Integrating Computer Architecture and Coding Theory to Advance Emerging Memory Technologies

by

Georgios Mappouras

Department of Electrical and Computer Engineering
Duke University

Date:_____

Approved:

_____
Daniel J. Sorin, Advisor

_____
Robert Calderbank

_____
Alvin R. Lebeck

_____
Miroslav Pajic

_____
Benjamin C. Lee

Dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor
of Philosophy in the Department of
Electrical and Computer Engineering in the Graduate School
of Duke University

2020

ABSTRACT

Integrating Computer Architecture and Coding Theory
to Advance Emerging Memory Technologies

by

Georgios Mappouras

Department of Electrical and Computer Engineering
Duke University

Date:_____

Approved:

_____
Daniel J. Sorin, Advisor

_____
Robert Calderbank

_____
Alvin R. Lebeck

_____
Miroslav Pajic

_____
Benjamin C. Lee

An abstract of a dissertation submitted in partial
fulfillment of the requirements for the degree
of Doctor of Philosophy in the Department of
Electrical and Computer Engineering in the Graduate School of
Duke University

2020

Abstract

New memory technologies constantly emerge promising higher density, bandwidth, latency, and power efficiency comparing to traditional solutions. However, these technologies often suffer from substantial drawbacks like limited lifetime or low fault tolerance. These drawbacks prevent the integration of these technologies in modern computer systems and increase their cost of implementation. In this work, we utilize solutions both from the disciplines of computer architecture and coding theory to address the drawbacks of emerging memory technologies. By integrating computer architecture and coding theory we can design more optimized solutions, paving the way for emerging memory technologies to become viable and reliable options for modern computer systems.

More specifically we design MinWear codes to increase the lifetime of Flash memory, providing larger lifetime gains for smaller capacity costs comparing to prior work. We also design GreenFlag codes to address shift errors in 3D racetrack memory, providing double shift error detection and correction. Additionally, we enhance the fault tolerance of 3D-stacked DRAM with a two-level coding technique called Jenga. Jenga provides fault tolerance in all the granularity levels of the 3D-stacked DRAM with minimal performance overheads while outperforming prior solutions.

Dedication

Dedicated to my parents for their constant support, love, and guidance without which this achievement would not be possible.

Contents

List of Tables

List of Figures

Acknowledgments

Although completing a Ph.D. degree requires a lot of personal hard work, this long journey would not have come to completion without the guidance and support of my family, colleagues, friends, and collaborators that I would like to acknowledge here.

I would like to first acknowledge my advisors Prof. Daniel J. Sorin and Prof. Robert Calderbank that patiently guided me throughout my research. Specifically, I want to thank Prof. Daniel J. Sorin for tirelessly providing his feedback and insight, helping me to grow as a researcher. I also want to thank Prof. Robert Calderbank, for the many hours he dedicated, patiently introducing me to complex and fascinating concepts from the coding theory community. I would also like to thank Prof. Miroslav Pajic, Prof. Alvin R. Lebeck, and Prof. Benjamin C. Lee for their valuable feedback and contributions as members of my thesis committee.

I also had the opportunity to work with some great collaborators during my internship at AMD Research in Santa Clara. I want to especially thank my mentor Dr. Amin Farmahini-Farahani, as well as Dr. Gabriel H. Loh, Dr. Abhinav Vishnu, and Dr. Sudhanva Gurumurthi for their contributions to my research during and after my internship.

Additionally, I want to express my gratitude to all my friends and colleagues at Duke not only for their help and support but also for making this journey a fun experience.

I want especially thank Dr. Opeoluwa Matthews, Dr. Songchan Fan, Dr. Qiuyun Llull, Dr. Tamara Lehman, Atefeh Mehrabi, and all my labmates for their willingness to provide their help and feedback to better my work, papers, and talks, as well as for their friendship and the many amazing and fun experiences that I had the pleasure sharing with them.

Above all, I want to thank my parents for their constant support and love, as well as their many sacrifices that enabled me to achieve this milestone. I want to thank my father, Demetrios, who thought me to be patient and persistent, and my mother, Despoina, who thought me to be ambitious and self-driven. Also, a big thank you to my three beautiful sisters, Maria, Andriani, and Eleftheria.

Special thanks to Jingwen Du; thank you for your patience.

Additional Acknowledgments.

This material is based on work supported by the National Science Foundation under grants CCF-142-1177 and CCF-171-7602.

Material in Chapter 4 was previously published in the 2019 IEEE Conference on Dependable Systems and Networks (DSN)

Material in Chapter 5 was previously published in the 2017 IEEE International Conference on Computer Design (ICCD)

1. Introduction

Computer systems come in many sizes and forms, from personal computers to mobile devices, embedded systems in household electronics, or even large-scale datacenters. Regardless of their shape, all computer systems share two basic components, the compute units and the memory systems. The compute units are responsible for analyzing data and making useful calculations while the memory systems are responsible for storing the data. Although the quality of compute units is a very important aspect in a computer system, as faster compute units allow for faster computation, the focus in recent years has shifted to improving memory systems [35, 43, 46]. There are many reasons behind this shift, and we mention the most important here.

Regardless of how fast compute units are, they can process data only as fast as the memory system can provide them. As the number of compute units in a single system has kept increasing in recent years [29], the demand for faster memory technologies that can keep up with the large number of compute units has also increased. Furthermore, the amount of data that is consumed and stored has vastly increased as we entered the era of big data [8]. Memory systems must now be able to store larger amounts of data and be able to recover them reliably and efficiently when requested by the compute units. Naively increasing the capacity and/or the number of memory devices in a system is not a viable solution due to the large amount of power that memory devices consume [13, 78]. Due to these and other reasons, scientists and engineers shifted their focus from compute

units to memory systems, seeking technologies that could provide higher bandwidth, higher density, and lower power.

These efforts led scientists and engineers to discover new materials and exploit new chemical and physical properties that could be used to store information. Using these new materials engineers designed new and exciting emerging memory technologies [62, 60, 66]. Flash memory [62] uses floating gate transistors that can trap and isolate electrons. This enables Flash memory to store information with a significantly higher density than hard disk drives. Racetrack memory [60] uses phrenomagnetic nanowires where information can be represented based on the polarity of the electrons' spin, providing higher capacity and reduced power consumption compared to traditional DRAM. Through-silicon vias [66] allowed us to stack multiple DRAM channels together, creating 3D-stacked DRAM dies that can provide orders of magnitude higher memory bandwidth compared to traditional 2D DRAM.

All these advancements in memory technologies brought us one step closer to implementing better performing memory systems. However, as with every new technology, the advancements in memory systems also come with new challenges that limit the adoption of emerging memory technologies in modern computer systems. The floating gate transistors wear out every time they are erased limiting the lifetime of Flash memory. Racetrack memory and 3D-stacked DRAM have insufficient fault tolerance, a

prohibiting factor in using these technologies in computer systems that require high reliability.

Both computer architects and coding theorists seek to address the drawbacks of these technologies to enable wider use of them in modern computer systems. On one hand, architects tend to propose simple solutions that can be easily adapted to specific technologies. However, these solutions often lack the mathematical background and thus achieve sub-optimal results. Theorists on the other hand, propose mathematically structured solutions that can provide clear tradeoffs, and (if possible) an understanding of the optimal theoretical solution for a given problem. However, theorists typically propose solutions for ideal, theoretical models that do not accurately resemble a realistic memory system. Furthermore, many optimization opportunities are missed as the theoretical solutions do not take into account the specific attributes of a given technology.

In this work, we attempt to integrate both computer architecture and coding theory to achieve solutions that are both mathematically sound, but also are optimized for specific memory technologies. Through this work, we will demonstrate that by merging the two disciplines we can advance emerging memory technologies and make them viable solutions for modern computer systems. In Chapter 3 we will present MinWear codes that are designed specifically for Flash memory and can provide more lifetime gains for less capacity cost compared to prior work. Additionally, in Chapter 4 we will show how we can utilize existing coding techniques and combine them with architectural solutions to

enhance the fault tolerance of racetrack memory. Our proposed solution, which we refer to as GreenFlag, enables racetrack to be a viable main memory solution. In Chapter 5 we present Jenga, a scheme that increases the reliability of 3D DRAM by providing fault tolerance in every level of the 3D DRAM. Jenga has minimal performance impact compared to an uncoded baseline and significantly better performance than prior work.

2. Emerging Memory Technologies

In this chapter, we give a short background and describe the technologies that we target throughout this work. We present Flash memory, racetrack memory, and high bandwidth memory (HBM). These three memory technologies have very different characteristics (e.g., bandwidth, density, etc.) that are summarized in Table 2.1. For comparison, we also include DRAM and hard disk drives (HDD). The reported characteristics are estimations based on current trends. These values will vary depending on the specific generation of each technology.

2.1 Flash Memory

One of the memory technologies that we target in this work is Flash memory. Flash memory is an exciting non-volatile memory (NVM) technology that has been widely used as a storage solution and an alternative option to HDDs.

Flash memory can provide orders of magnitude lower read and write latency, higher read and write bandwidths, and higher density than HDDs. Additionally, as Flash memory consists of silicon transistors it can be easily intergraded in smaller systems like

Table 2.1: Comparing different memory technologies.

	Flash [40]	Racetrack [84]	HBM [48]	DRAM [4]	HDD [64]
Volatile	No	No	Yes	Yes	No
Effective Density (F^2) [61]	4-5	1-2	6	6	N/A
Read Latency (ns)	10^4	3	20-30	20-30	10^7
Write Latency (ns)	10^5	10	20-30	20-30	10^7
Bandwidth (GB/s)	1	N/A	128-256	10-20	0.1

embedded devices, something that was infeasible with HDDs. These features make Flash memory a very attractive option for storage in a variety of computer systems (personal computers, embedded systems and enterprise servers).

However, Flash memory suffers from a limited lifetime [71, 12, 25]. A Flash transistor (a.k.a., Flash cell) can only be programmed (i.e., written) a fixed number of times before wearing out. This limitation can be crucial for systems with large lifetime spans (e.g. embedded devices, enterprise servers). In Chapter 3.1 we further discuss the limited lifetime of Flash memory and propose a novel solution that enables Flash memory to extend its lifetime and become a viable, more cost-efficient solution for such systems.

We now explain the organization and interface of Flash memory.

2.1.1 Flash Organization

The core element in Flash memory is the Flash cell. Depending on the Flash memory technology, the Flash cell can store one (SLC), two (MLC), or three (TLC) bits by being able to differentiate between 2 (Level 0 – Level 1), 4 (Level 0 – Level 3) or 8 (Level 0 – Level 7) levels of voltage respectively as seen in Figure 2.1. A Flash cell can only be programmed (written) by increasing its voltage level. To decrease its voltage level, a Flash cell must be erased, always setting its voltage level to Level 0.

Multiple Flash cells are used to construct a Flash page. For example, 32,768 SLCs could be used to construct a single 4KB Flash page. Alternatively, 32,768 MLCs (TLCs)

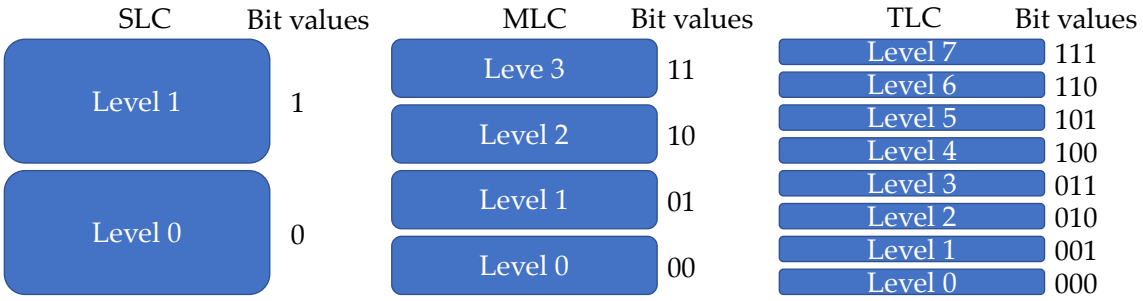


Figure 2.1: The different types of Flash cells and the bits they store.

could construct two (three) 4KB Flash pages. Flash pages are the minimum read and write granularity in a Flash device. Due to the underlying physics of the Flash cells, the bits in a Flash page can only be programmed from 0 to 1. To reset the bits back to 0, all cells in the Flash page must be erased.

Multiple Flash pages (256-512) are grouped to create a Flash block. A Flash block is the minimum erase granularity in a Flash memory device. Thus, to erase any Flash page, all Flash pages in the same Flash block must be erased.

2.1.2 Flash Interface

A Flash device is managed by the Flash translation layer (FTL), a software interface that is responsible for managing read and write requests of memory pages from the OS. Without loss of generality, we assume that memory pages are the same size as Flash pages. To store a memory page (*M*) to a Flash device, the FTL selects a clean (erased) Flash page (*F*) and programs the cells of that page accordingly. A mapping from memory page *M* to the flash page *F* is also stored so that the FTL can correctly recover the data in any subsequent read requests on memory page *M*.

To update the value of M with M', the FTL needs to mark the Flash page F as invalid, select a new, clean Flash page F' and again program the cells of F' accordingly. To reclaim invalidated Flash pages, the FTL needs to invoke its garbage collection mechanism. During garbage collection, the FTL selects a block to be erased (B). Any valid pages on block B are first copied to a clean, already erased block B'. Then the entire block B is erased, thus reclaiming all invalidated Flash pages in B. An overview of a Flash device with the FTL, Flash blocks, and Flash pages can be seen in Figure 2.2.

2.2 Racetrack Memory

Racetrack memory is a relatively new NVM technology that has shown great promise as a main memory (DRAM alternative) solution and even a last level cache option [76, 75, 84, 72, 56]. As we can see in Table 2.1, racetrack memory provides better read and write latency, energy per bit, and density compared to a typical DRAM. Furthermore, contrary to other NVM technologies, racetrack does not suffer from a limited lifetime.

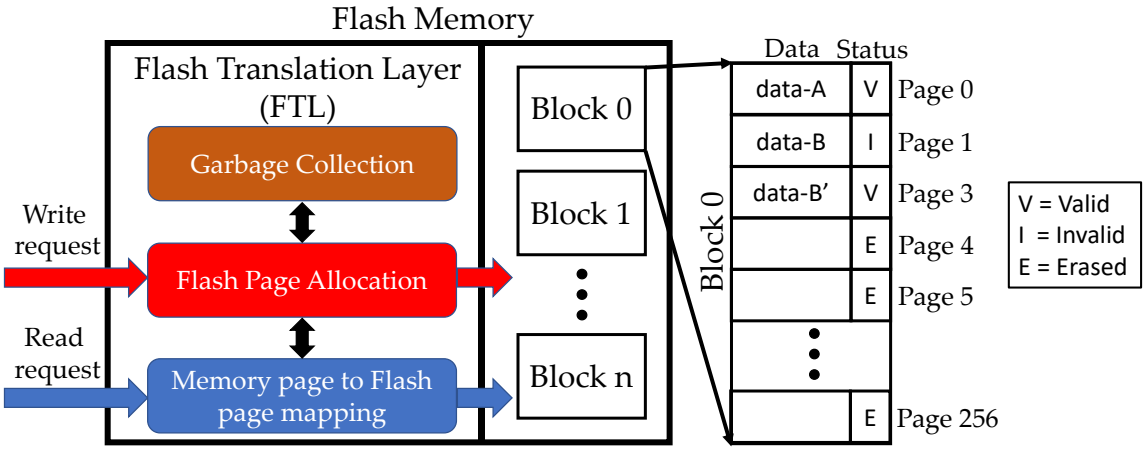


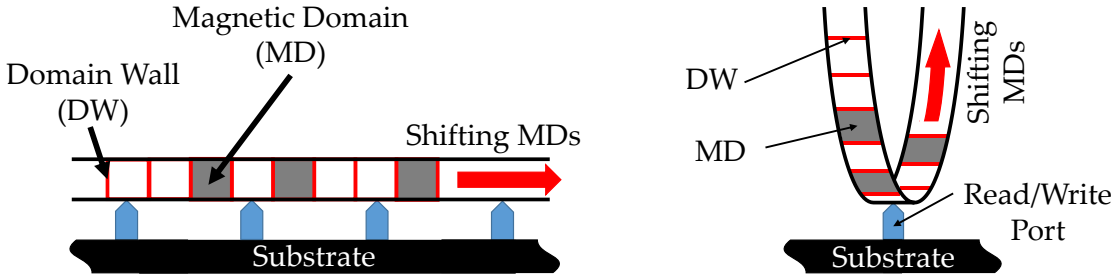
Figure 2.2: A Flash device overview.

However, racetrack memory has not yet found its way to real-life products. One of the reasons behind that is that racetrack memory has a somewhat unique error model.

Racetrack memory is prone to a new (for memory systems) error type, referred to as shift errors, where an error in one bit position can affect the bit values read or written in all other bit positions. In Chapter 4.1 we further discuss the details of shift errors and propose a novel solution to address them, eventually opening the way for racetrack memory to becoming a mainstream memory solution. We now give a short background on the physical properties of racetrack and explain how information is stored.

2.2.1 Physical Properties of Racetrack memory

The basic components of racetrack memory are the magnetic strip (track), and the read/write ports that sit upon a substrate as seen in Figure 2.3. The track stores information in terms of magnetic regions, known either as magnetic domains (MDs) [60, 59, 86, 34] or magnetic objects called skyrmions [73, 26, 33], depending on the underlying technology used to create the memory. For the purpose of this work, the two technologies



a. 2D Racetrack b. 3D Racetrack

Figure 2.3: Racetrack memory in (a) 2D, and (b) 3D geometries.

are equivalent and thus we assume MDs for the rest of this document. These magnetic regions can be shifted along the track by applying current that runs parallel with the track.

Another physical property of this technology is its geometry. Two different geometries have been proposed; a line-shaped 2D track (Figure 2.3a) and a U-shaped 3D track (Figure 2.3b). There are two main differences between the 2D and 3D racetrack memory. The first difference is that the 2D racetrack can have multiple read/write ports per track as the track is always adjacent to the substrate, whereas the 3D racetrack only allows for a single read/write port at the bottom of the U-shaped track. The 3D racetrack is projected to achieve significantly greater density because more tracks can be packed per area unit due to its geometry. For the rest of this document, we assume 3D racetrack with a single read/write port. However, for simplicity of illustration, our figures will use a 2D representation.

2.2.2 Racetrack System Operations

Regardless of the physical geometry of the racetrack memory, the system representation is the same and can be seen in Figure 2.4. That is, a track can store a number of bit positions (logical 1 or 0). To read or write a bit, the bit position must be shifted over the read/write port. By inducing current that runs parallel with the track, bit positions can be shifted along the track. The track and the ports are physically fixed and cannot move. A port can access the bit position above it by either sensing if an MD is present (i.e., read)

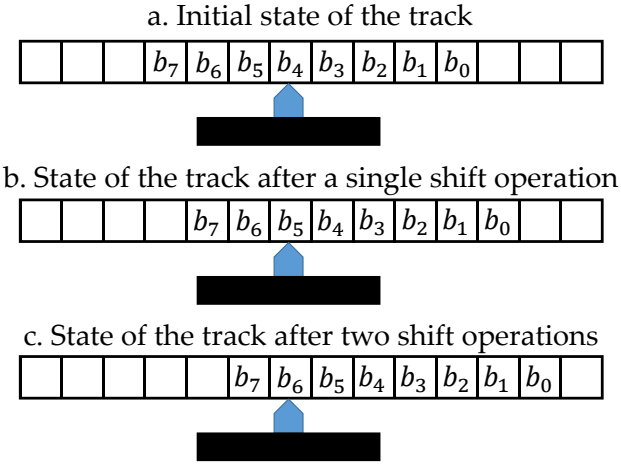


Figure 2.4: Different states of the same racetrack; (a) initial state, (b) after a shift operation, and (c) after two shift operations.

or inducing an MD (i.e., write). To read (write) multiple bits, consecutive read (write) and shift operations must be performed as shown in Figure 2.4a-c.

2.3 3D-Stacked DRAM: High Bandwidth Memory

A relatively recent advancement in memory technologies was the implementation of 3D-stacked DRAM [48, 38]. By integrating multiple DRAM dies on one chip-stack, higher density per unit area can be achieved (even though the memory cell size is the same with typical 2D DRAM). Furthermore, this integration allows for a higher number of DRAM channels per stack, that through a wide interconnection interface can provide a larger memory bandwidth. Thus, 3D-stacked DRAM is a very attractive main memory option for systems that require high bandwidth (e.g., GPUs).

However, integrating multiple DRAM dies in one stack also has its drawbacks. For example, faulty DRAM channels cannot be individually replaced as with typical 2D DRAM as they are part of the whole DRAM stack. In Chapter 5.1 we further discuss the

fault model of 3D-stacked DRAM and how it impacts the fault tolerance of this technology. We also present a novel solution to increase the fault tolerance of 3D-stacked DRAM, so that it can be a viable solution to systems that require both high bandwidth and high fault tolerance, like scientific computation, autonomous driving, and others.

Currently, there are two popular product implementations of 3D-stacked DRAM, High Bandwidth Memory (HBM) [48] and Hybrid Memory Cube (HMC) [38]. The main difference between HBM and HMC is that they use different internal organizations in terms of forming memory channels. In this work, we focus explicitly on HBM as it is more widely used in products [6, 5, 2].

HBM typically provides 4 or 8 dies per stack. Each die is divided into two channels. Each channel is organized in bank groups (typically 4 per channel) that contain banks (typically 4 per bank group). The process of accessing a row in a bank is similar to traditional DRAM technologies. A row needs to first be opened to read or write it and accessing an already open row is faster than a closed row.

A key feature of HBM is that it provides a wide interconnection interface for each channel by making use of through-silicon via (TSV) technology [66]. TSVs allow for point-to-point connections that run vertically through the silicon, creating a dense and compact interface. Typically, each HBM channel has 128 TSVs forming a 128 bit-wide interface per channel. To further maximize the bandwidth gains of HBM, each memory channel is

controlled by a dedicated memory controller, allowing channels to work independently of each other.

As HBM channels are completely independent, they are also responsible for storing ECC bits that provide bit-flip error protection for their stored data. In other words, data and their respective parity are collocated on the same channel. A typical error correction code is Hamming(72,64), which allows for single error correction and double error detection (SECDED). Figure 2.5 summarizes the features of an HBM stack.

