Y. Li, A. K. Jones, and Y. Chen, "Asymmetry of MTJ switching and its implication to STT-RAM designs," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1313–1318, March 2012.

[182] R. Takemura, T. Kawahara, K. Ono, K. Miura, H. Matsuoka, and H. Ohno, "Highly-scalable disruptive reading scheme for Gb-scale SPRAM and beyond," in *2010 IEEE International Memory Workshop*, pp. 1–2, May 2010.

[183] W. Wen, M. Mao, X. Zhu, S. H. Kang, D. Wang, and Y. Chen, "CD-ECC: Content-dependent error correction codes for combating asymmetric nonvolatile memory operation errors," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, Nov 2013.

[184] N. Kim and K. Choi, "A design guideline for volatile stt-ram with ecc and scrubbing," in *2015 International SoC Design Conference (ISOCC)*, pp. 29–30, Nov 2015.

[185] Yiming Huai, M. Pakala, Zhitao Diao, and Yunfei Ding, "Spin-transfer switching current distribution and reduction in magnetic tunneling junction-based structures," *IEEE Transactions on Magnetics*, vol. 41, pp. 2621–2626, Oct 2005.

[186] M. Jung, Y. Jin, and M. Shihab, "Area, power and latency considerations of stt-mram to substitute for main memory," in *The Memory Forum*, 2014.

[187] B. D. Bel, J. Kim, C. H. Kim, and S. S. Sapatnekar, "Improving stt-mram density through multibit error correction," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–6, March 2014.

[188] A. Raychowdhury, "Pulsed read in spin transfer torque (stt) memory bitcell for lower read disturb," in *2013 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pp. 34–35, July 2013.

[189] S. Mittal, J. S. Vetter, and L. Jiang, "Addressing Read-disturbance Issue in STT-RAM by Data Compression and Selective Duplication," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2017.

[190] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, "2mb spin-transfer torque ram (spram) with bit-by-bit bidirectional current write and parallelizing-direction current read," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, pp. 480–617, Feb 2007.

[191] W. Zhao, T. Devolder, Y. Lakys, J. Klein, C. Chappert, and P. Mazoyer, "Design considerations and strategies for high-reliable stt-mram," *Microelectronics Reliability*, vol. 51, no. 9, pp. 1454 – 1458, 2011. Proceedings of the 22th European Symposium on the RELIABILITY OF ELECTRON DEVICES, FAILURE PHYSICS AND ANALYSIS.

[192] R. Bishnoi, M. Ebrahimi, F. Oboril, and M. B. Tahoori, "Read disturb fault detection in stt-mram," in *2014 International Test Conference*, pp. 1–7, Oct 2014.

[193] S. Wang, A. Pan, C. O. Chui, and P. Gupta, "Tunneling negative differential resistance-assisted stt-ram for efficient read and write operations," *IEEE Transactions on Electron Devices*, vol. 64, pp. 121–129, Jan 2017.

[194] L. Yang, Y. Cheng, Y. Wang, H. Yu, W. Zhao, and A. Todri-Sanial, "A body-biasing of readout circuit for stt-ram with improved thermal reliability," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1530–1533, May 2015.

[195] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 325–334, ACM, 2012.

[196] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-link compression schemes: A value locality perspective," *IEEE Transactions on Computers*, vol. 57, pp. 916–927, July 2008.

[197] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ecc: Efficient and versatile memory error protection through fine-grained compression," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, (New York, NY, USA), pp. 12:1–12:12, ACM, 2015.

[198] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "Cop: To compress and protect main memory," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 682–693, June 2015.

[199] L. Chen, Y. Cao, and Z. Zhang, "Free ecc: An efficient error protection for compressed last-level caches," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 278–285, Oct 2013.

[200] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power i/o," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 49–58, March 1995.

[201] C. L. Chen and M. Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," *IBM Journal of Research and Development*, vol. 28, pp. 124–134, March 1984.

[202] C. Bracken and T. Helleseth, "Triple-error-correcting bch-like codes," *CoRR*, vol. abs/0901.1827, 2009.

[203] Micron, "DDR4 SDRAM." `https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf`, 2019.

[204] Everspin, "Embedded MRAM," 2019.

[205] X. Wang, M. Mao, E. Eken, W. Wen, H. Li, and Y. Chen, "Sliding basket: An adaptive ecc scheme for runtime write failure suppression of stt-ram cache," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 762–767, March 2016.

[206] S. Aly Ahmed, "Asymmetric and Symmetric Subsystem BCH Codes and Beyond," 04 2008.

