Coset Coding to Extend the Lifetime of Non-Volatile Memory

by

Adam Jacobvitz

Department of Electrical and Computer Engineering
Duke University

Date:________________________

Approved:

____

Daniel Sorin, Supervisor

____

Robert Calderbank

____

Jeff Chase

____

Andrew Hilton

____

Benjamin 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

2014

ABSTRACT

Coset Coding to Extend the Lifetime of Non-Volatile Memory

by

Adam Jacobvitz

Department of Electrical and Computer Engineering
Duke University

Date:_______________________

Approved:

____

Daniel Sorin, Supervisor

____

Robert Calderbank

____

Jeff Chase

____

Andrew Hilton

____

Benjamin 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

2014

Abstract

Modern computing systems are increasingly integrating both Phase Change Memory (PCM) and Flash memory technologies into computer systems being developed today, yet the lifetime of these technologies is limited by the number of times cells are written. Due to their limited lifetime, PCM and Flash may wear-out before other parts of the system. The objective of this dissertation is to increase the lifetime of memory locations composed of either PCM or Flash cells using coset coding.

For PCM, we extend memory lifetime by using coset coding to reduce the number of bit-flips per write compared to un-coded writes. Flash program/erase operation cycle degrades page lifetime; we extend the lifetime of Flash memory cells by using coset coding to re-program a page multiple times without erasing. We then show how coset coding can be integrated into Flash solid state drives.

We ran simulations to evaluate the effectiveness of using coset coding to extend PCM and Flash lifetime. We simulated writes to PCM and found that in our simulations coset coding can be used to increase PCM lifetime by up to 3x over writing un-coded data directly to the memory location. We extended the lifetime of Flash using coset coding to re-write pages without an intervening erase and were able to re-write a single Flash page using coset coding more times than when writing un-coded data or using prior coding work for the same area overhead. We also found in our simulations that using coset coding in a Flash SSD results in higher lifetime for a given area overhead compared to un-coded writes.

Dedication

To my family, mentors, and friends.

Table of Contents

List of Tables

List of Figures

Acknowledgements

This research was supported in part by the National Science Foundation under grant 111-5367 and through XSEDE resources provided by the XSEDE Science Gateways program.

1. Introduction

Memory is used to store data in computing systems. Memory is organized into fixed sized locations composed of *memory cells*—hardware elements that store one or more bits. For example, Dynamic Random-Access Memory (DRAM) [59] cells consist of a single transistor. Some new next-generation memory technologies produced to replace and/or augment existing memory technologies have cells that are *write-limited*, i.e., they fail after a certain number of writes. When a single memory cell fails in a memory location, the entire memory location becomes unusable[1]. This dissertation introduces methods to increase the lifetime of memory locations composed of write-limited memory cells.

In this introduction, we introduce a number of memory technologies and indicate which memories are write-limited. Section 1.1 lists the different memory technologies, their status, and whether they are write-limited. Section 1.2 presents the goals and contributions of this dissertation. Section 1.3 presents how this dissertation is organized and the contributions of each chapter.

1.1 Memory Technologies

Modern computing systems incorporate many different memory technologies. Table 1 lists memory technologies currently in-use and next-generation memory technologies that are being developed for new computing systems. Unshaded rows are memories used in current production systems, and shaded rows are memories either in development or in the process of being integrated into production systems. Table 1 also lists which memory technologies are write-limited and which memory technologies are in mass production. In this dissertation, we

[1] Techniques have been developed to tolerate failed cells in a memory location [25][54]

focus on improving the lifetime of memory locations composed of Phase Change Memory (PCM) cells and Flash cells. Both PCM and Flash are write-limited and are being mass-produced. Flash is currently in the process of being integrated into new and existing computing systems ranging from cell phones [62] to large scale datacenters [5] and super-computers [57]. PCM has started to enter mass production, but has not yet been integrated into new and existing computing systems.