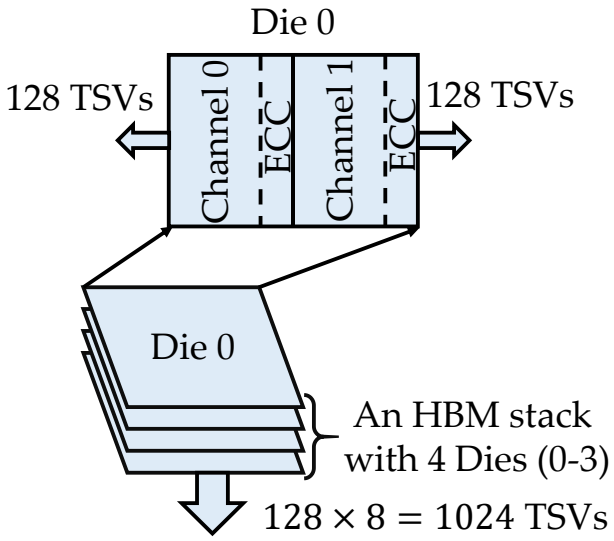


Figure 2.5: Organization of a typical HBM stack.

3. Increasing Lifetime in Flash Memory

In this chapter of the dissertation document, we discuss Flash lifetime and how we could use coding to extend it. We start by giving an extensive explanation of lifetime in Flash memory and the parameters it depends on in Chapter 3.1. We then give an overview of prior work on extending Flash lifetime, in Chapter 3.2. Next, in Chapter 3.3 we present and define precise metrics that will help us quantify and evaluate the lifetime gains in Flash memory. In Chapter 3.4 we present our proposed solution to extend Flash lifetime, called MinWear Coding (MWC). In Chapter 3.5 we explain the methodology we use to evaluate MWC and finally, in Chapter 3.6 we extensively evaluate MWC and compare it to prior work.

We summarize the contributions of this chapter here:

- We provide a set of precise metrics that we use to fairly evaluate and compare coding schemes on improving Flash lifetime.
- We present MinWear Coding, a new family of bias codes that provide better results than prior work, offering the largest Flash lifetime gains for the least capacity cost.
- We show that MinWear Coding can provide an unbounded amount of tradeoffs between lifetime gain and capacity cost.

3.1 Flash Memory Lifetime

Flash cells can only endure a limited number of program/erase cycles [71, 12, 25], after which a cell can no longer reliably retain its voltage level. When a cell is erased, charged particles get trapped in the oxide of the cell, between the floating gate and substrate. As the number of charged particles in the oxide increases, conducting paths may be created between the floating gate and the substrate. If this occurs, the cell can no longer trap electrons in the floating gate (i.e., be programmed).

The number of program/erase cycles that a cell can endure before becoming unreliable is a random variable with a distribution that depends on the technology generation, the type of cell, and manufacturing process variability. Typically, SLCs can tolerate the most program/erase cycles, followed by MLCs and then TLCs. Recent research has also demonstrated that the number of program/erase cycles a cell can endure depends on the cell's voltage level at the time of erase [81, 51, 37, 52, 69].

Flash typically uses an ECC to tolerate bit errors due to the first cell(s) that wear out in each codeword (as well as random errors). However, as a Flash device is rewritten over time, the number of cells failing increases, and the bit error rate (BER) eventually increases beyond the error correction capabilities of the ECC, at which point the Flash ceases to be useful [52, 22, 24, 18].

We provide an example to understand the dependencies between the number of program/erase cycles and the voltage levels at the time of erase. Assume that the BER at

time 0 is BER_0, while the BER at the end of its lifetime is BER_{end}. We could then measure how fast the BER increases from BER_0 to BER_{end}, when constantly programming all cells to a specific level. In Figure 3.1 we visualize our example, for a specific MLC technology based on real data reported from prior work [52]. As shown in Figure 3.1, cells in a Flash block wear out faster (and thus BER reaches BER_{end} faster) when programmed at higher levels. Similar results have been reported in other works as well [81, 51, 37, 52, 69].

3.2 Prior Work on Flash Lifetime

There are three complementary approaches for addressing Flash's limited lifetime. The first approach is wear-leveling [14, 16, 15], which seeks to maintain the same wear rate for all Flash cells. The second approach is using error correction codes (ECC) to tolerate failures due to cells that have worn out [22, 24, 18]. The third approach is using endurance codes to extend the lifetime of Flash. Often, all three approaches are combined

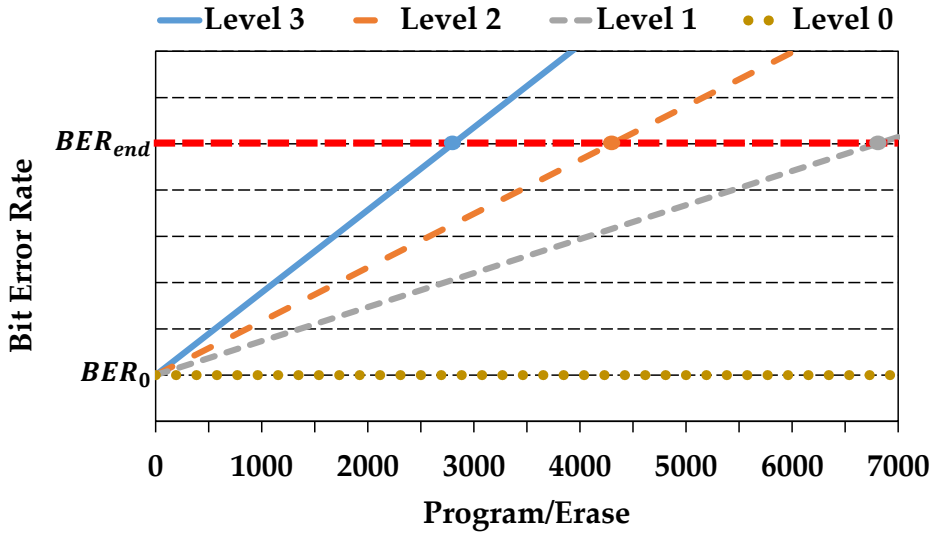


Figure 3.1. BER increase when Flash cells are constantly programmed to and erased from levels 0, 1, 2, and 3 in an MLC device.

to maximize Flash lifetime. This work focuses on endurance codes, of which there are two varieties: rewriting codes and bias codes.

3.2.1 Rewriting Codes

Almost all prior endurance codes for Flash have been rewriting codes that seek to increase the number of times a Flash page can be rewritten before needing to be erased. The intuitive premise behind rewriting codes is that cells wear out when erased and thus, if we increase the number of times a cell can be re-programmed before erased, we can increase the average program/erase cycles of each cell, thus extending the overall lifetime of Flash memory. However, as we mentioned in Chapter 3.1, not all erasures are equal. Erasing a cell from a higher level causes more wear than erasing a cell from a lower level (e.g., erasing from Level 1 is costlier than erasing from Level 0). Thus, because the objective function of rewriting codes ignores the cell levels, they can underperform. Furthermore, all rewriting codes require performing a read operation before each write operation, thus adding to performance and energy overheads. We now give specific examples of rewriting codes.

3.2.1.1 Write-Once-Memory Codes (WOM)

WOM codes [65, 79, 21, 9, 42, 44, 80] are lattice-based rewriting codes that avoid erasures by only programming 0s to 1s. When rewriting a codeword is no more feasible, the entire codeword is erased. A commonly used WOM code is WOM(3,2) [79], in which

2 input dataword bits are mapped to 3 output codeword bits, enabling up to 3 rewrites per codeword.

3.2.1.2 Coset Codes

Coset codes [27, 28, 55, 36] are also used to construct rewriting codes. With coset codes, each dataword can be mapped to multiple codeword representatives. The goal is to choose the representative that will maximize the number of consecutive rewrites. MFCs [55] are an example of rewriting codes based on coset codes. For a given input dataword, MFCs provide multiple output codeword representatives. Any of the output representatives can be selected as the codeword to be stored in the Flash device. MFCs provide a metric function that selects the representative that will maximize the number of rewrites for a given Flash page.

Other rewriting coding techniques have also been proposed [82, 54].

3.2.2 Bias Codes

Bias codes (also known as shaping codes) [52, 69] seek to reduce the average wear that each cell endures at erasure by biasing writes to favor programming cells at lower levels. Because bias codes are not rewriting codes, no additional read operations are required before each write operation. We know present examples of bias codes.

3.2.2.1 Dynamic Shaping Codes (DSC)

DSC [52] are biased programming codes where a dynamic encoding mapping is created per Flash page, such that dataword bit patterns that appear more often are

mapped to codewords that have cells in lower levels. However, because the mapping is dynamic (i.e., depends on the bit patterns in the datawords), different mappings must be stored for each Flash page. Furthermore, DSC can only provide lifetime gains if the same bit patterns are often repeated in a Flash page.

3.2.2.2 Flip-N-Write (FnW)

FnW [20] is an easily implemented, low-cost endurance code. More specifically, FnW is the simplest type of coset code, where each dataword has only two codeword representatives. Given a dataword, we can select as a codeword representative either the unchanged dataword or the compliment of the dataword. An extra bit at the end of the codeword indicates if the bits were flipped or not. However, contrary to MFCs, FnW does not aim to increase rewrites but rather to minimize bit flips. In Flash memory, where a flash page is erased (all bits set to zero) before programmed, FnW simply selects between the original dataword or its complement to minimize the number of bits programmed at logical 1. Thus, FnW can be seen as a bias programming code that favors logical 0 over logical 1 and could potentially reduce the average level that cells are programmed at. Similarly to DSC, the lifetime gains of FnW heavily depend on the dataword bit patterns.

The solution we propose in this work to extend Flash lifetime (MinWear Coding) is also a form of bias coding but without the drawbacks of the two codes described above.

3.3 Precise Metrics for Evaluation & Comparison

We now present a complete set of metrics that can be used to project how a coding scheme will affect the lifetime of Flash memory. We use these metrics to evaluate both our proposed solution and prior work.

3.3.1 Code Cost

Given a code that encodes datawords to codewords we can define its rate as the number of bits per dataword divided by the number of bits per codeword (as typically done in coding theory). The cost of a code is related to its rate. For a storage medium like Flash, if we fix the raw capacity, then rate is proportional to its host-visible storage capacity. For example, a rate-2/3 code would have only 2/3 as much host-visible capacity as raw capacity, because the other 1/3 is coding overhead.

Note that some codes may have rates greater than one if they reduce the number of raw bits stored in memory (e.g. compression). Codes, that add redundant bits (e.g., ECCs) have rates less than one.

3.3.2 Flash Cell Susceptibility to Wear

We present one metric that we introduce here to characterize the Flash cells' inherent susceptibility to wear.

3.3.2.1 Wear Profile (W^{prof})

The number of program/erase cycles a Flash block can tolerate before its BER increases to BER_{end} depends on the levels of its cells just before erasure. If we constantly

program all cells to Level i and then erase them, we define the expected amount of program/erase cycles to be C_i. Obviously, we never program any given cell to the same level every time, but we can assume this behavior for purposes of characterizing cells. We now define the wear profile of a Flash technology with L levels as,

$$\boldsymbol{W^{prof}} = \left[\frac{1}{C_0}, \frac{1}{C_1}, \ldots, \frac{1}{C_{L-1}}\right] = [W^{prof}{}_0, W^{prof}{}_1, \ldots, W^{prof}{}_{L-1}]$$

We then express every $W^{prof}{}_i$ in terms of $W^{prof}{}_{L-1}$; that is, for $i < L-1$, we have $W^{prof}{}_i = a_i * W^{prof}{}_{L-1}$. Now we can rewrite W^{prof} as,

$$\boldsymbol{W^{prof}} = [a_0, a_1, \ldots, 1] \times W^{prof}{}_{L-1} = \boldsymbol{a} \times W^{prof}{}_{L-1}$$

As an example, consider an MLC (4-level) technology where $\boldsymbol{W^{prof}} = [0, 0.2, 0.5, 1] \times W^{prof}{}_3$, meaning that erasing a cell in Level 0 incurs zero wear, but erasing from Levels 1 and 2 incur 0.2× and 0.5×, respectively, the wear of erasing from Level 3.

3.3.3 Runtime WearMetrics

Given the Wear Profile of the Flash cells, we want to compute the wear that actually occurs at runtime. This runtime wear is a function of W^{prof} and the cell levels that actually occur at erase time.

3.3.3.1 Cell Program Probability (P)

Given a wear-leveling technique and/or endurance coding scheme, each cell has a certain probability of being programmed to a certain level before erasure. We express this probability for cells in a Flash block as $\boldsymbol{P} = [P_0, P_1, \ldots, P_{L-1}]$, where P_i is the probability for a cell to be at Level *i* just before erasure.

3.3.3.2 Average Runtime Wear (W_{ave})

We define the average runtime wear of a Flash block as the inner product between $\boldsymbol{W^{prof}}$ and $\boldsymbol{P}$,

$$W_{ave} = \boldsymbol{W^{prof}} * \boldsymbol{P^T}$$

W_{ave} expresses the average rate at which cells wear out when erased. For our MLC example and assuming random data (i.e., all levels are equally likely),

$$W_{ave} = \left([0,0.2,0.5,1] \times {W^{prof}}_3\right) * [0.25, 0.25, 0.25, 0.25]^T = 0.425{W^{prof}}_3$$

3.3.4 Lifetime Metrics

3.3.4.1 Lifetime (LT)

We measure the lifetime (LT) of a Flash block as the expected number of program/erase cycles that the block can endure. Lifetime is thus calculated as,

$$LT = \frac{PageRewrites}{W_{ave}}$$

PageRewrites is the average number of times that a page is rewritten before erased if a rewriting code is used. For MWC, which is not a rewriting coding scheme, *PageRewrites* equals 1. This new metric allows us to account for the effect of rewriting codes and will help us to fairly compare our work against such codes. For our MLC example, assuming an uncoded baseline with random data,

$$LT_{base} = \frac{1}{0.425{W^{prof}}_3} = \frac{2.35}{{W^{prof}}_3}$$

Now consider instead a hypothetical rewriting code, where each page is written two times before each erase, but changes the cell program probability from random to $\boldsymbol{P} = [0.1, 0.2, 0.2, 0.5]$. This code has,

$$W_{ave} = \left([0, 0.2, 0.5, 1] \times {W^{prof}}_3\right) * [0.1, 0.2, 0.2, 0.5]^T = 0.64 {W^{prof}}_3$$

and it thus has a lifetime of,

$$LT_{code} = \frac{2}{0.64 W_3} = \frac{3.125}{{W^{prof}}_3}$$

3.3.4.2 Lifetime Gain (LG)

We define lifetime gain of a coding scheme as,

$$LG = \frac{LT_{code}}{LT_{base}}$$

where $\boldsymbol{LT_{code}}$ is the lifetime of the code and $\boldsymbol{LT_{base}}$ is the lifetime of an uncoded baseline. For our running example, the lifetime gain of the hypothetical rewriting code is,

$$LG_{code} = \frac{3.125}{{W^{prof}}_3} / \frac{2.35}{{W^{prof}}_3} = 1.33$$

By definition, the lifetime gain of the baseline always equals 1. Typically, in prior work [79, 55, 36], a rewriting code that could rewrite a page 2 times before erasure would be considered to have a lifetime gain of 2. However, with our more precise definition of lifetime gain, we account for the differences in wear when erasing from different levels, and we thus get a more precise result.

3.3.4.3 Aggregate Gain (AG)

The most important metric is aggregate gain because it reflects both the lifetime benefit of a code as well as its cost. We borrow the definition of aggregate gain from [55] as,

$$AG = LG * rate$$

An AG that is less than (or equal to) one indicates that the corresponding coding technique is not worth implementing, because we can achieve the same lifetime gain for less (the same) capacity cost with the uncoded baseline. The greater the aggregate gain, the less costly the code is at providing lifetime gain. For our running example, if we assume the rate of the hypothetical rewriting code to be 0.5, we get,

$$AG_{code} = 1.33 * 0.5 = 0.665$$

Thus, this code would not be selected for implementation because it is highly inefficient.

3.4 MinWear Codes

We now present MinWear Coding (MWC), a new family of codes that we have developed to increase the lifetime of Flash memory. The intuition behind MWC is that we want to reduce the probability of cells being programmed at higher levels. Thus, we can decrease the average runtime wear of cells and increase the lifetime. MWC maps dataword bits to the levels to which the Flash cells should be programmed. That is, the codeword is represented in terms of cell levels, not bits, which is somewhat unusual.

For a given dataword size m, we can construct MWC codes with any codeword size n as long as $L^n \geq 2^m$. Thus, we denote an MWC code with L-level cells, dataword size

m bits, and codeword size n cells as MWC(L, n, m). An MWC(L, n, m) code has a rate of $\frac{m}{log(L)*\mathrm{n}}$ where $log(L)$ is the number of bits that an L-level cell typically stores in an uncoded Flash device.

3.4.1 MWC Construction

To construct an MWC(L, n, m) code we first need to know the wear profile W^{prof} of the target Flash device. For a given Flash technology, we can determine W^{prof} using a similar methodology as in [52]. We constantly program the cells of a Flash block to Level i and erase them, counting the average program/erase cycles C_i until the BER increases to BER_{end}. We repeat this process for every $i \in [0, L-1]$ and calculate,

$$\boldsymbol{W^{prof}} = \left[\frac{1}{C_0}, \frac{1}{C_1}, \ldots, \frac{1}{C_{L-1}}\right] = [a_0, a_1, \ldots, 1] \times {W^{prof}}_{L-1} = \boldsymbol{a} \times {W^{prof}}_{L-1}$$

as defined in Chapter 3.3.

Based on W^{prof} we can now sort all theoretically possible codewords of MWC(L, n, m)—that is, all strings of n cells—based on a cost metric function. The cost of each codeword is calculated by summing up the wear of each cell. For example, the codeword "00123" has a total cost $= (a_0 + a_0 + a_1 + a_2 + 1) \times {W^{prof}}_3$. Note that we can ignore W_3 and simply express cost $= (a_0 + a_0 + a_1 + a_2 + 1)$ without losing any information. Thus, we only need $\boldsymbol{a}$ to apply our metric function. Then we select the 2^m n-cell strings with the least cost. For specific values of n and m this method ensures the mapping that induces the least amount of wear (given random input data) and thus the most lifetime gain. However, for a given m, different values of n would lead to different tradeoffs between

Table 3.1. MWC(4, 3, 3).

	Input	$\boldsymbol{MWC(4,3,3)}$
Index	Dataword Bits $[d_2, d_1, d_0]$	Codeword MLCs Levels $[c_2, c_1, c_0]$
0	0 0 0	0 0 0
1	0 0 1	0 0 1
2	0 1 0	0 1 0
3	0 1 1	1 0 0
4	1 0 0	0 0 2
5	1 0 1	0 2 0
6	1 1 0	2 0 0
7	1 1 1	0 1 1

lifetime gain and rate. As we later show in Chapter 3.6, with a brute-force search, we can determine the value of n that achieves the best aggregate gain for a given m.

In Table 3.1, we present MWC(4,3,3) for $\boldsymbol{a}$=[0,0.3,0.5,1]. The first column is the decimal representation of a dataword (index), the second column lists the datawords, and the third column lists the codeword MLC levels. Although this code guarantees that we have selected the 2^3 codewords that cause the least wear (for a given $\boldsymbol{a}$), it presents us with two important challenges. The first challenge is constructing an efficient encoding and decoding algorithm for this mapping. The second challenge is that some cells have a higher probability of being programmed than others, thus resulting in uneven wear. More specifically, by looking at the third column of Table 3.1, we can observe that although on average, cells are programmed at Level 1 with probability 0.208, cells c_0 and c_1 are programmed at Level 1 with a higher probability of 0.25 and cell c_2 with a lower probability of 0.125. Hence, cells c_0 and c_1 would wear out faster than cell c_2 and that

would limit the lifetime gains of this code. We now discuss how we can address both these challenges.

3.4.2 MWC Encoding & Decoding

Although we could opt to use look-up tables to encode and decode MWC(L, n, m) codes, that could lead to significant capacity and latency overheads for large values of m. To avoid these overheads and achieve efficient encoding and decoding algorithms, we instead use a method known as shell mapping [47]. First, we map our codewords to shells (i.e., equivalence classes) of the same total wear cost. For our MWC(4,3,3) example, we have 4 shells. The first shell contains the single codeword with cost=0 (i.e., the codeword with index 0 in Table 3.1). The second shell contains codewords with cost=0.3 (i.e., the codewords with indices 1, 2, and 3 in Table 3.1), the third shell the codewords with cost=0.5 (i.e., the codewords with indices 4, 5, and 6 in Table 3.1), and finally the fourth shell the codeword with cost=0.6 (i.e., the codeword with index 7 in Table 3.1).

Each dataword is mapped to the same shell as its respective codeword. A unique encoding and decoding algorithm is designed for each shell. Before encoding (decoding) a dataword (codeword), we first determine in which shell it belongs. In Table 3.2 we show the algorithms for encoding and decoding MWC(4, 3, 3). As we can observe from Table 3.2, this divide-and-conquer method allows us to have very simple encoding and decoding algorithms for each shell.

Table 3.2. Encoding and Decoding for MWC(4,3,3).

Encoding Algorithm for MWC(4,3,3):

Input: *m-bit dataword* = $[d_2, d_1, d_0]$
Output: *n-cell codeword*

n-bit codeword = $[c_2, c_1, c_0] = [0,0,0]$;

if(m==0) Shell=1;
else if(0<m<4) Shell=2;
else if(3<m<7) Shell=3;
else Shell=4;

if (Shell==1) return codeword; // Shell 1
if (Shell==2) idx=m-1; c_{idx}*= 1;* // Shell 2
if (Shell==3)idx=m-4; c_{idx}*= 2;* // Shell 3
if (Shell==4) codeword= $[0,1,1]$; // Shell 4

return codeword;

Decoding Algorithm for MWC(4,3,3):

Input: *n-cell codeword* = $[c_2, c_1, c_0]$, $a = [0, 0.3, 0.5, 1]$
Output: *m-bit dataword*

m-bit dataword = $[d_2, d_1, d_0] = [0,0,0]$;
cost = $a * codeword^T$; // *total wear of codeword*

if (cost == 0) return dataword; // Shell 1
if (cost == 0.3) // Shell 2
  find idx where $c_{idx} \neq 0$; *dataword = 1+idx;*
if (cost == 0.5) // Shell 3
  find idx where $c_{idx} \neq 0$; *dataword = 4+idx;*
if (cost == 0.6) // Shell 4
  dataword = 7;

return dataword;

3.4.3 MWC Wear-leveling

To address the uneven wear in MWC, we use a simple but effective wear-leveling technique that has no rate overhead. This wear-leveling technique is based on how the

FTL maps memory pages to Flash pages. For simplicity, and without loss of generality, we assume that a memory page has the same capacity as a Flash page.