[207] S. Wang, H. Lee, F. Ebrahimi, P. K. Amiri, K. L. Wang, and P. Gupta, "Comparative evaluation of spin-transfer-torque and magnetoelectric random access memory," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, pp. 134–145, June 2016.

[208] B. Jacob, S. W. Ng, and D. T. Wang, *Memory Systems: Cache, DRAM, Disk*. 01 2008.

[209] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, (New York, NY, USA), pp. 193–204, ACM, 2009.

[210] H. Liu, D. Chen, H. Jin, X. Liao, B. He, K. Hu, and Y. Zhang, "A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions," *CoRR*, vol. abs/2010.04406, 2020.

[211] D. Waddington, M. Kunitomi, C. Dickey, S. Rao, A. Abboud, and J. Tran, "Evaluation of intel 3d-xpoint nvdimm technology for memory-intensive genomic workloads," in *Proceedings of the International Symposium on Memory Systems*, pp. 277–287, 2019.

[212] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules," in *Proceedings of the International Symposium on Memory Systems*, pp. 288–303, 2019.

[213] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pp. 914–919, 2010.

[214] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the intel optane byte-addressable nvm," in *Proceedings of the International Symposium on Memory Systems*, pp. 304–315, 2019.

[215] A. Kokolis, D. Skarlatos, and J. Torrellas, "Pageseer: Using page walks to trigger page swaps in hybrid memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 596–608, 2019.

[216] Intel, "3D XPoint™: A Breakthrough in Non-Volatile Memory Technology." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[217] P. P. I. R. Life, "NEW MEMORIES CREATE NEW WAYS TO COMPUTE." https://pirl.nvsl.io/2019/11/07/new-memories-create-new-ways-to-compute/, 2019.

[218] VentureBeat, "Intel shows off working 3D XPoint memory that is 1000X faster than Flash." https://venturebeat.com/2015/08/18/intel-shows-off-working-3d-xpoint-memory-that-is-1000x-faster-than-flash/.

[219] TECHENABLEMENT, "3D XPoint Memory Poised to Revolutionize System Memory and Storage." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[220] 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 1.8v 8gb pram with 40mb/s program bandwidth," in *2012 IEEE International Solid-State Circuits Conference*, pp. 46–48, 2012.

[221] G. F. Close, U. Frey, J. Morrish, R. Jordan, S. C. Lewis, T. Maffitt, M. J. BrightSky, C. Hagleitner, C. H. Lam, and E. Eleftheriou, "A 256-mcell phase-change memory chip operating at ¡formula formulatype="inline"¿¡tex notation="tex"¿2+¡/tex¿¡/formula¿ bit/cell," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 6, pp. 1521–1533, 2013.

[222] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 2–13, 2009.

[223] "Process integration, devices structures," tech. rep., International Technology Roadmap for Semiconductors, 2007.

[224] 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 *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 14–23, 2009.

[225] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 383–394, 2010.

[226] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montaño, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA - 16*

2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pp. 1–11, 2010.

[227] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, "Safer: Stuck-at-fault error recovery for memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 115–124, 2010.

[228] B. Li, S. Shan, Y. Hu, and X. Li, "Partial-set: Write speedup of pcm main memory," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–4, 2014.

[229] S. Lavizzari, D. Ielmini, D. Sharma, and A. L. Lacaita, "Transient effects of delay, switching and recovery in phase change memory (pcm) devices," in *2008 IEEE International Electron Devices Meeting*, pp. 1–4, 2008.

[230] D. Mantegazza, D. Ielmini, A. Pirovano, A. Lacaita, E. Varesi, F. Pellizzer, and R. Bez, "Explanation of programming distributions in phase-change memory arrays based on crystallization time statistics," *Solid-State Electronics*, vol. 52, no. 4, pp. 584–590, 2008. Special Issue: Papers Selected from the Ultimate Integration on Silicon Conference 2007 - ULIS 2007.

[231] 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 *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 466–477, IEEE, 2011.

[232] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 83–93, 2010.

[233] W. Zhang and T. Li, "Helmet: A resistance drift resilient architecture for multi-level cell phase change memory system," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, pp. 197–208, 2011.

[234] "How 3D XPoint Phase-Change Memory Works." `https://pcper.com/2017/06/how-3d-xpoint-phase-change-memory-works/2/`.

[235] "gem5-20 Working Status of Benchmarks." `https://www.gem5.org/documentation/benchmark_status/gem5-20`.

[236] A. Chiang and I. Reed, "Arithmetic norms and bounds of the arithmetic an codes," *IEEE Transactions on Information Theory*, vol. 16, no. 4, pp. 470–476, 1970.