Table 1: Memory Technology Properties

Memory Technology	Write-Limited	Mass Produced	Highest Level Used
SRAM	No	Yes	Cache
Memristors [54]	Yes	No	Cache
eDRAM [19]	No	Yes	Cache
DRAM	No	Yes	Main Memory
PCM [68]	Yes	Limited	Main Memory
STT-RAM [29]	No	No	Main Memory
ReRAM [5]	Yes	No	Main Memory
FeRAM [7]	No	No	Main Memory
RRAM [73]	No	No	Main Memory
Flash [44]	Yes	Yes	Secondary Storage
Hard Drives	No	Yes	Secondary Storage
Optical	No	Yes	Archival Storage
Tape	No	Yes	Archival Storage

1.2 *Goals and Contributions*

This dissertation introduces methods to extend lifetime of memory locations composed of write-limited memory cells. We focus on extending the lifetime of memory locations composed of either PCM cells or Flash cells, both of which are write-limited. We propose methods to extend the lifetime of memory locations with either PCM or Flash cells by incorporating coding techniques during the write process.

This dissertation makes three main contributions:

- We extend the lifetime of write-limited memories by modifying a coding technique used in digital communications to shape the signals input to a Gaussian channel. We apply this technique to memory composed of both Flash and PCM cells respectively.
- We develop enhancements to our method that extend lifetime and allow writes even when the stored value of a subset of the memory cells cannot be changed.
- We provide implementations of an encoder and a decoder of our coding technique for both PCM and Flash.

1.3 Outline

This dissertation has five chapters including the introduction (Chapter 1). Chapter 2 provides background on the coding technique we use. Chapter 3 discusses how we extend the lifetime of memory locations composed of PCM cells using coding. Chapter 4 discusses how we extend the lifetime of memory locations composed of Flash cells using coding. Chapter 5 concludes the dissertation and provides insight into future research directions.

2. Coset Coding

This section introduces coset coding [18][19] and our use of it to create a technique to extend the lifetime of memory locations composed of PCM (Chapter 3) and Flash memory cells (Chapter 4). Current memory writing techniques have a one-to-one map between a given *dataword* (the data the host wants to write) and a given *codeword* (the data that is written). In contrast, coset coding has a one-to-many map between a dataword and a set of codewords. Each codeword in the set can be mapped back to the dataword. Coset coding enables the selection of a representative codeword to write from a number of possibilities. We use the flexibility of the coset coding technique to write many possible codewords to minimize wear on a given memory location.

Coset coding was first introduced in the context of communications [11][18][19] to reduce average transmitted signal power in voiceband modems. When used in voiceband modems, all vectors in a given coset represent the same information and the vector sent over the communication line is chosen to minimize signal power.

The following sections provide background information on the different mechanisms we used in our implementation of coset coding. Section 2.1 introduces coset coding terminology. Section 2.2 discusses the steps required to read and write to memory. Sections 2.3 and 2.4 discuss how we implemented the steps.

2.1 Coset Code Terminology

We use the following coset coding terminology throughout our discussion. Coset coding is based on the creation of *cosets*—sets with special properties we use to represent a single dataword with multiple codewords. A codeword inside a coset that represents the coset is a *coset representative. Encoding* is the process of mapping from a dataword to the corresponding *coset label* (the coset representative that has a one-to-one mapping with the dataword)*. Decoding* is the process of mapping from a coset representative back to the corresponding dataword. Encoding occurs during the write process, and decoding occurs during the read process.

2.2 Coset Coding Encoding and Decoding Steps

This section presents the steps we used in our coset coding technique to encode using coset coding for writing codewords to memory and to decode using coset coding for reading codewords from memory. We implemented coset codes both without error correction (for PCM) and with error correction (for Flash). Section 2.2.1 discusses the process for encoding. This process is the same for coset coding both with and without error correction. Section 2.2.2 discusses the processes for decoding both with and without error correction, respectively.

2.2.1 Encoding Steps

The coset coding encode process consists of mapping from a dataword to the coset representative that, when written, has the least impact on memory location lifetime. The coset coding encode process consists of the two steps as illustrated in Figure 1. The first step is mapping from a dataword to a coset. The second step is selecting the coset representative to write. A coset contains multiple write options all of which represent the original dataword. After encoding, we write the selected coset representative to memory. We present our implementation of step 1 and step 2, respectively, in Sections 2.3 and 2.4.