As we explained in Chapter 2.1.2, over its lifetime, a Flash page *F* will store different memory pages *M* with different addresses. As the values of these addresses are expected to be random for a given Flash page *F*, we will use them to perform wear-leveling. We use $z = \lfloor log_2(n) \rfloor$ bits from the address of the memory page ($A_{z-1}, \dots, A_1, A_0$). Let x be the integer representation of the address bits. We then perform x clockwise cyclic rotations on the cell levels of each codeword after encoding. For example, if we were to store the codeword 011 from MWC(4,3,3) with $x = 1$, we would instead store 110. If $x = 0$ we would store the codeword as is. Before decoding a codeword, we first perform a counterclockwise cyclic rotation by x. Note that the value of x can always be retrieved from the FTL mapping and no additional information needs to be stored.

3.5 Methodology

We now describe the methodology used to evaluate MWC (as well as prior work on endurance codes). Our evaluation is based on the metrics we defined in Chapter 3.3.

We simulate repeated writes to a single Flash block. The block has 128 rows of physical cells, comprising 128 Flash pages for SLCs and 256 Flash pages for MLCs. Each Flash page has a capacity size of 4KB. Store requests are first encoded based on the coding scheme in question and then the encoded data are stored in a page of the block.

A block is erased when none of its pages can be written again. For non-rewriting codes, each page is written only once before erasure. For rewriting codes, we consider a page to be unwritable after the first time it cannot be rewritten (i.e., because at least one cell that needs to be rewritten is already saturated).

When the block is about to be erased, we determine the percentage of cells that are programmed at each level. As this process is repeated for many Flash block erasures, we statistically calculate the cell program probability $\boldsymbol{P}$. For rewriting codes, we also calculate the average number of rewrites per Flash page ($PageRewrites$) before erasure.

For input datawords, we consider three possibilities: random data, real data that have been compressed, and real data without compression. For real data, we use the same input data set as prior work [52]. Note that random data is equivalent to real data that has been through the utmost possible compression.

The compared coding schemes are:

- MWC(L, n, m) that has a rate of $\frac{m}{log(L)*\mathrm{n}}$.
- WOM(3,2), a WOM code [79] with rate 2/3.
- MFC-1/2, an MFC that was reported to have the best LG and AG among all MFCs [55] with a rate of 1/2.
- DSC [52] with datawords of length m. We denote such a code as DSC(m) and it has a rate of $\frac{4KB}{4KB+2^m\times m}$ where 4KB is the size of a Flash page and $2^m \times m$ is the extra capacity in bits required to store the dynamic mapping.

- FnW [20] with datawords of length m. We denote such a code as Fnw(m) and it has a rate of $m/(m+1)$

3.6 Evaluation

We now evaluate and present results on MWC. First, we show some general results for various MWCs and investigate how sensitive MWCs are to different W^{prof}. We then focus on a specific, real-world MLC technology with a real-world W^{prof} [52]. For this specific W^{prof}, we report results both for MWCs as well as prior work, focusing on random data. Next, we repeat the same experiments for real data, both when compressed and uncompressed.

We emphasize that for a given dataword input size m, an MWC can have different codeword sizes n that provide different AGs. For example, if we assume an MWC(4, n, 3) for $\boldsymbol{W^{prof}} = [0, 0.33, 0.66, 1] \times {W^{prof}}_3$ we can get different AGs for different values of n as seen in Figure 3.2. We observe that the AG is maximized for n=7. Although in this example there are larger values of n that have the same AG, they are not preferable to n=7. This is

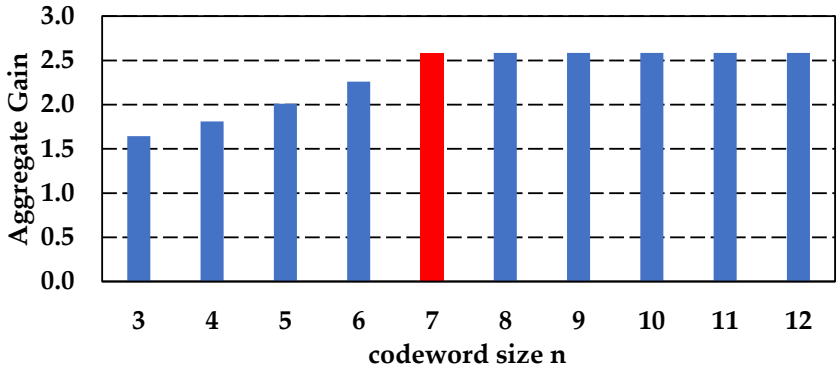


Figure 3.2: The AG of MWC(4, n, 3) maximizes for n=7.

because higher values of n result in lower code rates (but higher LG and thus overall the same AG). For all future results, we always choose the minimum value of n that maximizes AG for a given MWC(L, n, m) implementation. Additionally, we focus our evaluation on MLC Flash technology (i.e., L=4) as is currently the most popular type of Flash memory. We emphasize however that MWC could work for any Flash technology; SLC (L=2), MLC (L=4), TLC (L=8) (or any other arbitrary value of L).

3.6.1 MinWear Codes for Various W^{prof}

We start by exploring the sensitivity of MWC to different $\boldsymbol{W^{prof}} = \boldsymbol{a} \times W^{prof}{}_{L-1}$ assuming random data. We consider four different W^{prof}. The first profile is from a real-world MLC device that was extracted from prior work [52] where $\boldsymbol{a} = [0, 0.41, 0.65, 1]$. We also consider the case of linear increase in wear with respect to the level of the cells, $\boldsymbol{a} = [0, 0.33, 0.66, 1]$. Additionally, we consider the case where Levels 1-3 cause the same amount of wear, $\boldsymbol{a} = [0,1,1,1]$. Finally, we consider the case in which all levels induce the same amount of wear $\boldsymbol{a} = [1, 1,1,1]$. Although the latter profile is unrealistic it is the case that has been implicitly assumed by rewriting codes [55, 36, 82, 54, 9] and we include it for comparison.

Figure 3.3 presents the AG of MWC(4, n, m) for the different values of $\boldsymbol{a}$. For each profile $\boldsymbol{a}$, we include a variety of (n, m) pair values. We observe that the AG of MWC(4, n, m) for a given $\boldsymbol{a}$, increases linearly with m. The only exception is when all levels (including Level 0) cause equal wear (i.e., $\boldsymbol{a} = [1, 1,1,1]$), in which case MWCs are inefficient. That

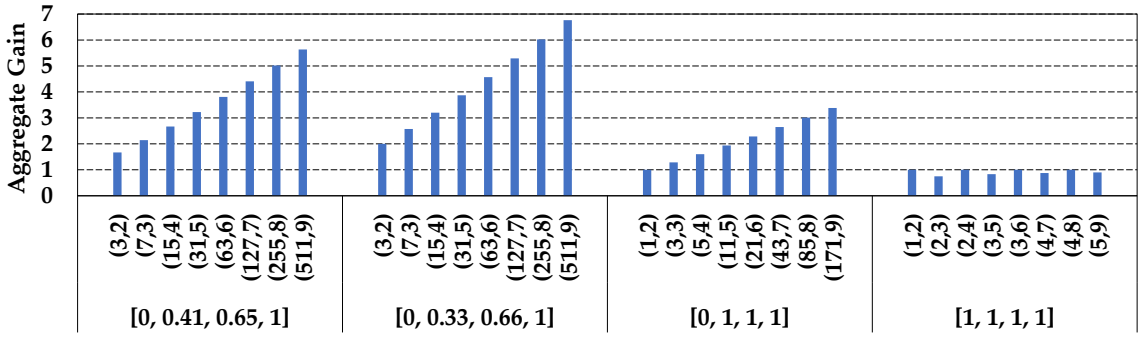


Figure 3.3: The *AG* of MWC(4, *n*, *m*) for different values of (n, m) and different W^{prof}.

indicates that in such a scenario, rewriting codes would be more preferable than bias codes.

In Figure 3.4 we present the rate and lifetime gains of MWC(4, *n*, *m*) when $\boldsymbol{a} = [0, 0.41, 0.65, 1]$. From Figure 3.4 we observe that rates are relatively low and as *m* increases, they asymptotically reach 0. However, *LG* increases exponentially (causing the overall *AG* to linearly increase as we presented in Figure 3.3) and can theoretically keep increasing without bound as *m* increases. However, an extremely large *LG* is not practically needed. Thus, for the rest of this work, we mainly focus on MWC(4, *3*, *2*), which provides the highest rate=0.33, *LG*=5, and *AG*=1.67. We will also focus on MWC(4, *63*, *6*) as an extreme case, with rate=0.05, *LG*=80, and *AG*=3.82.

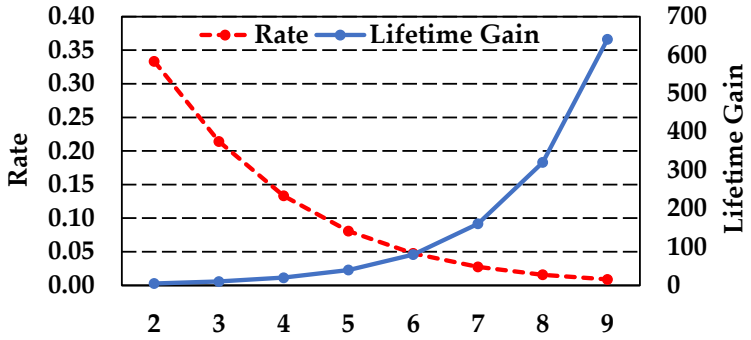


Figure 3.4: Rate and *LG* of MWC(2, *n*, *m*) for different *m* with $\boldsymbol{a} = [0, 0.41, 0.65, 1]$.

3.6.2 Comparing Codes – Random Data

We now compare MWC with prior work. We consider a real MLC device with $\boldsymbol{a} = [0, 0.41, 0.65, 1]$ as reported in [52]. We start by considering random data written in the Flash memory. In Figure 3.5, we present the best *AG* each code can achieve. For FnW and DSC codes, we run our simulations for different dataword sizes *m* and simply select the dataword size that achieves the best *AG*. WOM(3,2) and MFC-1/2 codes have a fixed dataword size. As we see from Figure 3.5, the simplest MWC, MWC(4,3,2), achieves the same *AG* as the state-of-the-art rewriting code, MFC-1/2. However, as we showed in Chapter 3.6.1, MWC(4, *n*, *m*) can provide arbitrarily large *AG* as *m* increases. For example, MWC(4,63,6) has an *AG* of 3.82 that outperforms all other codes.

This comparison to prior work highlights the importance of the precise metrics we introduced in this paper. Typically, prior work on rewriting codes does not account for different wear when erasing cells from different cell levels, and they could overestimate the *AG* of different implementations. MFC-1/2, which rewrites a Flash page 12 times on average, with a rate of 1/6, would be estimated to have an *AG* of 2. However, its *AG* is

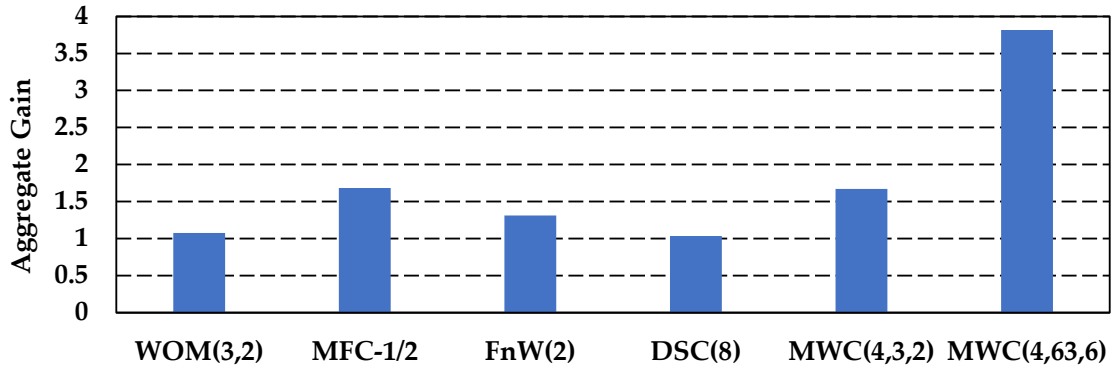


Figure 3.5: Aggregate gains for different implementations.

only 1.67 when accounting for the variable wear caused by erasing cells at different levels. Similar is the case for WOM(3,2) that can rewrite a Flash page twice on average with a rate of 2/3 and typically would be estimated to have an *AG* of 1.33. However, because WOM(3,2) significantly increases the probability of programming cells in higher levels its *AG* is reduced to 1.08. Both FnW and DSC try to exploit the presence of repeated bit patterns to increase *LG*. However, under random data, repeated bit patterns are rare and thus their *AGs* are minimal.

Although the *AG* results show that MWC is the most cost-efficient way to increase *LG*, we also want to investigate how different codes perform for a given target *LG*. Assume we want to design a Flash memory with target *LG*=*X* and host-visible capacity *HC*=*Y*. We want to know the total amount of raw Flash capacity (host-visible capacity + redundant capacity) the Flash memory system must have to achieve *LG*=*X* and *HC*=*Y*. Figure 3.6 presents the required raw capacity (smaller is better) for different target *LG*s. We present results for all codes, as well as naïve redundancy. For each target *LG*, we normalize our results to the required raw capacity needed by naïve redundancy.

As we observe from Figure 3.6, MWC always achieves the most efficient codes in terms of required raw capacity. Furthermore, as the target *LG* increases, the efficiency of MWC increases, while the rest of the codes stay constant. The only exception is MFC-1/2 for *LG*=5, where its capacity cost is actually higher than naïve redundancy. The reason

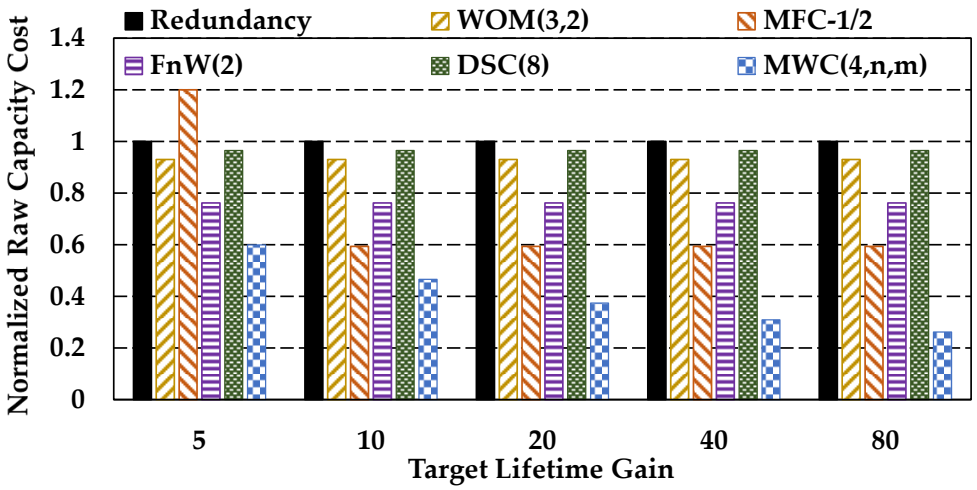


Figure 3.6: Capacity cost for a fixed target lifetime gain and host-visible capacity. Results are normalized per target lifetime gain for the uncoded baseline implementation.

behind this counter-intuitive result is that MFC-1/2 has a minimum *LG*=10 and thus it is inefficient for lower *LG*s.

MWC(4, n, m) is flexible and can provide a variety of tradeoffs. For example, when *LG*=5, MWC(4,3,2) only requires 60% of the raw capacity needed by naïve redundancy. For *LG*=10, the cost is reduced to 46% (MWC(4,7,3)) and goes as low as 26% (MWC(4,63,6)) for *LG*=80. The cost will always keep reducing, asymptotically approaching zero, as we consider larger *LG*s.

3.6.3 Comparing Codes – Real Data

We also compare the different implementations for real data, both when compressed and uncompressed. We use the same input data set as in [52]. To understand the impact of real data we first write them in the uncoded baseline and record the

percentage of cells at different levels, as seen in Figure 3.7. We repeat this procedure both when the data are uncompressed and compressed. Because the uncompressed input data are relatively sparse, most of the cells (~37%) are programmed in Level 0. However, after compressing the input data, it closely resembles random data and as a result, we have roughly the same number of cells (~25%) programmed at each level.

For this specific input data, an average compression of 60.5% is achieved using a simple and widely available compression program [1]. Thus, we can think of compression as a code with rate 2.53 (achieving 60.5% savings in capacity) that increases the probability of cells being programmed at higher levels. Thus, we can now use our precise metrics and evaluate compression, similar to all other codes, and calculate its *LG* and *AG*. This showcases the importance of our defined metrics as they can predict the effect of any coding scheme in terms of *LG* and *AG*, even if the coding scheme in question does not target Flash endurance.

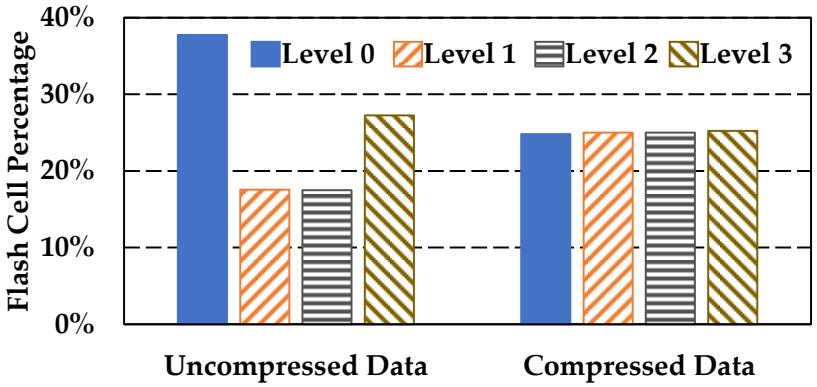


Figure 3.7: Percentage of cells programmed at different levels for real, uncompressed and compressed data.

We now compare all codes under the real data both when compressed and uncompressed and calculate their *AG*. Our results in Figure 3.8 are normalized to the baseline when using uncompressed data. When using compression, the *LG* of the uncoded baseline is reduced (*LG*=0.90), because cells are programmed to higher levels and thus cause more wear during erasure. However, the amount of capacity savings outweighs the loss in *LG*, causing the *AG* to increase to 2.24. In other words, if we use the capacity savings of compression as redundant capacity to increase lifetime, we could achieve an *LG*=2.24. This observation is consistent for all the codes, showing that when combined with compression they can further increase their *AG*s.

From Figure 3.8 we also observe that MWC(4, *n*, *m*) achieves the biggest boost in *AG* when combined with compression. This is because it does not depend on repeated data patterns to reduce wear. On the contrary, MWC(4, *n*, *m*) best reduces wear when data are random. More specifically, when MWC(4, 3, 2) is combined with compression, *AG* is

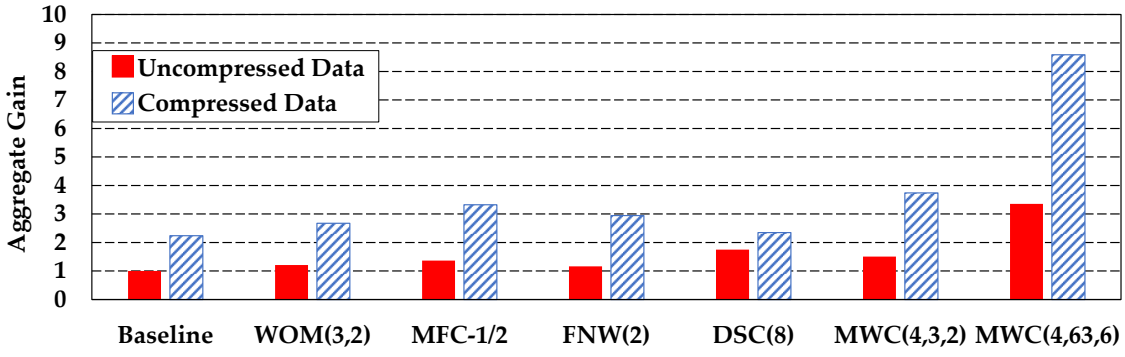


Figure 3.8: Aggregate gain for different implementation under real, uncompressed and compressed data. Results are normalized to the uncoded baseline under uncompressed data.

increased to 3.7 (from 1.5), outperforming all prior work. The overall rate of MWC(4,3,2) when combined with compression increases to 0.83 (from 0.33). MWC(4, 63, 6) increases its *AG* to a remarkable 8.59 with a rate of 0.13.

We should mention that, when the data are uncompressed, DSC(8) can achieve better *AG* than MWC(4, 3, 2), because it can exploit the existence of repeated bit patterns through its dynamic construction of dataword-to-codeword mapping. However, in data centers (where Flash lifetime is more important than in personal computers), sophisticated compression and deduplication techniques are used to reduce the capacity footprint of data. Thus, our results showcase that MWC is the best coding scheme to achieve high *LG* for the least cost (i.e., to maximize *AG*) in such environments.

3.7 Conclusion on MinWear Coding

We have presented a new bias coding scheme for Flash memory. MinWear Coding is based on the critical observation that not all Flash erases cause equal wear and that the ideal optimization objective is to minimize wear (rather than to maximize the number of rewrites per erase). MWC lends itself to simple encoding and decoding, and the experimental results show that it achieves superior aggregate gains that can be increased arbitrarily. To fairly compare MWC to prior coding schemes, we developed a set of precise metrics that eliminate confusion and mischaracterizations.

4. Tolerating Shift Errors in Racetrack Memory

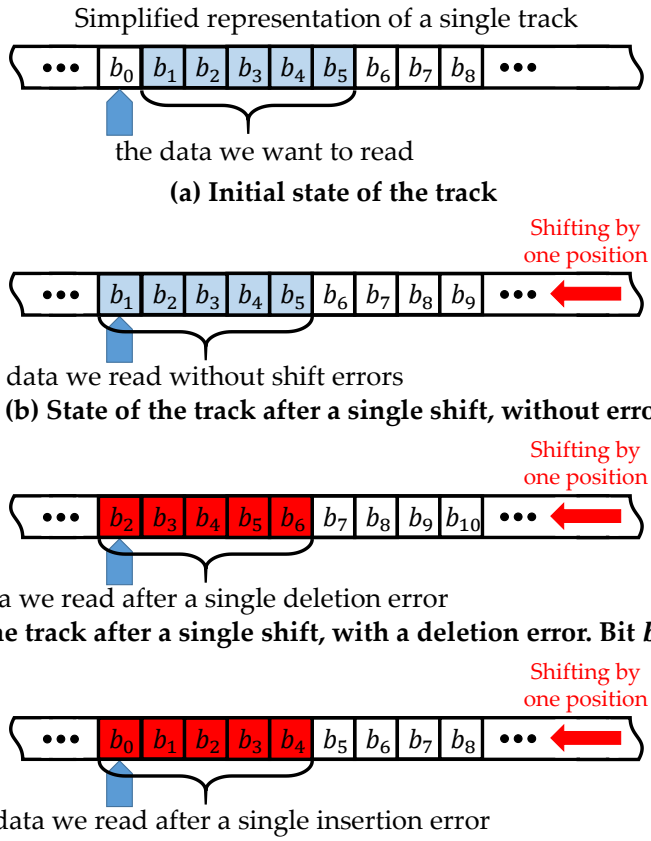
In this chapter, we discuss shift errors in racetrack memory and propose a new solution called GreenFlag, that detects and corrects shift errors in 3D racetrack memory. We start by explaining shift errors extensively in Chapter 4.1. In Chapter 4.2 we discuss why traditional ECCs, that were originally designed to tolerate bit flips, cannot effectively tolerate shift errors. We then present prior work on shift errors in Chapter 4.3. We present GreenFlag in Chapter 4.4. In Chapter 4.5 we present the methodology we followed to evaluate GreenFlag and finally, we present our results in Chapter 4.6.

The contributions of this chapter are summarized here:

- We present the GreenFlag coding scheme that combines a novel construction of Varshamov-Tenengolts codes with specially crafted delimiter bits to efficiently detect and correct shift errors.
- We implement and evaluate the GreenFlag hardware circuitry required to encode datawords and decode codewords.
- We present the first analysis of the viability of racetrack memory—based on the trade-off between code length, code rate, latency, and bandwidth—at each level of the memory hierarchy.

4.1 Shift Errors in Racetrack memory

To explain shift errors, we use an example in which several bits are stored on a racetrack as in Figure 4.1(a). To read (write) the stored bits, we need to perform a sequence



(a) Initial state of the track

(b) State of the track after a single shift, without errors

(c) State of the track after a single shift, with a deletion error. Bit b_1 was deleted

(d) State of the track after a single shift, with an insertion error. Bit b_0 was inserted

Figure 4.1: Accessing bits in the absence and presence of shift errors.

of consecutive shift and read (write) operations. A shift operation, as shown in Figure 4.1(b), is performed to place the next bit in racetrack memory under the read/write port.

Suppose we want to read bit b_1 and currently bit b_0 is positioned under the read/write port. Ideally, we inject the correct amount of current to perform a single shift operation and b_1 is placed under the read/write port as in Figure 4.1(b). If the injected current is larger than expected, then we skip the domain that contains b_1, and b_2 is wrongly positioned under the read/write port as in Figure 4.1(c). We refer to this error as a *deletion* error since b_1 is skipped (i.e., a bit is deleted). On the other hand, if the injected

current is smaller than expected, the port's position does not change and b_0 remains under the read/write port as in Figure 4.1(d). We refer to this error as an *insertion* error (i.e., an extra bit is inserted).

We emphasize that, regardless of whether a shift error occurs, the memory controller always provides the desired number of bits, say n. A shift error just affects which n bits are provided. For instance, if a deletion happens as in Figure 4.1(c), then we receive the first $n = 5$ bits (excluding the deleted bit b_1), which are b_2, b_3, b_4, b_5, b_6. We observe that a single deletion (or insertion) can cause as many as n bit errors, which greatly complicates error coding.

4.2 Traditional ECCs in Racetrack Memory

Coding for racetrack is different than for typical memory technologies because of its bit-serial nature (which affects performance) and its susceptibility to shift errors (which makes many standard codes ineffective). As we explained in Chapter 4.1 even a single shift error can cause multiple bit errors.

To explain the challenges of coding for racetrack, we use the well-known Hamming code as a running example. Hamming codes add parity bits to datawords to form codewords, and they are parameterized by the bit lengths of the datawords and codewords. With no loss of generality, we assume a Hamming(8,4) code that encodes 4-bit datawords as 8-bit codewords. The code provides SECDED protection.

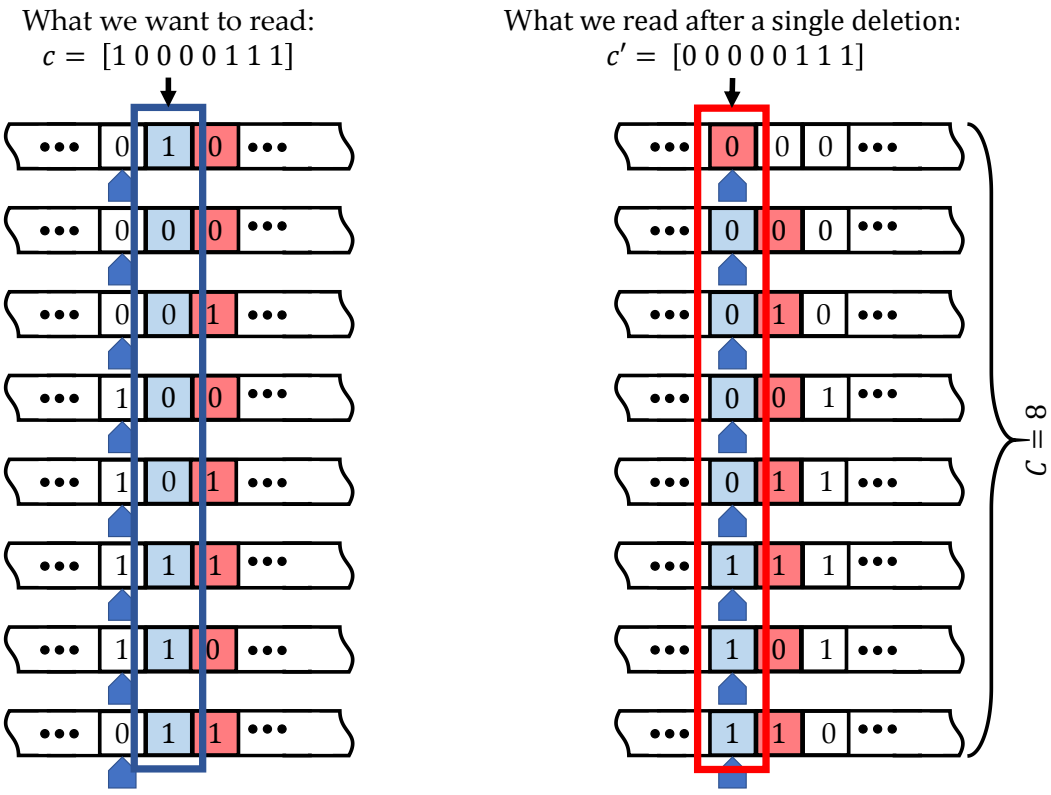
We now discuss two broad approaches to coding for racetrack—vertical (across tracks) and horizontal (within a track)—and show why commonly used coding techniques like Hamming(8,4) do not work.

4.2.1 Vertical Coding

Ideally, for performance, we would like to employ a vertical code that uses domains across multiple tracks to store a codeword, as illustrated in Figure 4.2. We would use one bit from each of the C different tracks to store the C bits of the codeword. With such a code, we could read all C bits in parallel with a single shift and read operation on each track.

In Figure 4.2(a) we illustrate an 8-bit codeword c =10000111 from the Hamming (8,4) code striped vertically across C = 8 tracks, that we attempt to access. Figure 4.2(b) shows the state of these 8 tracks after a single deletion in the first track, that changes c to c'= 00000111.

Only one bit position is affected and thus a SECDED Hamming code like Hamming(8,4) can correctly recover c. However, the code cannot fix the still-erroneous position of the domains on that track with respect to the read/write port. Note that, because the code cannot differentiate between deletions and insertions, the codeword bits that are striped across tracks cannot be correctly aligned. Once the position of the domains becomes incorrect (misaligned) with respect to the port, it remains incorrect, which can lead to further errors on that track. Worse, a subsequent error on another track can now



(a) Initial state of C tracks, before attempting to read c

(b) The state of C tracks after a single shift operation in each track. The first track had a deletion error.

Figure 4.2: Example of using Hamming(8,4) to attempt vertical coding in racetrack.

lead to a multiple-error scenario that cannot be corrected by a SECDED code. Thus, a shift error on one track can easily lead to a situation in which the data on all 8 tracks are lost.

We are unaware of any existing coding scheme that would overcome the problem we have just described. We do not claim that such a code is impossible, but to our knowledge, it has not yet been invented.

4.2.2 Horizontal Coding

Given that vertical coding is not possible, we are left with horizontal coding, in which each C-bit codeword is written on the domains of a single track, as illustrated in

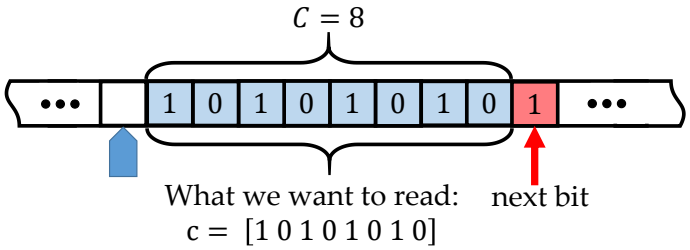
Figure 4.3. The main drawback of horizontal coding is that we need to perform C read and C shift operations to read all the domains (bits) of a single codeword. That means that latency depends on C.

To explain why standard codes do not suffice for handling shift errors, we use an example. Our Hamming(8,4) code has a generate matrix G shown below.

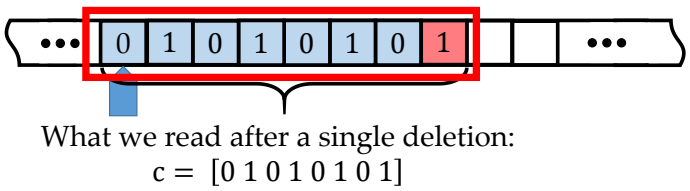
$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Assume that we want to store the dataword $d = [1\ 0\ 1\ 0]$ on a track. The dataword is first encoded to the codeword c and then written to the track as shown in Figure 4.3(a).

$$c = d * G = [1\ 0\ 1\ 0\ 1\ 0\ 1\ 0]$$



(a) Initial state of a track with C bits before attempting to read c



(b) State of the track after a single shift operation that resulted to a deletion error

Figure 4.3: Example of using Hamming(8,4) to attempt horizontal coding in racetrack.

Assume that a single deletion happens during the first shift operation while we are trying to read c. In Figure 4.3(b) we present the state of the track after the deletion error converts c into c'.

$$c' = [\,0\;1\;0\;1\;0\;1\;0\;1]$$

We observe that c' differs from c in every bit position, which overwhelms the capability of a SECDED code. Nevertheless, it is instructive to see the resulting dataword d' that is produced by our code:

$$c' = d' * G = [0\;1\;0\;1\;0\;1\;0\;1\,]$$

$$d' = [0\;1\;0\;1]$$

We observe that d' differs from the correct dataword d in every bit position, despite experiencing only a single shift error. Through this example, it is easy to see that even if we had used stronger codes (e.g., DECTED), we still could not address shift errors. Additionally, even if we could detect and correct the bit-value errors, the position of the read/write port with respect to the domains would still be wrong. Thus, any future read/shift operations would suffer additional bit-value errors.

4.2.3 Racetrack Error Model

Now that we have discussed shift errors, we present the racetrack error model that we will assume throughout this work. Our racetrack error model includes single and

double shift errors[1]. That is, every shift operation has some probability to result in a single or double shift error. We do not consider triple or higher shift errors as prior work [85] has shown them to have a negligible probability. However, multiple single shift errors (though multiple erroneous shift operations) may have the same effect as a triple or higher shift error.

Similar to prior work [85, 17, 58], our error model does not include bit-flip errors but only considers shift errors. Although tolerating both shift errors and bit flips is an interesting and challenging problem, it is outside the scope of this work.

4.3 Prior Work

While there is a large body of work on racetrack memory itself and how to apply it to various levels of the memory and storage hierarchy [76, 75, 84, 72, 56], we focus here on fault tolerance. We divide prior work into two sections, based on whether it targets 2D or 3D racetrack; all of the schemes for 2D racetrack rely on having multiple read/write ports, which is not feasible for 3D racetrack. All of the prior works use horizontal coding implementations. Our proposed solution GreenFlag, that we present in detail in Chapter 4.4, is also a horizontal code that uses only a single read/write port making it a feasible solution for 3D racetrack.

---

[1]A single shift error occurs when a single shift operation deletes or inserts a single bit. A double shift error occurs when a *single* shift operation deletes or inserts two bits.

4.3.1 Fault Tolerance for 2D Racetrack

Sima et al. [17] propose a scheme where data are encoded using run-length limited (RLL) codes. RLL codes impose a minimum and a maximum number of zeros between consecutive ones in a codeword. When a codeword is shifted through a read/write port the number of zeros between ones is recorded. By shifting the same data through multiple ports that have pre-determined distances between them, shift errors can be detected (and corrected) as a mismatch between the number of zeros recorded by different read/write ports. Depending on the number of ports available and the RLL code used, different fault tolerance levels can be achieved.

HiFi [85] adds predetermined bit patterns (a.k.a., delimiters) to the edges of each track. Data bits are stored in the middle section of a track. At runtime, data bits and delimiter bits are read simultaneously via multiple ports. If the bit values of the delimiter bits differ from the expected values, a shift error is detected. HiFi can detect double shift errors and correct single shift errors. As bits are shifted during consecutive reads, some of the delimiter bits in the edge of the track are "shifted out" of the track and virtually get deleted. Thus, for every shift in racetrack, two delimiter bits are read from one edge (to detect possible shift errors), while at the same time a new delimiter bit is written in the other edge of the racetrack (to replace the deleted delimiter bit). A subsequent extension of HiFi, called np-ECC[77], additionally provides single bit-flip detection for the delimiter bits (but not the data).

4.3.2 Fault Tolerance for 3D Racetrack

To the best of our knowledge, there is only one other work that can tolerate shift errors in racetrack memory, using a single read/write port.

Ollivier et al. [58] present a shift protection scheme that combines multiple coding and physical techniques. They propose a new physical way to access information on the racetrack, called Traverse Read (TR). With TR, one can calculate the number of 1s in a track without performing shift operations. Additionally, delimiter bits are added to the edges of each track. Using TR, the weight (number of 1s) of the data and the delimiter bits can be calculated. This weight is called the signature of a track. Based on the current signature of a track, the signature after a shift operation can be predicted. If a mismatch occurs between the predicted value of the signature and the signature calculated after a shift operation, a shift error is detected. To guarantee that the current signature of a track can always be reliably recovered, they store it in STT-RAM, a memory technology that does not incur shift errors.

Although a clever idea, this scheme has significant drawbacks. First, it can only tolerate single shift errors (cannot detect double shift errors). Furthermore, only one shift error in an entire track can be tolerated, limiting the practical length of each track. On the contrary, GreenFlag splits a track to multiple codewords tolerating up to two single shift errors or one double shift error in each codeword.

4.4 GreenFlag Coding

In this section, we present GreenFlag coding. GreenFlag is based on Varshamov-Tenengolts (VT) codes [74], a graph-based coding technique that chooses codewords such that no one codeword can be confused with any other codeword after a single deletion or insertion error. VT codes achieve that by selecting codewords that satisfy the constraint:

$$\sum_{i=1}^{n} ic_i \quad \text{mod } n+1 \quad \equiv \quad 0$$

We refer to this constraint as the checksum. When a shift error occurs in a VT codeword, the checksum is no longer equal to zero and thus a shift error can be detected. However, the nature of the shift error (deletion or insertion) is unknown.

GreenFlag coding is created by integrating a VT-based horizontal code with specially crafted delimiter bits that overall allow us to detect and correct not only single but also double shift errors. We first explain how we construct VT codes for GreenFlag and how we encode and decode in the error-free case. Then we introduce different delimiter options to help us categorize detected shift errors as deletions or insertions. Finally, we describe how GreenFlag corrects shift errors by both fixing erroneous bits and realigning the racetrack domains with respect to the read/write port.

4.4.1 GreenFlag Construction: A Novel VT Construction

With an eye on implementation, we introduce a novel construction algorithm for Varshamov-Tenengolts codes. We can construct VT(n) codes for any value of n; however,

to provide fast and efficient encoding and decoding, we use powers of two, i.e., $n = 2^l$. Our encoding and decoding algorithms are based on the mathematics of VT codes, and we have proven them correct. These proofs are presented in Appendix A.

4.4.1.1 Encoding

Suppose $n = 2^l$ and $k = n - l - 1$. We encode k data bits, $d_1, d_2, \ldots, d_k$, into a codeword c of length n such that it satisfies the VT checksum. Such a code is referred to as GreenFlag$(n, k, delimiter_size)$.

We present the encoding algorithm and an example for $n = 8$ in Table 4.1. In this example, $l = 3$ and we encode $k = n - l - 1 = 4$ data bits (1011) into a codeword of length $n = 8$. The encoding algorithm creates $l + 1$ check bits to ensure that the checksum holds. We index bit positions in a codeword from 1 to n (rather than 0 to $n - 1$). The rate of this code is given by,

$$R(n) = \frac{n - log_2(n) - 1}{n}$$

which approaches 1 in the limit of large n.

4.4.1.2 Error-Free Decoding

To recover a dataword, we first have to read the codeword and recalculate the checksum. If the checksum is zero then no shift errors have occurred. Because of our novel construction, we can then simply recover our error-free dataword by extracting all the bits that are not powers of two. Thus, this allows for efficient decoding in the common, error-free case.

Table 4.1. Encoding algorithm with an example.

Encoding Algorithm	Example for $n = 8, l = 3$
Step 1: Start with a zero-vector c of length $n = 2^l$	$\boldsymbol{c} = 0\,0\,0\,0\,0\,0\,0\,0$
Step 2: Set positions that are not powers of two to data bits	$\boldsymbol{d} = 1011$ $\boldsymbol{c} = 0\,0\,\underline{1}\,0\,\underline{0}\,\underline{1}\,\underline{1}\,0$
Step 3: Calculate the checksum $\sum_{i=1}^{n} ic_i$	$\sum_{i=1}^{8} ic_i = 3 + 6 + 7 = 16$
Step 4: Set s to be the minimum value that needs to be added to the checksum to make it equal to 0 modulo $n + 1$	$s = 9 - mod(16,9) = 2$
Step 5: Set the $l + 1$ positions that are powers of two to the binary expansion of s. Start from c_1 and set it to be the MSB. Move all the way to c_n and set it to the LSB	$s = 2 = (0010)_2$ $c_1 = 0, c_2 = 1$ $c_3 = 0, c_4 = 0$ $\boldsymbol{c} = \underline{0}\,\underline{1}\,1\,\underline{0}\,0\,1\,1\,\underline{0}$

In the case that the checksum is non-zero we know that a shift error has occurred. However, to correct it we must first decide if it was a deletion or an insertion. We now discuss how we use delimiters to determine the nature of shift errors.

4.4.2 Greenflag's Delimiters: Categorizing Shift Errors

We use specially crafted delimiters to identify the type of shift error and distinguish insertions from deletions. Intuitively, the delimiters help us to separate codewords from each other in the presence of shift errors. We refer to a codeword that is concatenated with delimiter bits as an *extended codeword*.

We choose delimiters to most efficiently achieve our goals, and tolerating greater numbers of shift errors requires longer delimiters. All delimiters must be functional when read from right to left or left to right because domains on a track can be shifted and read

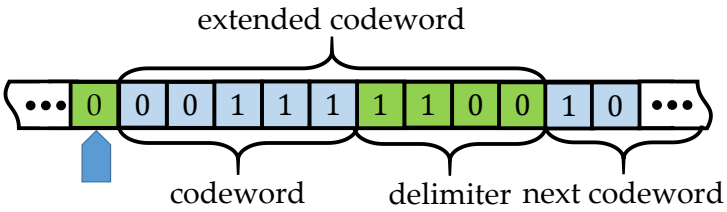


Figure 4.4: An example of constracting an extended codeword by concutinating a codeword with a delimiter.

in both directions. In other words, delimiters should either be palindromic or have their less significant bits equal to the complement of the corresponding more significant bits.

We illustrate an example of a racetrack storing extended codewords in Figure 4.4. The example has 5-bit VT codewords and 4-bit delimiters with values 1100. In the absence of shift errors, when we read an extended codeword, its last four bits will be the known bits 1100 of the delimiter.

Choosing the values of the delimiter bits can be tricky, because of edge cases like a shift error in the delimiter bits themselves. Our crafted delimiters, with the help of the VT codes, can help us detect and correct shift errors even in such corner cases, and we exhaustively tested our scheme against every possible combination of shift errors in data bits and delimiter bits (see Appendix A). We now present two delimiter options, with different lengths, to show how the choice of delimiter impacts the ability to identify shift errors.

Delimiter-1 is the 4-bit pattern $d = (d_1, d_2, d_3, d_4) = 1100$. We use delimiter-1 along with the VT checksum to categorize shift errors as shown in Table 4.2. Note that in Table 4.2 we do not include the observation of d_4, because any error that affects the last

delimiter bit will be detected when we read the next extended codeword. Because delimiter-1 can detect some double shift errors (1 insertion plus 1 deletion) but not all (double deletion or double insertion), we classify a GreenFlag(*n,k,4*) code as a SECSED code for shift errors.

Delimiter-2 is a 6-bit delimiter with the pattern 111000. Table 4.3 shows how to use the VT checksum and delimiter-2 to categorize shift errors. Unlike delimiter-1, delimiter-2 can detect and differentiate double deletions and double insertions. Although VT codes can only recover codewords from a single shift error, delimiter-2 (with the help of the VT checksum) enables GreenFlag to correct the relative position of the read/write port for up to double shift errors. Thus, double shift errors can now be detected and corrected by

Table 4.2. How delimiter 1100 identifies errors (X = 0 or 1).