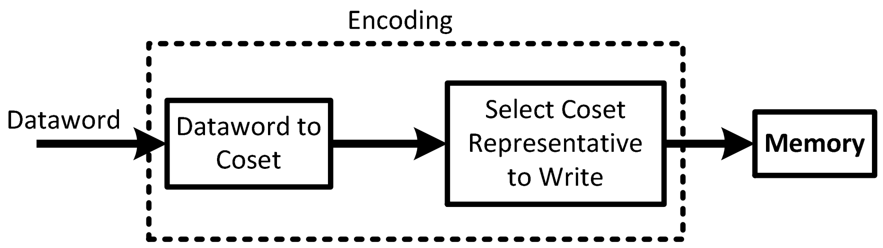


Figure 1: Coset Code Encoding Process

2.2.2 Decoding Steps

Figure 2 shows the coset code decoding process. The top of the figure shows how we decode coset codes without error correction (a single step process). The bottom of the figure shows how we decode coset codes with error correction (a two-step process). To decode, we map from the read-out coset representative back to the corresponding dataword by correcting any errors in the coset representative and then mapping back to the corresponding dataword.

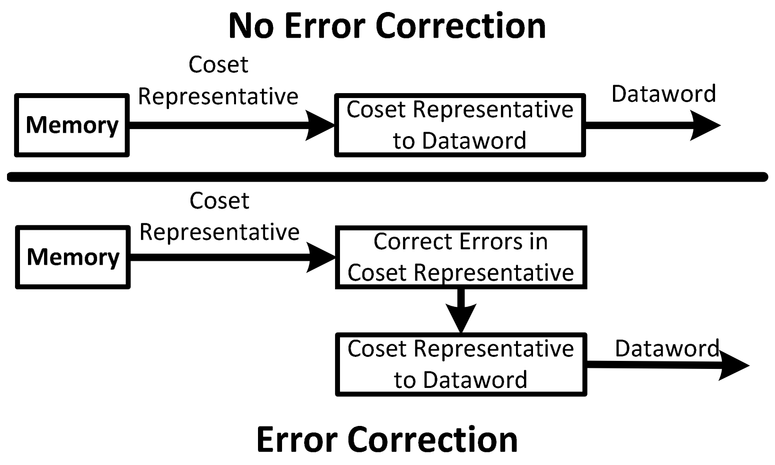


Figure 2: Coset Coding Decoding Process Both Without (top) and With (bottom) Error Correction

2.3 Mapping Between Dataword and Coset

In this dissertation, we use matrices to map between datawords and cosets. We describe how we create coset coding matrices in general in Section 2.3.1 and how to construct coset coding matrices that can also perform error correction in Section 2.3.2.

2.3.1 Coset Code Matrix Construction

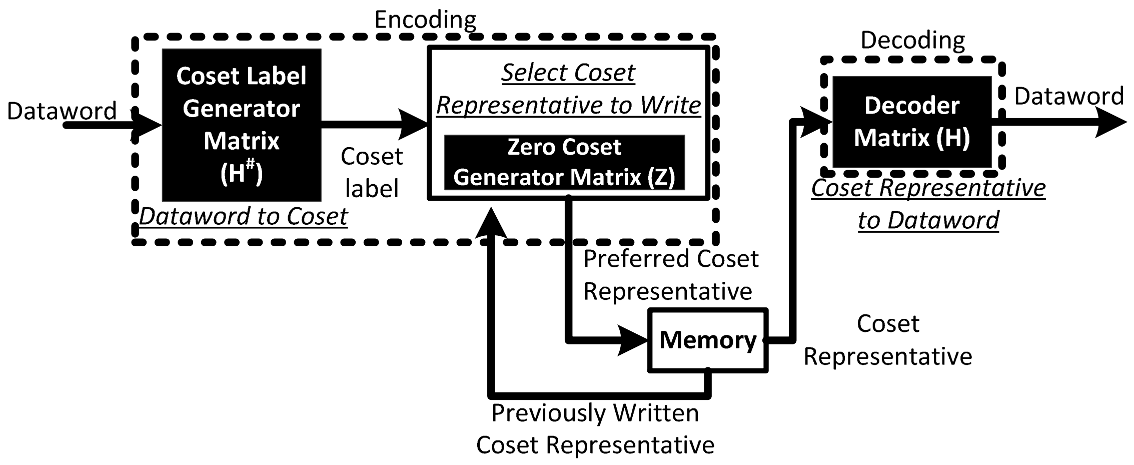
Table 2 lists the three matrices we used and their function in the coset code encoding and decoding processes. We named each matrix based on the matrix's function. We first create the *Zero Coset Generator Matrix* (**Z**) that we used to generate the zero coset representatives (the zero coset is defined as the coset that maps to the all zero dataword [17]) (Section 2.3.1.1). From **Z,** we are able to create the *Decoder Matrix* (**H**), which is used to map from the read-out coset representative to the corresponding dataword, and the *Coset Label Generator Matrix* ($\mathbf{H}^{\#}$), which maps between the dataword and the coset label (Section 2.3.1.2).