Observed delimiter bits			**Checksum**	**Decisions**
d_1	d_2	d_3		
1	1	0	0	**no error**
1	1	0	≠ 0	**1 insertion and 1 deletion**
1	0	0	≠ 0	**1 deletion**
X	1	1	≠ 0	**1 insertion**

Table 4.3. How delimiter 111000 identifies shift errors (X = 0 or 1).

Observed delimiter bits					**Checksum**	**Decision**
d_1	d_2	d_3	d_4	d_5		
1	1	1	0	0	0	**no error**
1	1	1	0	0	≠ 0	**1 deletion and 1 insertion**
1	1	0	0	0	X	**1 deletion**
X	1	1	1	0	X	**1 insertion**
1	0	0	0	X	X	**2 deletions**
X	X	1	1	1	X	**2 insertions**

rereading (read-replay) the extended codeword in the opposite direction. Note that VT codes enable us to correct single shift errors without the need for a read-replay and thus single shift error correction will be substantially faster than correcting double shift errors (requires read-replay). Overall a GreenFlag(n,k,6) code is a DECDED code for shift errors.

Note that GreenFlag combines delimiters and the VT checksum to categorize shift errors. Delimiters alone do not suffice to detect nor to determine the type of shift errors.

4.4.3 Decoding in Presence of Shift Errors

If a shift error is detected, we apply additional shift operations so that we correctly realign the domains with the read/write port. If it was a double shift error, we need to reread the entire extended codeword. In the case of a single shift error, we correct the erroneous bit values as explained next.

4.4.3.1 Correcting a Single Deletion Error

Correcting a single deletion error can be accomplished using the elegant algorithm proposed by Levenshtein [50] and which we present in Table 4.4 alongside an example for n=8. The main idea is that we want to find a position in the codeword where if we insert a bit, the checksum will become equal to zero. Using this decoding algorithm, we reconstruct the correct codeword. Then we recover the dataword as discussed for the error free case. It is worth mentioning that while the decoding algorithm corrects the codeword, it does not reveal the exact position where the deletion occurred. In fact, in the example

Table 4.4. Deletion correction with an example.

Deletion Correction Algorithm	Example for $\boldsymbol{n = 8, l = 3}$
Step 1: Suppose a VT codeword $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$ is stored in racetrack memory. Using GreenFlag we have detected a single deletion and thus we observe $\boldsymbol{c'} = (c'_1, c'_2, \dots, c'_{n-1})$.	$\boldsymbol{c} = 0\ 1\ 1\ 0\ 0\ 1\ \underline{1}\ 0 \rightarrow \boldsymbol{c'} = 0\ 1\ 1\ 0\ 0\ 1\ 0$
Step 2: Set ω to be the Hamming weight (number of 1's) of c'.	$\omega = 3$
Step 3: Calculate the checksum $\sum_{i=1}^{n} ic'_i$	$\sum_{i=1}^{8} ic'_i = 2 + 3 + 6 = 11$
Step 4: Set s to be the minimum amount that needs to be added to the checksum to make it 0 modulo $n + 1$	$s = 7$
Step 5: If $s \leq \omega$, we restore the codeword by adding a 0 immediately to the left of the rightmost s 1's. Otherwise, we restore the codeword by adding a 1 immediately to the right of the leftmost $(s - \omega - 1)$ 0's.	$s > \omega \rightarrow$ restore the codeword by adding a 1 immediately to the right of the leftmost $(s - \omega - 1) = 3$ zeros to arrive at the correct codeword $\boldsymbol{c}$. $\boldsymbol{c} = 0\ 1\ 1\ 0\ 0\ \underline{1}\ 1\ 0$

in Table 4.4, if either of the 1's in positions 6 or 7 was deleted, the algorithm would have behaved in exactly the same way.

4.4.3.2 Correcting a Single Insertion Error

Handling an insertion is very similar to handling a deletion and follows the same logic. We modify the deletion correction algorithm to handle an insertion as given in Table 4.5. Again, it is worth mentioning that while the algorithm corrects the codeword, it does not reveal the exact position where the insertion occurred.

Table 4.5. Insertion correction with an example.

Insertion Correction Algorithm	Example for $\boldsymbol{n = 8, l = 3}$
Step 1: Suppose a VT codeword $\boldsymbol{c} = (c_1, c_2, \ldots, c_n)$ is stored in racetrack memory. Using GreenFlag we have detected a single insertion and thus we observe $\boldsymbol{c'} = (c'_1, c'_2, \ldots, c'_{n+1})$.	$\boldsymbol{c} = 0\,1\,1\,\underline{0}\,0\,1\,1\,0 \rightarrow \boldsymbol{c'} = 0\,1\,1\,0\,0\,0\,1\,1\,0$
Step 2: Set ω to be the Hamming weight (number of 1's) of c'.	$\omega = 4$
Step 3: Calculate the checksum $\sum_{i=1}^{n} ic'_i$	$\sum_{i=1}^{8} ic'_i = 2 + 3 + 7 + 8 = 20$
Step 4: Set s to be the minimum amount that needs to be deducted from the checksum in order to make it 0 modulo $n + 1$	$s = 2$
Step 5: If $s \leq \omega$, we restore the codeword by removing a bit immediately to the right of the leftmost $\omega - s$ 1's. Otherwise, we restore the codeword by removing a bit immediately to the right of the leftmost $(s - \omega)$ 0's.	$s \leq \omega \rightarrow$ restore the codeword by removing a 0 immediately to the right of the leftmost $s = 2$ ones to arrive at the correct codeword $\boldsymbol{c}$. $\boldsymbol{c} = 0\,1\,1\,\cancel{0}\,0\,1\,1\,0.$

4.4.4 Block Organization

In order to minimize the access latency of a block, GreenFlag uses multiple tracks to store a single block similar to prior work [85, 76]. Assuming a 512-bit block, prior work uses 512 tracks to stripe a block at a bit granularity. GreenFlag stripes a block at an extended codeword granularity. For example, GreenFlag(8,4,6) uses 128 tracks that each store an (8+6)-bit-long extended codeword. All 128 extended codewords can be read in parallel to retrieve a single block.

4.5 Methodology

In this section, we discuss the methodology followed to evaluate GreenFlag both in terms of performance and fault tolerance. Because GreenFlag(*n,k,4*) only provides

SECSED shift error protection, while GreenFlag(n,k,6) can provide DECDED shift error protection with just two more delimiter bits, we focus our evaluation on the latter.

4.5.1 Performance Methodology

To understand the impact that GreenFlag has on racetrack memory performance we must understand both the inherent overheads of racetrack memory and the overheads of the GreenFlag encoding and decoding processes. In other words, the significance of the GreenFlag overheads depends on how they compare with the inherent overheads of the racetrack memory. For example, if the read latency in racetrack memory is orders of magnitude larger than the GreenFlag decoding latency, then GreenFlag would have insignificant performance overheads.

Thus, we choose to evaluate GreenFlag under the most strict scenario, assuming a best-case racetrack that has the minimum possible latency and energy for read, write, and shift operations. We use results from prior work on quantitative analysis for racetrack memory [84] to estimate these values and present them in Table 4.6.

To understand the overheads of GreenFlag, we design hardware encoder and decoder modules for different GreenFlag(n,k,6) codes, using the hardware description language Verilog. These modules implement the respective encoding and decoding algorithms discussed in Chapter 4.4. We then synthesize the hardware modules using Synopsys design compiler with the 15nm CMOS technology node and estimate the

latency, energy, and area overheads of these designs. In Table 4.7 we summarize the overheads for different GreenFlag(n,k,6) codes.

From Table 4.7 we can observe that the overheads tend to increase as n (codeword length) increases. However, we observe that the latency for error-free decoding is constant with respect to n. That happens because part of the decoding process is done in parallel with reading the codeword from the racetrack. We also observe that the rate of the codes (host-visible capacity) increases with respect to n. Thus we can already observe that larger values of n are desired to increase the host-visible capacity where lower values are desired to reduce the GreenFlag overheads.

We also study decoding performance in the presence of errors. For single shift errors, the latency depends on the position of the error. On average, the expected decoding latency is 8.75ns (the error-free latency) plus an additional $n/2$ cycles (at 0.8GHz) to find the position in the codeword a bit must be inserted or deleted to correct the shift error. For $n = 32$, that latency equals 28.75ns. For double shift errors, the latency is twice the error-free latency plus the latency of the read-replay. A read-replay has a latency of n shift and n read operations (i.e., 3.7ns for each shift and 2.1ns for each read, as shown in Table 4.6). For $n = 32$, the total latency is 203ns.

Lastly, we note that a designer could choose to use GreenFlag only for shift error detection (and not correction) as to reduce the decoding hardware complexity. When an

Table 4.6. Racetrack Characteristics.

Operation (per bit)	Latency (ns)	Energy (pJ)
Read	3.7	224
Write	10.2	998
Shift	2.1	124

Table 4.7. Hardware Implementation Overheads.

	Encoding					Decoding (error free)				
Codeword length *n* (bits)	4	8	16	32	64	4	8	16	32	64
Cycles	1	12	14	15	16	11	15	23	39	71
Frequency (GHz)	1	1	1	1	1	0.8	0.8	0.8	0.8	0.8
Latency (ns)	1	12	14	15	16	8.75	8.75	8.75	8.75	8.75
Power (uW)	10	204	311	755	974	212	286	413	675	812
Energy (pJ)	0.01	2.45	4.35	11.33	15.58	0.24	5.36	11.87	32.91	72.07
Area (um^2)	11	326	646	1117	2704	652	1014	1361	1754	2321
Rate (w/ 6-bit delimiter)	0.1	0.286	0.5	0.684	0.814	0.1	0.286	0.5	0.684	0.814

error is detected, we could then simply apply read-play to recover the correct codeword, regardless if it was a single or double shift error. Although that solution would save some area (~45%) and energy (~12%) for the decoding hardware, it would cause all shift error corrections (single and double) to incur the latency of double shift error correction. Because single shift errors are far more common [85], we do not consider this option during our evaluation.

In Chapter 4.6 we use both the inherent overheads of racetrack memory and the added overheads of GreenFlag to estimate the overall latency, energy, and bandwidth of our implementation.

4.5.2 Fault Tolerance Methodology

To evaluate the fault tolerance of GreenFlag we build a mathematical model that analytically calculates the probability of failure (P_f). We define failure as the occurrence of a silent data corruption (SDC). That is, three or more shift errors occur while reading a single extended codeword (as GreenFlag(n,k,6) is a DECDED shift error code). This probability depends on the length of the codeword n and the probability of a single shift operation incurring an error. However, this probability is not well known as racetrack memory is still an experimental technology. Prior work [85] has estimated shift error probabilities based on models and simulations but currently, there are no empirical results. Thus, we test GreenFlag over a wide range of single shift error probabilities

ranging from 10^{-6} to 10^{-9}. We account for both double shift errors and the possibility of multiple single shift errors in the same extended codeword.

We also calculate the mean time to failure (MTTF) for GreenFlag i.e., the expected time in years until GreenFlag incurs an SDC. MTTF depends both on the P_f and on the memory bandwidth demand. The dependence on bandwidth distinguishes racetrack memory from other technologies, as racetrack errors only occur during shifts and thus the more the racetrack is used (shifted), the more likely it is that a shift error will occur. Thus, we calculate MTTF based on different shift intensities that represent peak bandwidths for a variety of memory technologies like DDR3, SSDs, and 3D-DRAM. Overall, MTTF can be calculated as,

$$MTTF = \frac{1}{P_f} \times \frac{n}{Bandwidth} \times \frac{1}{3.154 \times 10^7}$$

where $3.154 \times 10^7$is the number of seconds in one year and n is the length of a codeword.

4.6 Evaluation

In this section, we evaluate GreenFlag in terms of performance and fault tolerance. The goal of this evaluation is to investigate the viability of 3D racetrack with effective fault tolerance and understand in which part of the memory hierarchy it could be used.

4.6.1 Block Read and Write Latency

For 64-byte blocks, Figure 4.5 presents the block read and write latency of GreenFlag for different rates (i.e., for different codeword lengths). These latencies include

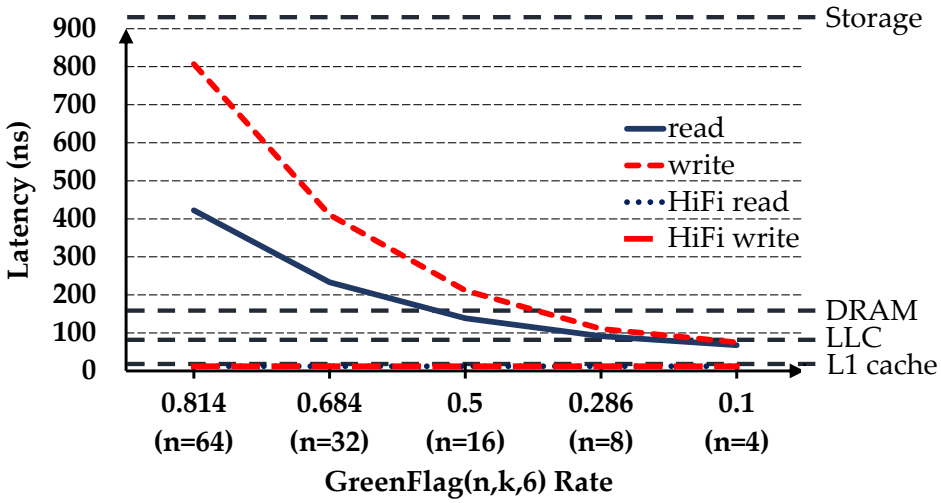


Figure 4.5: Block access latency.

shift, read/write and decoding/encoding delays. For example, GreenFlag(8,4,6) has 8-bit codewords and 6-bit delimiters, and reading an extended codeword requires 8+6 shifts and 8+6 reads; its rate is 0.286. For comparison, we also include the latency of prior work on tolerating shift errors (HiFi) [85].

The results of Figure 4.5 show that racetrack memory with GreenFlag cannot achieve the latency expected of an L1 cache and probably not even the latency of a last-level cache (LLC). GreenFlag(4,1,6) can provide the lowest read latency (67ns) but with a rate of only 0.1. Even if we use GreenFlag(4,1,4) (that provides SECSED shift protection) read latency will only drop to 56ns and the rate will be 0.125.

However, the results show that racetrack with GreenFlag is a viable option for main memory and storage devices. GreenFlag(32,26,6) and GreenFlag(16,11,6), with rates of 0.684 and 0.5, respectively, can provide latencies comparable to modern DRAM designs like DDR3 (~100ns), while GreenFlag(64,57,6) can be used as an extremely low latency

storage device with a rate of 0.814. While these rates may still seem somewhat low we emphasize that GreenFlag is, to the best of our knowledge, the only DECDED shift error solution that can be implemented with only one read/write port. Compatibility with a single port makes GreenFlag the only DECDED coding scheme that can be used with 3D racetrack and benefit from its superior density compared to 2D racetrack [59, 34, 67]. On the contrary, HiFi has very low latency but cannot be implemented on 3D racetrack as it uses multiple read/write ports.

4.6.2 Bandwidth per Track

In Figure 4.6 we present GreenFlag's maximum data bandwidth per track. Bandwidth per track is a more insightful metric than aggregate bandwidth, at this early stage in the development of racetrack memory, because it eliminates orthogonal issues like I/O bottlenecks and open questions about how large-scale racetrack will be organized. We calculate the bandwidth per track as the average number of data bits we can read or write per track per second, including delimiter and encoding/decoding overheads.

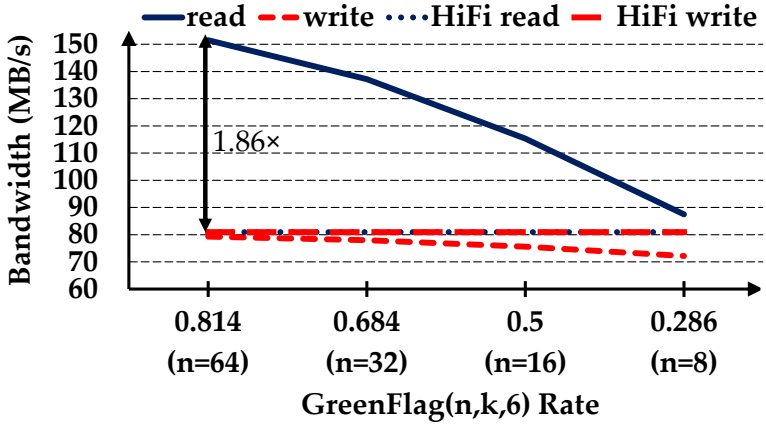


Figure 4.6: Bandwidth per track.

We observe that, as the rate increases, the read and write bandwidth per track of GreenFlag also increases. This result occurs because we are accessing more data bits per codeword (higher rate), while the decoding and encoding overheads remain almost constant. GreenFlag achieves up to 1.86× more read bandwidth per track compared to HiFi, while providing similar write bandwidth. The reason is that HiFi must perform a write after every shift operation, which limits how many bits it can stream per second.

4.6.3 Energy

We calculate the average energy per bit for read and write operations. For GreenFlag this is calculated by amortizing the cost of an extended codeword, including the decoding and encoding energy overheads, per bit. Figure 4.7 shows that the energy overheads of GreenFlag decrease as rate increases. Additionally, we observe that the energy overheads are significantly lower than HiFi (up to 6.6×).

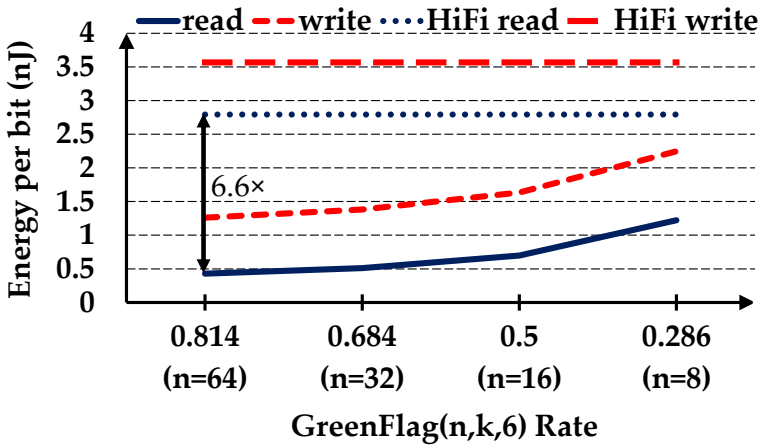


Figure 4.7: Energy per bit.

4.6.4 Mean Time to Failure (MTTF)

Figure 4.8 presents the failure probability P_f (i.e., the probability of having an SDC) per extended codeword for different GreenFlag codes and different single shift error probabilities; note that values on y-axis are in decreasing order. We observe that P_f increases as the codeword length increases. This was expected as longer codewords result in more shift operations and thus higher P_f. Additionally, we observe that P_f linearly increases with respect to the single shift error probability.

Figure 4.9 presents the MTTF for GreenFlag(64,57,6) and GreenFlag(32,26,6) for different bandwidths. For these calculations, we assume that the racetrack memory provides dataword bits with a constant bandwidth throughout its lifetime. These bandwidths represent the average bandwidth usage under different memory system designs (i.e., storage, main memory, and high bandwidth memory). We set a desired MTTF goal of 10 years. For very high shift error probabilities (i.e., greater than 10^{-7}), it is

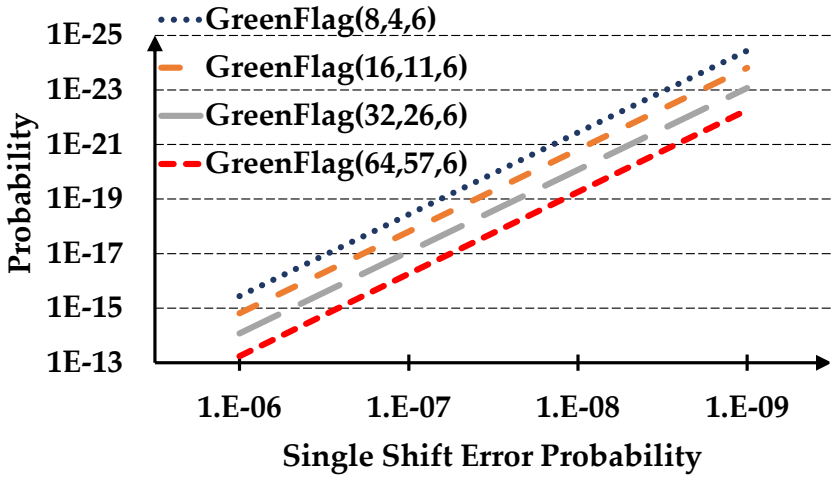


Figure 4.8: Probability of SDC error per extended codeword.

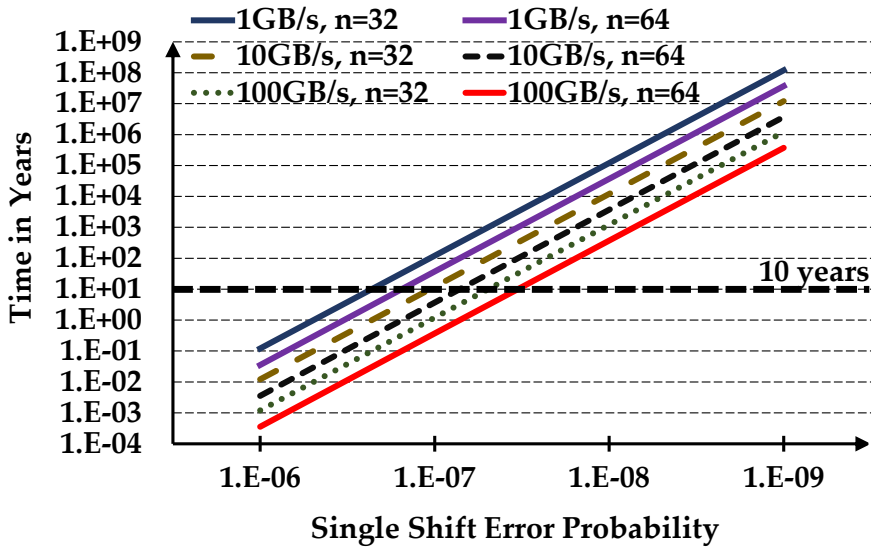


Figure 4.9: MTTF for different GreenFlag(n,k,6) and different bandwidth intensities.

difficult to achieve that goal. However, for lower probabilities, we can achieve MTTFs well beyond 10 years. Overall Figure 4.9 can be used to extrapolate MTTFs for any single shift error probability and can be used as a guideline when designing reliable memory systems based on racetrack memory.

4.7 Conclusion on GreenFlag coding

Through this chapter, we have presented GreenFlag, the first error tolerance scheme for 3D racetrack memory that can provide DECDED shift error protection. GreenFlag combines a novel construction for Varshamov-Tenegolts codes with specially crafted delimiter bits to detect, categorize, and correct shift errors. Additionally, we designed and synthesized hardware for encoding and decoding so that we could estimate latency and energy overheads and, more significantly, analyze the viability of racetrack for different levels of the memory hierarchy. Based on current technology, it appears that

fault-tolerant 3D racetrack memory is not attractive for caches but a promising option for main memory and storage designs.

5. Increasing the Fault Tolerance of HBM

In this chapter, we introduce Jenga, a new scheme for protecting 3D DRAM, specifically HBM, from failures in bits, rows, banks, channels, dies, and TSVs. Jenga's key innovation—which enables better performance and less complexity than previous work with similar fault tolerance goals—is to provide additional redundancy within the granularity of a cache block instead of across multiple blocks. By not involving multiple blocks, Jenga's procedures for writes and error recovery are much simpler. Jenga requires no modifications to the existing HBM protocol.

We start this chapter by discussing the fault tolerance in HBM and how it differs from typical 2D DRAM in Chapter 5.1. We present prior work on fault tolerance for HBM in Chapter 5.2. We then present our proposed solution (Jenga) in Chapter 5.3. We discuss how Jenga detects and corrects errors in every granularity of the HBM (bits, row, banks, etc.) in Chapter 5.4. Our methodology and evaluation of Jenga are presented in Chapters 5.5 and 5.6 respectively. We conclude Jenga in Chapter 5.7.

The main contributions of this chapter are summarized here:

- We present Jenga, a technique that allows us to tolerate failures in bits, rows, banks, channels, dies, and TSVs of the HBM.
- We use a cycle accurate simulation (gem5) to evaluate Jenga and show that it can outperform prior work both in error-free scenarios and in the presence of errors.

- We show that Jenga's runtime only increases by 1.03× when compared to an unmodified baseline (with non-sufficient fault tolerance), thus enabling HBM to be used in systems that require both high bandwidth and fault tolerance.

5.1 Fault Tolerance in HBM

All DRAM technologies (i.e., including HBM) have been shown to be vulnerable to transient and permanent bit errors [68, 87]. To tolerate these errors, memories are often protected with error correcting codes (ECC). Although there are many types of ECCs, memory is often protected with codes that provide single error correction and double error detection (SECDED). A commonly used SECDED code is Hamming (72,64), in which a 64-bit dataword is encoded as a 72-bit codeword. In a common implementation of this code with x8 DRAM chips, a ninth additional chip is used. The 64 bits of each dataword are striped across the first eight chips, and the 8 ECC bits are on the ninth chip. To tolerate the problem of "chipkill," in which an entire DRAM chip fails, the data and ECC bits can be striped across chips such that the loss of any one chip can be tolerated (i.e., no chip holds more bits of a given codeword than can be corrected by the code) [23].

Naively, one might expect to be able to straightforwardly adapt SECDED codes (without chipkill tolerance) to HBM. In an HBM stack with 8 channels, one could imagine adding a ninth channel for ECC and striping the data across all nine channels as is done with 2D DRAM. Unfortunately, this design, while good for fault tolerance, would greatly hurt performance. Each read or write would require accessing every channel and would

thus eliminate much of the bandwidth benefit of using HBM. Furthermore, as each HBM channel is controlled by a separated memory controller (as seen in Figure 5.1), for a single read or write request the last level cache (LLC) would have to issue nine requests to the nine different memory controllers, and thus increase the on-chip network traffic by 9×. Furthermore, adding a ninth channel could add fabrication challenges in the HBM stack.

Because dedicating a channel in the stack for ECC has many challenges, the HBM2 standard [3] specifies memory channels that contain rows that are wide enough to store data and ECC bits together as shown in Figure 5.1. This collocation of data with ECC is good for performance because each memory request needs to access only one channel (instead of one channel for data and one for the ECC bits). However, this collocation provides less fault tolerance. Specifically, with collocated data and ECC, the 3D DRAM cannot tolerate failures of rows, banks, channels, dies, or TSVs. For example, a single hard error in one of the 256 data TSVs of a given channel would result in 2 (or 4) bit errors for every block access of size 32B (or 64B) from every bank/row of that channel. Then, the

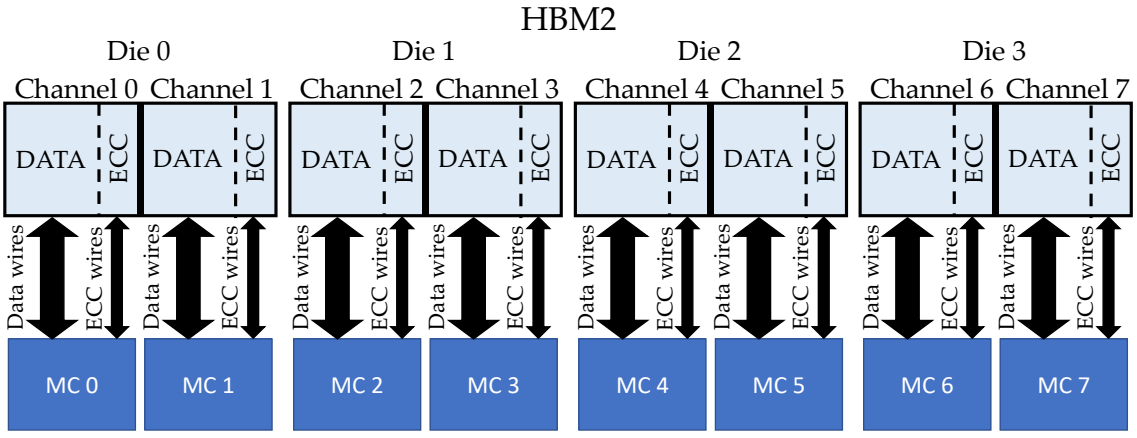


Figure 5.1: The logical organization of HBM2.

occurrence of a single transient error in any of the rows of that channel would result in an uncorrectable error.

5.2 Prior Work

There are four prior works that stand out in fault tolerance for 3D DRAM [41, 19, 39, 57]. All of them share the same high level structure as Jenga. They all have two levels of error coding. A first level code (denoted L1C) that provides error detection and perhaps a small amount of error correction, and a second level code (L2C) that provides correction for large-scale errors. The L2C in all prior schemes protects a given number of blocks, say N, by maintaining a redundant block that is the XOR (i.e., parity) of the N data blocks. These schemes use spatial redundancy to spread the N data blocks and one parity block such that no single fault is unrecoverable.

This structure tolerates many faults at relatively low capacity cost, but it has four drawbacks, all due to the L2C being a parity across multiple blocks. Consider a data block A in a parity group with parity block P (i.e., $P = A \oplus B \oplus \ldots$):

- Every write to A requires first reading the old values of A and P. The old value of A must be XORed out of the old value of P, and the new value of A must be XORed into P. Then A (with its L1C) and P can be written.[1]

[1] There are ways to reduce that penalty (e.g., with caching and proactive XORing of the old values of blocks), but we prefer to avoid it entirely.

- Writing to *A* and writing to *P* must be atomic with respect to other writes to other blocks in *A*'s parity group. If *P* can be cached, this involves modifying the coherence protocol and being careful to avoid deadlock. If *P* cannot be cached, this involves memory controller support to "lock" *P* until the writes to *A* and *P* complete.
- Correcting an error in *A* with the L2C requires accessing all blocks in *A*'s parity group. Moreover, until the correction completes, no blocks in *A*'s parity group may be written. Once again, these requirements may affect the coherence protocol and/or memory controller.
- Most prior work [19, 39, 57] (but not [41]) assumes a unified memory controller that is aware of all memory. However, each HBM channel has its own deticated controller that is only aware of that channel's memory. Without a unified controller, the LLC bank that submits a memory request would be burdened with calculating parity bits, issuing extra requests to load and store the parity bits, and identifying and correcting errors.

Jenga overcomes these undesirable aspects of prior work with one key innovation; by adding redundancy at the granularity of a single block instead across multiple blocks. We now discuss the key characteristics of each of the prior work and summarize them in Table 5.1. For completeness, we also include the key characteristics of Jenga.

Efficient RAS (E-RAS) [39] distributes the parity of the L1C and L2C so that data and parity are not collocated. Additionally, the L2C is spread across all the different

Table 5.1. Comparing Jenga to Prior Work

Issue	**E-RAS** [39]	**Citadel** [57]	**RATT-ECC** [19]	**Parity-Helix** [41]	**Jenga**
Read Block	**2 reads** on the same channel (sequential).	**2 reads** to different channels (1 of which is always ECC channel)	**2 reads** to different channels (1 of which is always ECC channel)	**1 read** to a single channel	**2 half-block reads** to different channels
Write Block	**2 reads, 3 writes** (2 channels)	**4 reads, 5 writes** (2 channels, 1 of which is ECC channel)	**2 reads, 3 writes** (2 channels, 1 of which is ECC channel)	**2 reads, 2 writes** (2 channels)	**0 reads, 3 half-block writes** (3 channels)
Correct Error	**all channels** 1 read each	**all channels** multiple reads each	**all channels** multiple reads each	**all channels** 1 read each	**1 channel** 1 half-block read
Level-1 Code	detection: CRC-8	detection: CRC-32	detection and some correction: Reed-Solomon	detection and some correction: Hamming(72,64)	detection and some correction: Hamming(72,64)
Level-2 Code	1-dimensional XOR **across channels**	3-dimensional XOR **across rows, banks and channels**	2-dimensional XOR **across banks and channels**	1-dimensional XOR **across channels**	1-demensional XOR **intra-block**
Hardware modifications	none	added ECC channel	added ECC channel	none	none
Normalized host-visible capacity	0.84	0.86*	0.875*	0.875**	0.66
Centralized HBM controller?	yes	yes	yes	no	no

***0.8 assuming ECC-die instead of ECC-channel** ****0.75 assuming 2 Channels per Die instead of 1.**

channels to increase reliability. Citadel [57] and RATT-ECC [19] use an additional dedicated ECC channel to store the L1C and L2C parity. That ECC channel can become a bottleneck as it needs to be accessed for every memory read and write. Although Citadel, RATT-ECC, and E-RAS achieve high fault tolerance with relatively high host-visible capacity, they do not provide tolerance for die or channel failures. Parity-Helix [41] uses the ECC that is collocated with the data as the L1C, and the L2C is distributed in a modulo fashion across different channels. Parity-Helix's use of spatial redundancy allows it to tolerate die and channel failures. However, it induces significant performance overheads and coherence challenges as the L2C parity is generated across multiple blocks.

5.3 Jenga Coding

We propose Jenga coding for 3D DRAM in order to provide comprehensive fault tolerance while minimizing costs (due to extra storage) and performance degradation (due to extra accesses). As with any fault tolerance scheme, it requires redundancy, and redundancy inherently has costs.

We designed Jenga for HBM stacks that comply with the HBM2 standard, but there is nothing fundamental about that baseline system model other than that it provides ECC collocated with the data in each row. Jenga requires no modifications to the HBM2 standard. We assume the ECC is a Hamming(72,64) code that provides SECDED protection at a cost of 12.5%. That is, each 64B cache block requires an extra 8B for ECC, and we assume that block accesses involve all 72B. Like prior work [41, 19, 39, 57], we

assume the memory controllers implement the ECC logic (i.e., errors are detected and corrected by the memory controllers) and can diagnose failed rows, banks, channels, and chips.

Jenga, like prior work, uses two levels of error coding. Like Parity-Helix [41], the L1C used by Jenga is Hamming(72,64). However, Jenga innovates in the design of the L2C code to tackle the shortcomings and challenges of prior work.

5.3.1 Jenga's Finer-Grained L2C Redundancy

As we discussed in Chapter 5.2, all prior schemes [41, 19, 39, 57] face significant performance and coherence challenges due to cross-block parity used as L2C. Jenga overcomes the undesirable aspects of prior work with one key innovation: adding redundancy at the granularity of a single block, instead of across multiple blocks. Every memory block A (64B) is divided into two sub-blocks, A_1 (32B) and A_2 (32B). We then perform a bit-wise XOR between A_1 and A_2 to produce a redundant sub-block A_3 (32B) that is the parity of sub-blocks A_1 and A_2. Thus, reading any two of the 3 sub-blocks can recreate our data. Concatenating A_1 and A_2 recreates block A or we can perform a bitwise XOR of A_1 (or A_2) with A_3 in order to generate A_2 (or A_1) and then concatenate the two to again recreate block A.

With this technique, reading or writing block A does not require us to access any other block, and thus avoids possible coherence or consistency issues.

5.3.2 Adding Spatial Redundancy

Like prior work, Jenga exploits spatial redundancy and distributes sub-blocks of the L2C across different dies and channels, so that we can guarantee protection against both die and channel failures. For a given block A, Jenga stripes the three sub-blocks A_1, A_2, and A_3 in different channels and different dies, and it does this striping in a modulo-like fashion as shown in Figure 5.2. To explain the mapping, we use an example in which, for unmodified HBM (i.e., without Jenga), block A would have been stored in channel 0, row 0, and columns 0 and 1 (each column is 32B wide).

We treat the parity sub-block A_3 differently from the original sub-blocks A_1 and A_2. To store A_3, we first need to find space in our HBM. Because we do not want to add extra dies (i.e., assume HBM stacks with an arbitrary number of dies), we choose to sacrifice the host-visible capacity of our memory. As illustrated in Figure 5.2, Jenga uses 2/3 of the total capacity to store data and reserves 1/3 of the total capacity to store the additional parity sub-blocks like A_3. In our example, Jenga stores sub-block A_3 in channel 4 of that reserved

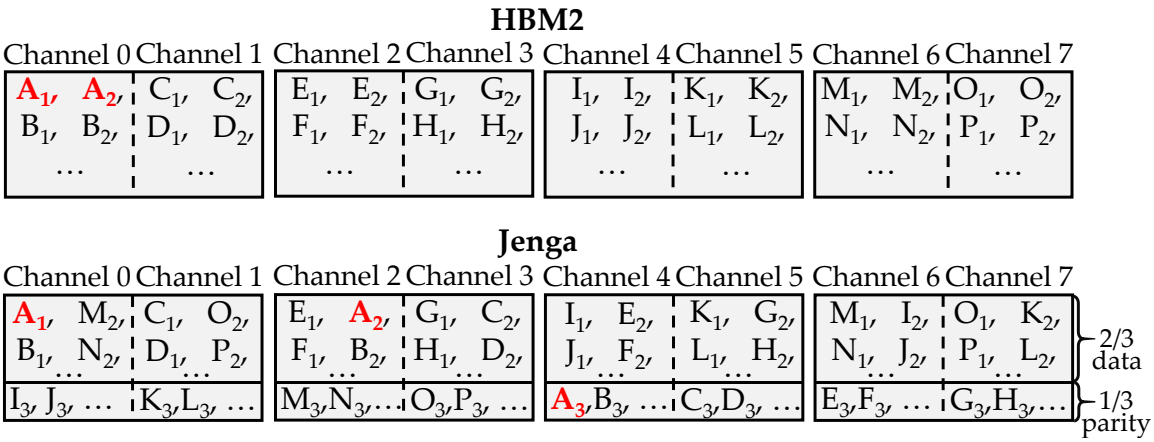


Figure 5.2: Comparing Jenga to HBM2, in terms of mapping (sub-)blocks to channels

space. Observe that this space is enough to store all the extra sub-blocks (A_3, B_3, …) corresponding to blocks that were originally stored in channel 0 (A, B, …).

There are many functions that work for mapping the sub-blocks A_1 and A_2, and we can choose from among those that best preserve DRAM row locality. In our example, sub-block A_1 is in channel 0, and it is at row 0, column 0. The mapping function must ensure that (a) a sub-block from some other block gets mapped to channel 0, row 0, column 1, and (b) sub-block A_2 gets mapped to some column 1 in row 0 of another channel (channel 2 in our example). The function we implement and present in Figure 5.2 interleaves the sub-blocks such that if block A is located in channel i under HBM2, then with Jenga A_1 is located in channel i, A_2 in channel (i+2) modulo 7, and A_3 in channel (0+4) modulo 7.

5.3.3 Reads

On a read miss in the last-level cache (LLC), the LLC bank acquires a block of data by accessing two different HBM channels on two different chips. Thus, a memory read generates two separate half-block read requests. Although the overall amount of data we read does not change, we do access two channels rather than one. Additionally, a read is only completed once both channels have replied.

Because Jenga needs only two of the three sub-blocks to obtain a block, we have the opportunity to choose which two we access, and there are several ways one could try to exploit this freedom of channel selection. For example, if one channel is malfunctioning

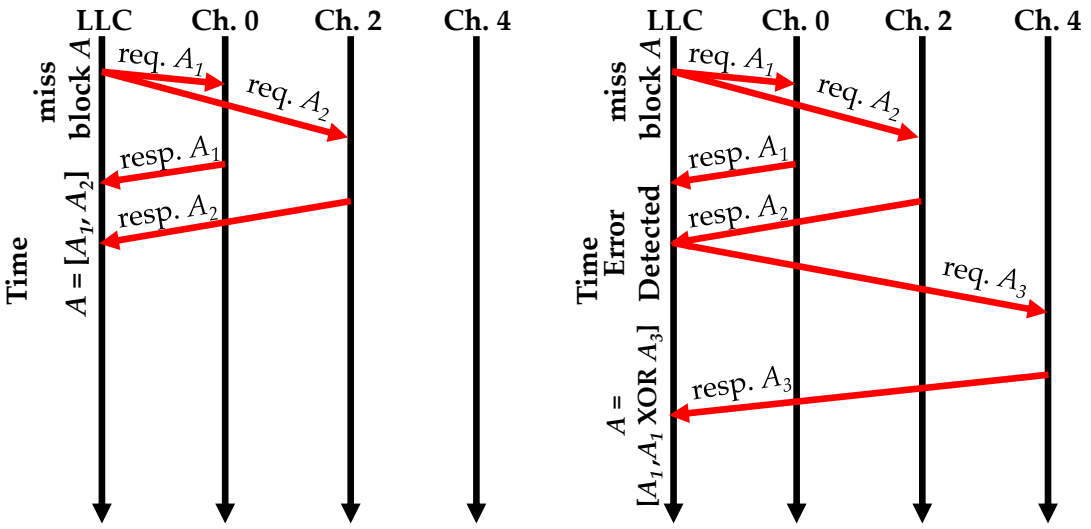


Figure 5.3: The life of a LLC miss request in the (a) absence of errors and (b) presence of errors where the third sub-block is needed to perform correction.

(e.g., due to multiple TSV failures), then we always want to access the other two. Additionally, it would be possible to select the channels with the least occupied queues. To keep our implementation simple, Jenga selects the channels depending on their Manhattan distance from the LLC bank that issued the corresponding request.

Figure 5.3(a) presents a timing diagram of an LLC read miss in Jenga in the absence of errors. The LLC controller issues two read requests to channels 0 and 2 in order to retrieve sub-blocks A_1 and A_2. When both channels reply, data block A is reconstructed by concatenating the two sub-blocks.

5.3.4 Writes

To perform a memory write, Jenga generates three separate half-block accesses to different channels on different chips. In our running example, writing block A requires

us to update all three sub-blocks A_1, A_2 and A_3 and thus access channels 0, 2, and 4. Jenga also requires the computation of A_3 as the logical XOR of sub-blocks A_1 and A_2, but this is simple and adds minimal overhead.

5.3.5 Error Detection and Correction

As we discussed in Chapter 5.3 Jenga uses Hamming(72,64) as L1C to provide SECDED protection. If SECDED cannot correct an error, Jenga then uses the third sub-block of the L2C to recreate the correct data. For example, assume that we are reading block A and thus issued read requests for A_1 and A_2. As shown in Figure 5.3(b), if SECDED cannot correct errors in sub-block A_2, then the LLC controller requests A_3. When the LLC has A_1 and A_3, it can bit-wise XOR them to recover A_2 and thus obtain A.

5.3.6 Half-Block Accesses

Typically, the majority of accesses to memory are at least one block. However, systems occasionally perform half-block accesses, for a variety of reasons, including segmented cache designs [32, 45, 63, 83], I/O operations, or if the LLC maintains dirty bits at a sub-block granularity.

For Jenga, half-block reads require less work. Instead of performing the two sub-block reads required to obtain a full block, Jenga would need to perform only one sub-block read. (An uncorrectable SECDED error in that sub-block could be reconstructed from the other two sub-blocks.) A half-block write requires reading two of the three sub-blocks before writing these two sub-blocks to correctly compute the parity sub-block. This

need to read before writing is a drawback that prior work has for *all* writes (not just sub-block writes), and that Jenga has only for sub-block writes.

5.3.7 Costs

We now summarize the costs of Jenga, bearing in mind that all fault tolerance schemes require redundancy and more fault tolerance incurs more cost.

- Storage: For each block, Jenga adds another half-block of storage. Thus, Jenga memory has 2/3 of the host-visible memory as the baseline HBM2.
- Read latency: Each read requires waiting for the completion of two concurrent half-size accesses to different channels on different chips. Ideally, these latencies overlap but that overlap will not be perfect. Each half-block access takes somewhat less time than a full-block access, which slightly benefits Jenga.
- Write latency: Each write requires three half-size writes. Writes are rarely on the critical path of performance, and thus only bandwidth really matters.
- Bank conflicts: Jenga may introduce more HBM bank conflicts due to having more accesses (even if each access is smaller), and these bank conflicts may degrade performance. (On the flip side, Jenga may increase the number of row hits.)
- Interconnection network bandwidth: Reads require approximately the same bandwidth as the baseline (two half-block requests and responses compared to one full-block request and response) if we ignore packet header overheads. Writes require

1.5× the bandwidth as the baseline (three half-size requests and responses compared to one full-size request and response).

- Power/Energy: The power and energy overheads of Jenga are a function of how much extra work it does and thus track closely with its bandwidth overheads.

5.4 Error Detection and Correction Analysis

In this section, we analyze Jenga's ability to tolerate errors. We consider one block, A, and its three sub-blocks. If at least two of its three sub-blocks are correct, Jenga will produce correct data when reading A. Thus, the analysis devolves to determining when at least two of the three sub-blocks are correct.

An additional subtlety arises because the SECDED L1C (i.e., Hamming(72,64)) is at a finer granularity (64 bits) than a sub-block (32B per sub-block in DRAM). Consider the case where A_1 has a detectable but uncorrectable error (DUE) in its first 64 bits (i.e., two bit-flips), and A_2 has a DUE in its last 72 bits. Assume A_3 is error-free. Even though two of the three sub-blocks have DUEs, Jenga could still reconstruct A because the first 64 bits of A_1 can be reconstructed from A_2 and A_3 and the last 64 bits of A_2 can be reconstructed from A_1 and A_3. To achieve this benefit, we would need to modify the memory controllers such that, when they detect a DUE, they still send to the LLC the data as well as the locations in the data where DUEs had been detected.

5.4.1 Individual Bit Errors

Any number of 64-bit datawords with a single bit error will be corrected by the SECDED code. A 2-bit error in a 64-bit dataword in a given sub-block will be uncorrectable by SECDED, but it will be corrected by using the other two sub-blocks as long as neither of those sub-blocks has a multi-bit error in the same dataword position.

5.4.2 TSV Failures

Any single TSV failure will cause two bit errors in one of the three sub-blocks (128 TSVs used twice to read 32B). These bit errors are separated by 128 bits and thus will be corrected by the SECDED code. However, once a TSV has failed, a subsequent bit error can cause a DUE in one 64-bit dataword in one sub-block. This DUE will force Jenga to fetch the third sub-block and thus produce correct data.

Multiple TSV failures can be tolerated if the failed TSVs are separated by at least 64 TSVs, in which case the corresponding bits that are read through the faulty TSVs are protected by separate SECDED codes (separated datawords). Additionally, in the absence of individual bit errors, any two arbitrary TSV failures can be corrected.

5.4.3 Row and Bank Failures

Each of these large-scale failures can cause one of the three sub-blocks to be uncorrectable by SECDED. Jenga will still have two correct sub-blocks and thus produce correct data. Multiple row and bank failures can be tolerated as long as no two rows or banks that store sub-blocks of the same cache block fail. For example, in the case that both

the rows that hold sub-block A_1 and sub-block A_2 fail then block A can no longer be recovered. However, considering the number of rows and banks in all channels this is an unlikely event.

5.4.4 Chip and Channel Failures

Chip failure is a well-known failure mode for DRAM chips [23, 53], and channel failure can occur for a variety of reasons including a permanent fault in a channel controller. Because any chip holds no more than one of the three sub-blocks, a chip failure can be tolerated like a channel failure. Jenga tolerates one channel or chip failure, and it can tolerate two channel failures if both channels are on the same chip.

5.4.5 Using Different Collocated Error Codes

In Jenga, we assume the use of a typical SECDED code, like Hamming(72,64). However, we could use other codes that offer different trade-offs between cost and fault tolerance. A straightforward change that would improve fault tolerance but reduce the host-visible capacity would be to strengthen the code from SECDED to DECTED (double error correcting and triple error detecting) or even stronger. A slightly different approach, with a subtler trade-off, would use a code like CRC that has very strong error detection capability but no error correction ability. With CRC, Jenga would detect some errors that are missed by SECDED (or DECTED) and enable them to be corrected using the third sub-

block. However, with CRC, Jenga would not be able to correct *any* errors without having to resort to fetching the third sub-block.

5.5 Methodology

5.5.1 Simulation Methodology

We simulate Jenga with the gem5 simulator [11] in full-system mode. We use the Ruby memory system simulator for modeling and simulating our memory system (interconnection network, caches, HBM, coherence protocol etc.) in detail. By using Ruby, our simulation results will include any traffic overheads and delays that may occur because of the additional read and write memory requests that Jenga imposes.

Table 5.2: System Configuration

Multicore Processor Chip	
Cores	32 out-of-order cores
L1 Caches	private per-core, 32KB
L2 Caches	shared 4MB, 32 banks
Interconnection Network	2D mesh
Coherence Protocol	MESI with state at L2 cache
HBM Configuration	
Number of Dies	4 dies in one stack
Channels: number and size	2 @ 256MB each
Banks per Channel	16
Bank Groups per channel	4
Number of TSVs per Channel	128
HBM Timing Details	
Clock Cycle	t_{CK} = 2ns
Row Precharge	t_{RP} = 15ns
Row to Column Address Delay	t_{RCD} = 15ns
CAS Latency	t_{CL} = 15ns
Row Active Time	t_{RAS} = 33ns
Burst Delay	t_{BURST} = 2ns

5.5.2 Benchmarks

Our benchmarks are a set of multithreaded programs from the PARSEC benchmark suite [10]. We include benchmarks with different memory demands, and we run them under large size inputs with 32 threads. The benchmarks with high memory demand, like canneal, are most relevant, but we run all of them for completeness.

5.5.3 System Model

5.5.3.1 System Organization

The system configuration is summarized in Table 5.2. We simulate a multicore chip with 32 cores that communicate through a mesh interconnection network. The 8 HBM channels are connected in a diamond-like fashion across the mesh as shown in Figure 5.4. We choose the diamond-like connectivity as this configuration has shown to be better for balancing traffic in the network [7].

5.5.3.2 HBM

We simulate an unmodified HBM stack with 4 dies. Each die consists of two channels (i.e., 8 channels in total). Each channel connects to our system through 128 TSVs. Each channel contains 4 bank-groups that consist of 4 banks each. A bank contains 8192 rows that store 2KB of data and the appropriate ECC bits. In total, each channel can hold 256MB (i.e., 2GB per stack). More details about the timing specifications of the simulated HBM can be found in Table 5.2.

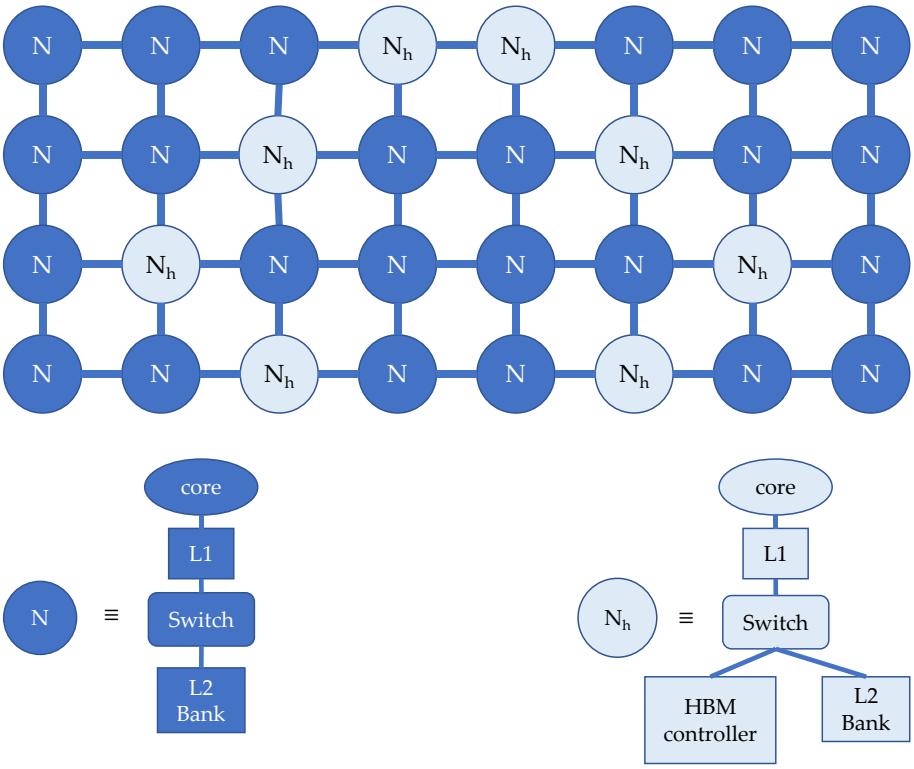


Figure 5.4: The nodes of the system and how they are interconnected. Only nodes marked as N_h connect to an HBM channel

5.5.4 Comparison Schemes

We compare Jenga against two schemes. First, we consider a baseline system that uses unmodified HBM with just the collocated SECDED code. Second, we consider a scheme that, like prior work, uses an L2C that maintains parity across multiple blocks (denoted as "PW" in figures). Our PW implementation closely resembles Parity-Helix [41] with one exception. To make the comparison fair—so that Jenga and prior work have the same raw capacity—we assume that the L2C in PW uses two blocks from different channels and logically XORes them to produce a parity block. The three blocks in each parity group are spread across different channels in a modulo fashion (similar to Jenga)

in order to ensure spatial redundancy. Jenga and PW results will be normalized to the baseline.

5.6 Evaluation

We now present the results of our simulations. We focus on measuring the impact of Jenga on performance in the absence and presence of memory errors.

5.6.1 Error-Free Performance

In Figure 5.5, we present the runtime for all the benchmarks, normalized to that of the baseline. The benchmarks are organized, from left to right, in order of decreasing memory demand [10], because memory demand is a key factor in these performance results.

The key result is that the performance of Jenga is very close to that of Baseline. For workloads with light memory demand, that is unsurprising. However, even for workloads with greater memory demand, like canneal, performance is almost identical

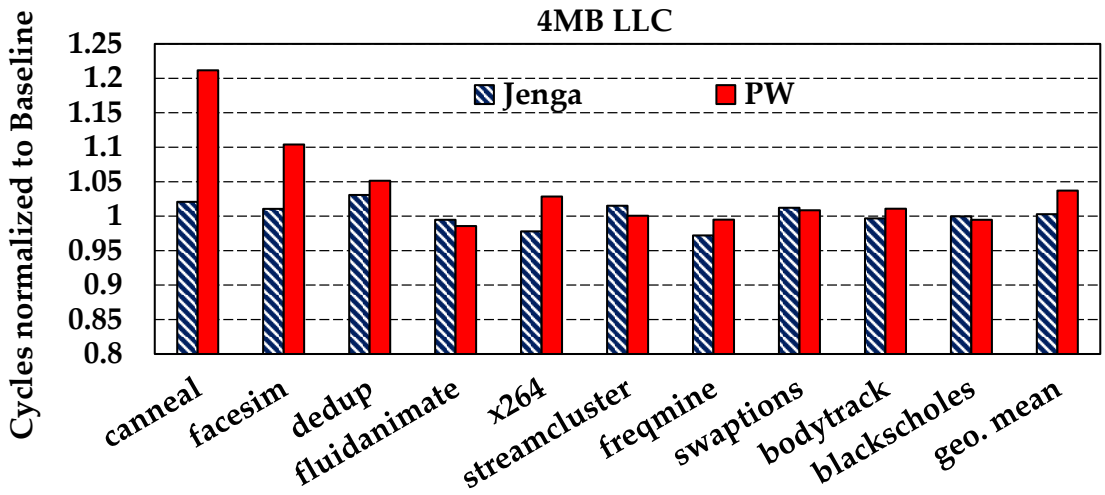


Figure 5.5: Normalized runtime in the absence of errors using a 4MB LLC.

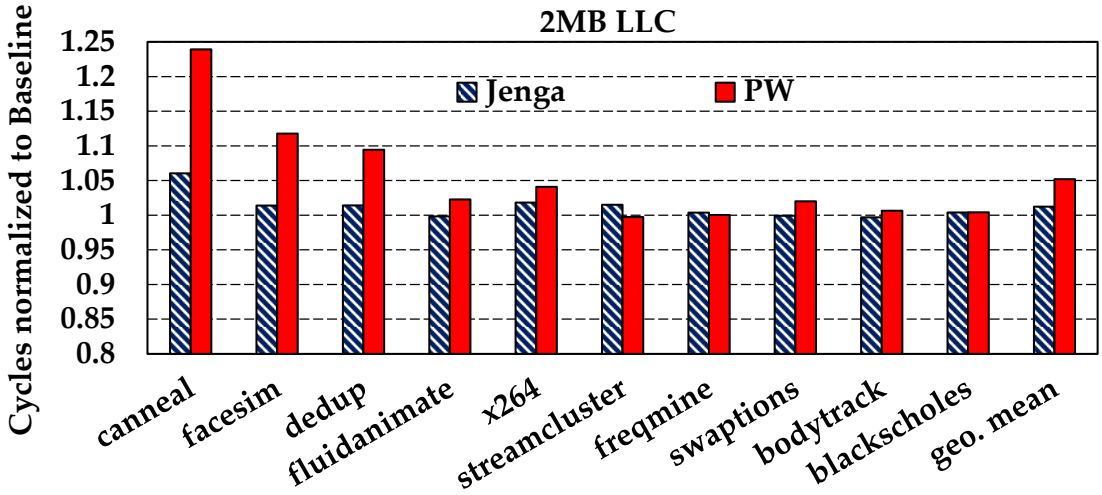


Figure 5.6: Normalized runtime in the absence of errors using a 2MB LLC.

(less than 1.03× runtime comparing to Baseline). Conversely, we observe that PW suffers a significant performance overhead for benchmarks that are memory intense. To highlight the impact of memory demand, we ran the same experiment with a smaller L2 cache; the results in Figure 5.6 show that performance degradation tends to increase. More specifically, Jenga achieves an average speedup of 1.11× over PW for the 2MB LLC across the 3 most memory intense benchmarks (canneal, facesim, dedup).

These results are primarily due to memory bandwidth and memory latency differences, although differences in the interconnection network traffic could potentially also have an impact. The key difference in memory latency is that Jenga, unlike PW, does not have to issue reads before each write. Jenga's reads can take a bit longer than PW's, because Jenga has to wait for both sub-block reads to complete, but those latencies often overlap considerably. Both Jenga and PW increase the memory bandwidth demand during writes. However, Jenga only increases the memory bandwidth by 1.5× (writing 3

sub-blocks) while PW by 4× (reading 2 blocks and writing 2 blocks). Both Jenga and PW place more traffic on the interconnection network compared to Baseline. However modern interconnection networks tend to be highly overprovisioned [31, 30], and thus this increase has a relatively small impact on performance.

One anomalous result is that Jenga is sometimes marginally faster than the Baseline. This is a minor artifact and not a claimed benefit of Jenga, and there are two phenomena that contribute to it. First, Jenga's reads are half the size of Baseline's reads and thus each one can be completed slightly faster; if the two reads by the L2 overlap their latencies perfectly, a Jenga read will incur a slight benefit. Second, because the LLC issues two half-block read requests for each read miss, it opens two different DRAM rows at the same time. For certain memory access patterns, the second row opening is useful for future requests (i.e., like a prefetch) and the latency to open the row is overlapped with that of the first row.

Jenga could also impact performance indirectly if its multiple requests and responses cause significantly more congestion in the interconnection network. To study this phenomenon, we measure the average utilization of the on-chip mesh network. Although only writes to memory generate more data than the baseline (96B rather than 64B), the multiple requests (data and packet header) need to travel through the mesh in order to find the destination channel. Regardless, our results reveal low network utilization and negligible differences with respect to the baseline.

From these results, we conclude that although the error-free performance degradation due to Jenga is small, it is more evident for memory intensive applications. It is possible that benchmarks with even more memory demand than parsec—or systems with many more threads—could reveal a larger performance penalty for Jenga. However, Jenga achieves a significant performance gain comparing to PW and this performance gap only increases for higher memory demand benchmarks.

5.6.2 Performance in the Presence of Errors

We consider two scenarios. First, as an extreme stress test, we consider the worst case in which every pair of read requests results in an uncorrectable error in the second response to arrive at the L2. Thus, after the second response arrives, the L2 issues an additional request to the third half-block in order to recreate the block. Second, we present a somewhat less extreme scenario in which 1% of every pair of read requests results in an uncorrectable error in the second response to arrive at the L2. We focus these tests on the canneal benchmark, whose performance is more greatly affected.

In Figure 5.7 we observed that even for the improbable worst-case error scenario on our most memory intensive benchmark, Jenga incurs an overhead that is less than 40% (~70% for PW). Jenga with 1% error rate has performance only slightly worse than that of the error-free Jenga. We can safely conclude that Jenga's performance impact in the presence of occasional errors will be minimal.

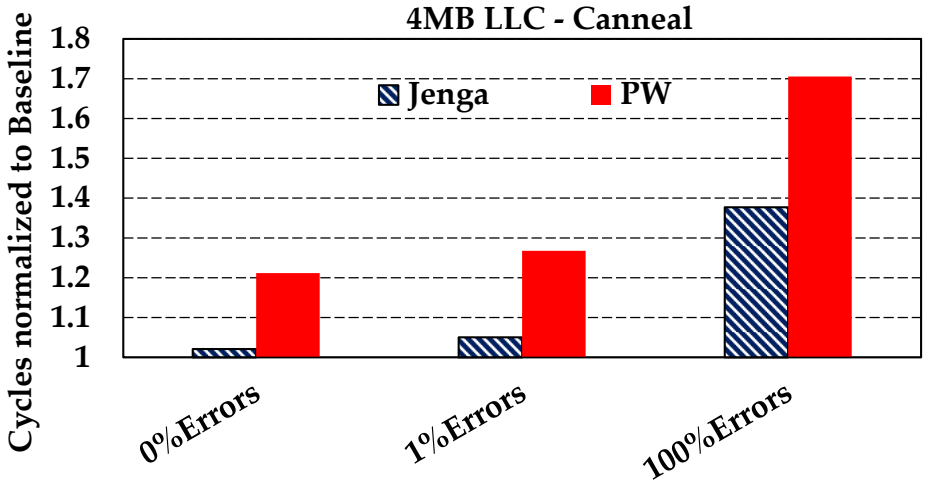


Figure 5.7: Normalized runtime in the presence of errors where L2C is used to recover the correct data. Reported results are for the canneal benchmark using a 4MB LLC.

5.6.3 Capacity Cost

The host-visible capacity of Jenga is 2/3 of that of Baseline. Although it is technically possible to increase Jenga's host-visible capacity by using even finer granularity redundancy (e.g., quarter-blocks), doing so would waste HBM's bandwidth. HBM uses 128 TSVs twice in one DRAM cycle to transfer 32B. Thus, transferring anything less than a half-block (32B) during that cycle is wasteful. Although we assume PW to have the same capacity cost as Jenga, we could easily increase PW's host-visible capacity by creating parity across more than two blocks. However, that would result in even higher performance overheads for writes and error recoveries, as blocks that share the same parity need to synchronize during writes and recoveries. Additionally, increasing the size of a parity group in PW decreases fault tolerance.

5.7 Conclusion on Jenga Coding

The goal of this chapter was to present a comprehensive fault tolerance scheme for 3D DRAM that would minimize the performance costs. Jenga achieves this goal by using a combination of a collocated SECDED code and a sub-block parity scheme to tolerate faults in bits, rows, banks, channels, and chips. Furthermore, due to its minimal overheads, Jenga could enable HBM to be used in computer systems that require both high bandwidth and high fault tolerance (e.g., autonomous driving and other embedded systems).

6. Conclusion

This work focused on advancing emerging memory technologies by addressing their technological drawbacks and enabling their usage as mainstream memory solutions in a wider scope of computer systems. Through this dissertation document, we showed that by approaching this problem from the viewpoints of a computer architect and a coding theorist we can find new ways to innovate and new opportunities for optimization. The integration of the two disciplines creates a unique research space where the physical characteristics of a technology, the interface of a memory system, and the mathematical infrastructure of coding theory are all utilized to design and optimize solutions.

Understanding the underlying physical properties that cause Flash cells to wear was a key part in designing MinWear codes. Additionally, by using shell mapping, a technique we borrow from coding theory, we were able to design efficient encoding and decoding algorithms for MinWear codes. Furthermore, we introduced a mathematical model that can quantify the lifetime of Flash memory under a coding scheme. We showed the importance of such a model as it allowed us to fairly compare how different endurance and non-endurance codes impact the lifetime of Flash memory.

Through our work in racetrack memory, we were able to utilize VT codes to address shift errors. Although VT codes are well understood in the theory community, they were practically ignored by architects. Understanding the system interface of the racetrack memory enabled us to also understand the limitations of VT codes in this

specific application, and the need to accommodate them with delimiter patterns to enable them to detect and correct double shift errors per codeword. Furthermore, we proposed a new construction algorithm that enabled easy and efficient decoding under the common, error free case.

In our final piece of work, we introduced Jenga, a fault tolerance scheme that enhanced the fault tolerance of 3D-stacked DRAM without sacrificing its performance. Rather than naively adding redundancy to increase reliability, we approached this problem with main goal to maintain the high memory bandwidth of 3D-stacked DRAM, a feature that makes this technology unique. Through our proposed spatial redundancy, we were able to increase fault tolerance with minimal overheads in the network's traffic and thus maintaining high performance even for memory intensive benchmarks.

Our work shows that this interdisciplinary approach is very promising and could be used in other research areas as well. One such example is reducing the noise due to electromagnetic interference between adjacent wires (a.k.a. crosstalk). Although typical constrained codes could be used to reduce crosstalk, designing solutions for specific communication channels (e.g., DDR, PCIe) could open new opportunities for optimizations. Furthermore, by considering that the power consumption of these channels depends on the bit values transmitted, codes could be designed to reduce the average power consumption by favoring certain bit patters (e.g., with bias codes).

As we reach toward the density limits of the current CMOS technology, and as the power wall becomes an ever more concerning limitation in computer systems, researchers need to find new ways to innovate. Breaching the boundaries between disciplines is a promising approach that generates new opportunities for innovation.

Appendix A

In this appendix we provide additional material and proofs of correctness on the GreenFlag codes presented in Chapter 4.4. We organize these proofs in tow sections. In Section A.1 we first prove that GreenFlag, as a VT code, can detect and correct any codeword that has a single shift error (insertion or deletion). In Section A.2 we prove that (with the help of delimiter bits) GreenFlag can detect double shift errors and correct them with read-replays.

A.1 *Single Shift Error Correction Proof*

Consider two binary strings $x = x_1, x_2, \dots, x_n$, and $y = y_1, y_2, \dots, y_n$ that are GreenFlag codewords with no errors. We denote $D_1(x)$ and $D_1(y)$ the set of all binary strings of x and y than can occur after as single deletion, respectively. We also denote $I_1(x)$ and $I_1(y)$ the set of all binary strings of x and y than can occur after a single insertion, respectively. In order to be able to detect and correct a single shift error then the following must be true:

$$\mathbf{D_1(x) \cap D_1(y) = \emptyset} \qquad \textbf{Equation 1}$$

$$\mathbf{I_1(x) \cap I_1(y) = \emptyset} \qquad \textbf{Equation 2}$$

Equation 1 is satisfied for all GreenFlag codewords as they are Varshamov-Tenengolts (VT) codewords [74] that can detect and correct single deletions. Now we

prove that all codewords that satisfy Equation 1 also satisfy Equation 2 and thus can detect and correct single insertions.

Proof. Suppose two binary strings x and y of length n satisfy Equation 1 but not Equation 2. Suppose inserting x′ in position i of x and y′ in position j of y results in the same bit sequence. Without loss of generality, we assume i ≤ j. This implies that deleting x_{j-1} and x_i from x and y respectively, would result in the same bit sequence. This contradicts Equation 1 and thus proving Equation 2 for all GreenFlag codewords.

The correction algorithm for single deletion errors that is used by GreenFlag is the same as the proposed algorithm by Levenshtein [*50*, *49*]. The mathematical analysis of this algorithm can be found in [*70*]. The correction algorithm for single insertion errors is a modified version of the deletion correction algorithm.

A.2 *Double Shift Error Correction Proof*

To correct double shift errors, we first need to be able to detect them with the help of delimiter bits as discussed in Chapter 4.4.2. Additionally, we need to be able to categorize shift errors to deletions, insertions or a mixture of both. This is achieved with the help of a 6-bit delimiter $d = d_1, d_2, d_3, d_4, d_5, d_6$ that is appended in front of each codeword.

We now extensively show how the extended GreenFlag codewords (VT codeword with delimiter bits) can be used to identify two deletions, or a deletion and an insertion. The case of two insertions can be derived with similar arguments.

1. Two deletions: There are three cases to consider as listed below.

1.1. Two deletions in the VT codeword: In this case, in positions d_1, d_2, d_3, and d_4 we observe 1000 and position d_5 corresponds to the first bit of the next extended codeword. In this case, we declare double deletion and we use read-replay to resolve this case.

1.2. Two deletions in the delimiter bits: For this scenario, the VT codeword is unaffected. However, we will declare either two deletions, one deletion, or no deletion depending on the delimiter bits that we observe in positions d_1, d_2, d_3, and d_4.

a) 1000: Declaring two deletions as shown by Table 4.3. Similar to the case when two deletions happen in the VT codeword we use read-replay to resolve this case.

b) 1100: Declaring one deletion as shown by Table 4.3. The single deletion correction algorithm is used that recovers the correct data (that are the same as the initial VT codeword).

c) 110X: Declaring no deletion. Since the VT codeword is not affected here, data can be reliably recovered. The position of the port is addressed when reading the next extended codeword

1.3. One deletion in the VT codeword and one deletion in the delimiter bits: For this case, in positions d_1, d_2, d_3, and d_4 we can observe two possibilities as described below and position d_5 corresponds to the first bit of the next extended codeword.

a) 1000: Similar to the case when two deletions happen in the VT codeword, we declare double deletion. The VT codeword only contains a single deletion but since we cannot determine this fact, we cannot trust the checksum. We use the read-replay in this case.

b) 1100: According to Table 4.3, we declare a single deletion. Since the VT codeword only contains a single shift error, data can be reliably recovered.

2. One deletion and one insertion: There are 3 cases:

2.1. Both the deletion and insertion occur in the VT codeword: In this case the delimiter bits will be correct, but the checksum will be non-zero and indicate an error. To prove this, consider any two codewords x and y that are VT codewords. We argue that y cannot be obtained from x after a single deletion followed by a single insertion. Thus, the checksum of x after such an error pattern will always be non-zero.

Proof. Suppose $x' \in D_1(x)$ and $y' \in D_1(y)$. If y can be obtained from x' after an insertion, then:

$$\mathbf{I_1(x')} \cap \mathbf{I_1(y')} \neq \emptyset \qquad \textbf{Equation 3}$$

In other words x' and y' do not satisfy Equation 2 which in turn implies that x and y do not satisfy Equation 1. This contradiction proves the result. A similar argument can be used when an insertion is followed by a deletion.

2.2. Deletion in the VT codeword and insertion in delimiter bits: In this case the delimiter bits d_1, d_2, d_3, d_4, and d_5 that we observe are:

a) 11000: According to Table 4.3, we declare a single deletion. Since the VT codeword only contains a single shift error, data can be reliably recovered.

b) 11100: According to Table 4.3, the checksum is non-zero and we detect 1 deletion and 1 insertion in the VT codeword. The read-replay is used to retrieve the correct codeword.

2.3. Deletion and insertion in delimiter bits: This can only result in two different observations for the delimiter bits d_1, d_2, d_3, d_4, and d_5**:**

a) 11100: No error is detected and the VT codeword is correct as not shift errors occurred there.

b) 11000: We declare a single deletion according to Table 4.3. We follow the deletion correction algorithm and we get the correct codeword that was the same as the initial VT codeword.

References

[1] "7-Zip: 7-Zip is a File Archiver with a High Compression Ratio," accessed: 2020-01-12. [Online]. Available: https://www.7-zip.org/

[2] "INTEL STRATIX 10 MX FPGAS," accessed: 2020-01-12. [Online]. Available: https://-www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html

[3] "JEDEC Updates Groundbreaking High Bandwidth Memory (HBM) Standard," accessed: 2020-01-12. [Online]. Available: https://www.jedec.org/news/-pressreleases/jedec-updates-groundbreaking-high-bandwidth-memory-hbm-standard-0

[4] "Memory Technology Evolution: An Overview of System Memory Technologies," accessed: 2020-01-12. [Online]. Available: https://support.hpe.com/hpsc/doc/public/-display?docId=emr_na-c01552458

[5] "NVIDIA Tesla P100: The World's First AI Supercomputing Data Center GPU," accessed: 2020-01-12. [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-p100/

[6] "RadeonTM RX Vega64 Graphics," accessed: 2020-01-12. [Online]. Available: https://-www.amd.com/en/products/graphics/radeon-rx-vega-64

[7] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving Predictable Performance Through Better Memory Controller Placement in Many-Core CMPs," *SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 451–461, 2009.

[8] K. Bakshi, "Considerations for Big Data: Architecture and Approach," in *Aerospace Conference*. IEEE, 2012, pp. 1–7.

[9] A. Bhatia, M. Qin, A. R. Iyengar, B. M. Kurkoski, and P. H. Siegel, "Lattice-based WOM Codes for Multilevel Flash Memories," *journal on Selected Areas in Communications*, vol. 32, no. 5, pp. 933–945, 2014.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 72–81.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 551–563.

[13] J. Carter and K. Rajamani, "Designing Energy-Efficient Servers and Data Centers," *Computer*, no. 7, pp. 76–78, 2010.

[14] L.-P. Chang, "On efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems," in *Proceedings of the Symposium on Applied Computing*. ACM, 2007, pp. 1126–1130.

[15] L.-P. Chang and C.-D. Du, "Design and Implementation of an Efficient Wear-Leveling Algorithm for Solid-State-Disk Microcontrollers," *Transactions on Design Automation of Electronic Systems*, vol. 15, no. 1, p. 6, 2009.

[16] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo, "Endurance Enhancement of Flash-Memory Storage, Systems: An Efficient Static Wear Leveling Design," in *Design Automation Conference*. IEEE, 2007, pp. 212–217.

[17] Y. M. Chee, H. M. Kiah, A. Vardy, E. Yaakobi *et al.*, "Coding for Racetrack Memories," *Transactions on Information Theory*, vol. 64, no. 11, pp. 7094–7112, 2018.

[18] B. Chen, X. Zhang, and Z. Wang, "Error Correction for Multi-Level NAND Flash Memory Using Reed-Solomon Codes," in *Workshop on Signal Processing Systems*. IEEE, 2008, pp. 94–99.

[19] H.-M. Chen, C.-J. Wu, T. Mudge, and C. Chakrabarti, "RATT-ECC: Rate Adaptive Two-Tiered Error Correction Codes for Reliable 3D Die-Stacked Memory," *Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 3, pp. 1–24, 2016.

[20] S. Cho and H. Lee, "Flip-N-Write: a Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance," in *Proceedings of the International Symposium on Microarchitecture*. ACM, 2009, pp. 347–357.

[21] W. J. M. Chua, K. Cai, and W. L. Goh, "Efficient Two-Write WOM-Codes for Non-Volatile Memories," *Communications Letters*, vol. 19, no. 10, pp. 1690–1693, 2015.

[22] E. Deal, "Trends in NAND Flash Memory Error Correction," *Cyclic Design*, vol. 6, no. 5, pp. 8–1, 2009.

[23] T. J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," *IBM Microelectronics Division*, vol. 11, pp. 1–23, 1997.

[24] G. Dong, N. Xie, and T. Zhang, "On the Use of Soft-Decision Error-Correction Codes in NAND Flash Memory," *Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 2, pp. 429–439, 2010.

[25] B. Euzent, N. Boruta, J. Lee, and C. Jenq, "Reliability Aspects of a Floating Gate E2 PROM," in *International Reliability Physics Symposium*. IEEE, 1981, pp. 11–16.

[26] A. Fert, V. Cros, and J. Sampaio, "Skyrmions on the Track," *Nature Nanotechnology*, vol. 8, no. 3, p. 152, 2013.

[27] G. D. Forney, "Coset Codes. I. Introduction and Geometrical Classification," *Transactions on Information Theory*, vol. 34, no. 5, pp. 1123–1151, 1988.

[28] G. D. Forney, "Coset Codes. II. Binary Lattices and Related Codes," *Transactions on Information Theory*, vol. 34, no. 5, pp. 1152–1187, 1988.

[29] D. Geer, "Chip Makers Turn to Multicore Processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[30] P. Gratz and S. W. Keckler, "Realistic Workload Characterization and Analysis for Networks-on-Chip Design," in *Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, 2010, pp. 1–10.

[31] R. Hesse, J. Nicholls, and N. E. Jerger, "Fine-Grained Bandwidth Adaptivity in Networks-on-Chip Using Bidirectional Channels," in *International Symposium on Networks-on-Chip*. IEEE, 2012, pp. 132–141.

[32] C.-C. Huang and V. Nagarajan, "Increasing Cache Capacity Via Critical-Words-Only Cache," in *International Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 125–132.

[33] Y. Huang, W. Kang, X. Zhang, Y. Zhou, and W. Zhao, "Magnetic Skyrmion-Based Synaptic Devices," *Nanotechnology*, vol. 28, no. 8, p. 08LT02, 2017.

[34] Y. P. Ivanov, A. Chuvilin, S. Lopatin, and J. Kosel, "Modulated Magnetic Nanowires for Controlling Domain Wall Motion: Toward 3D Magnetic Memories," *ACS NANO*, vol. 10, no. 5, pp. 5326–5332, 2016.

[35] P. Jacob, A. Zia, O. Erdogan, P. M. Belemjian, J.-W. Kim, M. Chu, R. P. Kraft, J. F. McDonald, and K. Bernstein, "Mitigating Memory Wall Effects in High-Clock-Rate and Multicore CMOS 3-D Processor Memory Stacks," *Proceedings of the IEEE*, vol. 97, no. 1, pp. 108–122, 2009.

[36] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin, "Writing Cosets of a Convolutional Code to Increase the Lifetime of Flash Memory," in *Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2012, pp. 308–318.

[37] A. Jagmohan, M. Franceschini, L. A. Lastras-Montaño, and J. Karidis, "Adaptive Endurance Coding for NAND Flash," in *Globecom Workshops*. IEEE, 2010, pp. 1841–1845.

[38] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *Symposium on VLSI Technology (VLSIT)*. IEEE, 2012, pp. 87–88.

[39] H. Jeon, G. H. Loh, and M. Annavaram, "Efficient RAS Support for Die-Stacked DRAM," in *International Test Conference*. IEEE, 2014, pp. 1–10.

[40] W. Jeong, J.-w. Im, D.-H. Kim, S.-W. Nam, D.-K. Shim, M.-H. Choi, H.-J. Yoon, D.-H. Kim, Y.-S. Kim, H.-W. Park *et al.*, "A 128 Gb 3b/cell V-NAND Flash Memory With 1 Gb/s I/O Rate," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 204–212, 2015.

[41] X. Jian, V. Sridharan, and R. Kumar, "Parity Helix: Efficient Protection for Single-Dimensional Faults in Multi-Dimensional Memory Systems," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 555–567.

[42] A. Jiang, V. Bohossian, and J. Bruck, "Floating Codes for Joint Information Storage in Write Asymmetric Memories," in *International Symposium on Information Theory*. IEEE, 2007, pp. 1166–1170.

[43] M. Jung, S. A. McKee, C. Sudarshan, C. Dropmann, C. Weis, and N. Wehn, "Driving into the Memory Wall: The Role of Memory for Advanced Driver Assistance Systems and Autonomous Driving," in *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 377–386.

[44] S. Kayser, E. Yaakobi, P. H. Siegel, A. Vardy, and J. K. Wolf, "Multiple-Write WOM-Codes," in *Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2010, pp. 1062–1068.

[45] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon, "Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy," in *International Symposium on Microarchitecture*. IEEE, 2012, pp. 376–388.

[46] Y. Kwon and M. Rhu, "Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 148–161.

[47] R. Laroia, N. Farvardin, and S. A. Tretter, "On Optimal Shaping of Multidimensional Constellations," *Transactions on Information Theory*, vol. 40, no. 4, pp. 1044–1056, 1994.

[48] J. C. Lee, J. Kim, K. W. Kim, Y. J. Ku, D. S. Kim, C. Jeong, T. S. Yun, H. Kim, H. S. Cho, S. Oh *et al.*, "High Bandwidth Memory (HBM) With TSV Technique," in *2016 International SoC Design Conference (ISOCC)*. IEEE, 2016, pp. 181–182.

[49] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," in *Soviet Physics Doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[50] V. Levenshtein, "Binary Codes Capable of Correcting Spurious Insertions and Deletions of Ones," *Problemy Peredachi Informatsii*, vol. 1, pp. 12–25, 1965.

[51] Y. Liu, P. Huang, and P. H. Siegel, "Performance of Optimal Data Shaping Codes," in *International Symposium on Information Theory*. IEEE, 2017, pp. 1003–1007.

[52] Y. Liu and P. H. Siegel, "Shaping Codes for Structured Data," in *Global Communications Conference*. IEEE, 2016, pp. 1–6.

[53] D. Locklear, "Chipkill Correct Memory Architecture," *Dell Enterprise Systems Group, Technology brief*, 2000.

[54] G. Mappouras, A. Vahid, R. Calderbank, and D. Sorin, "Extending Flash Lifetime in Embedded Processors by Expanding Analog Choice," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2462–2473, 2018.

[55] G. Mappouras, A. Vahid, R. Calderbank, and D. J. Sorin, "Methuselah Flash: Rewriting Codes for Extra Long Storage Lifetime," in *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 180–191.

[56] S. Mittal, "A Survey of Techniques for Architecting Processor Components Using Domain-Wall Memory," *Journal on Emerging Technologies in Computing Systems*, vol. 13, no. 2, p. 29, 2017.

[57] P. J. Nair, D. A. Roberts, and M. K. Qureshi, "Citadel: Efficiently Protecting Stacked Memory from TSV and Large Granularity Failures," *Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 4, pp. 1–24, 2016.

[58] S. Ollivier, D. Kline, R. Kawsher, R. Melhem, S. Banja, and A. K. Jones, "Leveraging Transverse Reads to Correct Alignment Faults in Domain Wall Memories," in *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 375–387.

[59] S. S. Parkin, M. Hayashi, and L. Thomas, "Magnetic Domain-Wall Racetrack Memory," *Science*, vol. 320, no. 5873, pp. 190–194, 2008.

[60] S. Parkin and S.-H. Yang, "Memory on the Racetrack," *Nature Nanotechnology*, vol. 10, no. 3, p. 195, 2015.

[61] S. S. Parkin, M. Hayashi, and L. Thomas, "Supporting Online Material for Magnetic Domain-Wall Racetrack Memory," accessed: 2020-01-12. [Online]. Available: https://-science.sciencemag.org/content/sci/suppl/2008/04/10/320.5873.190.DC1/-Parkin.SOM.pdf

[62] P. Pavan, R. Bez, P. Olivo, and E. Zanoni, "Flash Memory Cells-An Overview," *Proceedings of the IEEE*, vol. 85, no. 8, pp. 1248–1271, 1997.

[63] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines," in *International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 250–259.

[64] G. Rakshith, R. Rozario, M. Rhevanth, K. Nikitha, and B. R. Mohan, "Performance Analysis of Secondary Storage Media Through File Systems Benchmarking," in *International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE, 2019, pp. 1222–1226.

[65] R. L. Rivest and A. Shamir, "How to Reuse a "Write-Once" Memory," *Information and Control*, vol. 55, no. 1-3, pp. 1–19, 1982.

[66] C. L. Rumer and E. A. Zarbock, "Through Silicon Via, Folded Flex Microelectronic Package," Aug. 2 2005, US Patent 6,924,551.

[67] D. Sanz-Hernandez, R. F. Hamans, J.-W. Liao, A. Welbourne, R. Lavrijsen, and A. Fernandez-Pacheco, "Fabrication, Detection, and Operation of a Three-Dimensional Nanomagnetic Conduit," *ACS NANO*, vol. 11, no. 11, pp. 11066–11073, 2017.

[68] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2009.

[69] E. Sharon, S. Achtenberg, I. Alrod, A. Klein, and A. Eyal, "Data Shaping for Improving Endurance and Reliability in Sub-20nm NAND," *Flash Memory Summit, Santa Clara, CA*, 2014.

[70] N. J. Sloane, "On Single-Deletion-Correcting Codes," *Codes and designs*, vol. 10, pp. 273–291, 2000.

[71] A. S. Spinelli, C. M. Compagnoni, and A. L. Lacaita, "Reliability of NAND Flash Memories: Planar Cells and Emerging Issues in 3D Devices," *Computers*, vol. 6, no. 2, p. 16, 2017.

[72] Z. Sun, W. Wu, and H. Li, "Cross-Layer Racetrack Memory Design for Ultra High Density and Low Power Consumption," in *Design Automation Conference*. IEEE, 2013, pp. 1–6.

[73] R. Tomasello, E. Martinez, R. Zivieri, L. Torres, M. Carpentieri, and G. Finocchio, "A Strategy for the Design of Skyrmion Racetrack Memories," *Scientific reports*, vol. 4, p. 6784, 2014.

[74] R. Varshamov and G. Tenengolts, "Codes Which Correct Single Asymmetric Errors," *Automatika i Telemkhanika*, vol. 161, no. 3, pp. 288–292, 1965.

[75] R. Venkatesan, V. Kozhikkottu, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, "TapeCache: A High Density, Energy Efficient Cache Based on Domain Wall Memory," in *Proceedings of the International Symposium on Low Power Electronics and Design*. ACM, 2012, pp. 185–190.

[76] R. Venkatesan, S. G. Ramasubramanian, S. Venkataramani, K. Roy, and A. Raghunathan, "Stag: Spintronic-Tape Architecture for GPGPU Cache Hierarchies," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 253–264.

[77] X. Wang, C. Zhang, X. Zhang, and G. Sun, "np-ECC: Nonadjacent Position Error Correction Code for Racetrack Memory," in *International Symposium on Nanoscale Architectures*. IEEE, 2016, pp. 23–24.

[78] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter, "Architecting for Power Management: The IBM POWER7â„¢ approach," in

International Symposium on High-Performance Computer Architecture. IEEE, 2010, pp. 1–11.

[79] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. Wolf, "Codes for Write-Once Memories," *Transactions on Information Theory*, vol. 58, no. 9, pp. 5985–5999, 2012.

[80] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf, "Efficient Two-Write WOM-Codes," in *Information Theory Workshop*. IEEE, 2010, pp. 1–5.

[81] E. Yaakobi, G. Yadgar, N. Bundak, and L. Gilon, "A Case for Biased Programming in Flash," in *Workshop on Hot Topics in Storage and File Systems*. Boston, MA: USENIX Association, Jul. 2018. [Online]. Available: https://www.usenix.org/conference/-hotstorage18/presentation/yaakobi

[82] G. Yadgar, E. Yaakobi, and A. Schuster, "Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes," in *Conference on File and Storage Technologies*, 2015, pp. 257–271.

[83] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The Dynamic Granularity Memory System," in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 548–560.

[84] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative Modeling of Racetrack Memory, a Rradeoff Among Area, Performance, and Power," in *the 20th Asia and South Pacific Design Automation Conference*. IEEE, 2015, pp. 100–105.

[85] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu, "Hi-fi Playback: Tolerating Position Errors in Shift Operations of Racetrack Memory," in *SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 694–706.

[86] W. Zhao, Y. Zhang, H. Trinh, J. Klein, C. Chappert, R. Mantovan, A. Lamperti, R. Cowburn, T. Trypiniotis, M. Klaui *et al.*, "Magnetic Domain-Wall Racetrack Memory for High Density and Fast Data Storage," in *International Conference on Solid-State and Integrated Circuit Technology*. IEEE, 2012, pp. 1–4.

[87] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. Russell, W. Y. Wang, L. B. Freeman, P. Hosier *et al.*, "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)," *IBM Journal of Research and Development*, vol. 40, no. 1, pp. 3–18, 1996.

Biography

Georgios Mappouras was born on January 21st, 1990 in Holargos, Greece and grew up in Larnaca, Cyprus where he graduated high school and served as a subordinate officer in the Cypriot National Guard for two years. In 2009, and after the end of his military service, he moved to Athens, Greece where he started his undergraduate studies at the National Technical University of Athens, receiving his bachelor's degree in Electrical and Computer Engineering (ECE) in 2014. In the same year, he moved to Durham, NC, U.S. where he started his graduate studies at Duke University and was advised by Prof. Daniel J. Sorin. During his graduate studies, Georgios was the lead author of five paper publications and a co-author on an additional three paper publications. His research focused on integrating computer architecture and coding theory to advance emerging memory technologies. In 2015, he received his M.S. degree in ECE from Duke University. In 2018, and while still a graduate student, he spent eight months in Santa Clara, CA working as an intern in Advanced Micro Devices where he authored three patents. In 2020, Georgios received his Ph.D. degree in ECE from Duke University.