Table 2: Matrices Used as Part of Our Coset Coding Encoding/Decoding Implementation

Process Matrix is Used	Matrices	Purpose
Encoding	Zero Coset Generator Matrix (**Z**)	To generate the zero coset representatives
	Coset Label Generator Matrix ($\mathbf{H}^{\#}$)	To map from the dataword to the coset label
Decoding	Decoder Matrix (**H**)	To map from the read-out coset representative to the corresponding dataword.

Figure 3 shows the location of each matrix in the encoding and decoding processes. Each step is labeled in italics and underlined, and each matrix is shown in white text on a black background.

Both the "Dataword to Coset" and "Coset Representative to Dataword" steps consist of the **H**$^{\#}$ and **H** matrices, respectively; and the "Select Coset Representative to Write" step uses **Z**.



Encoding
Dataword
Coset Label Generator Matrix (H#)
Dataword to Coset
Coset label
Select Coset Representative to Write
Zero Coset Generator Matrix (Z)
Preferred Coset Representative
Memory
Previously Written Coset Representative
Coset Representative
Decoding
Decoder Matrix (H)
Coset Representative to Dataword
Dataword

Figure 3: Coset Coding Encode and Decode Processes

2.3.1.1 The Zero Coset Generator Matrix (Z)

In our experiments, we obtained the zero coset generator matrix (**Z**) from binary block and convolutional codes that have been shown to be well suited for coset coding [18]. All the other matrices in a given coset code (listed in Table 2) can be generated from the **Z** matrix. We evaluated coset coding for PCM using the Reed-Muller [52] and repetition binary block codes. We evaluated coset coding for Flash using convolutional codes [38].

We use the zero coset as part of our processes for searching for the best coset representative to write from the input dataword coset. We discuss how we use the zero coset with exhaustive search in Section 2.4.3 and with Viterbi search in Section 2.4.4.

2.3.1.2 The Decoder Matrix (H) and Coset Label Generator Matrix ($H^{\#}$)

We used **Z** to generate the decoder matrix (**H**). **H** maps from a coset representative back to a dataword as part of the decoding process. Generating **H** requires meeting the following conditions:

> **Condition 1:** *$ZH^T = 0$, where **0** is the matrix of all zeroes, and where "T" is the transpose operator* [45]. This condition ensures that all coset representatives in a given coset will map back to the same dataword.
>
> **Condition 2:** *All rows of **H** must be linearly independent* [75] *over GF(2)* [66]. For mathematical operations with coset codes, we use *GF(2),* a mathematical structure where values are calculated modulo 2. This condition ensures that all coset representatives multiplied by **H** will produce a unique dataword.
>
> **Condition 3:** *$H^{\#}H = I$, where "**I**" is the identity matrix* [67]. This condition ensures that the dataword mapped to the coset representative during encoding is the same dataword that is recovered during decoding.

The process of creating **H** is specific to the code used. For example, we evaluated extending the lifetime of PCM using Reed-Muller codes as coset codes. We generated **H** for a Reed-Muller coding using the method presented in Lin and Costello Jr. [38] as this **H** generation method ensured meeting both conditions 1 and 2. For the repetition code, we decoded using the method presented in Cho et al. [14].

To create $\mathbf{H}^{\#}$ and meet condition 3 for **H** ($\mathbf{H}^{\#}\mathbf{H} = \mathbf{I}$), we generated $\mathbf{H}^{\#}$ from **H** by taking the left pseudo-inverse [44] of **H** over GF(2).

2.3.2 Error Correcting Coset Code (ECCC) Matrix Construction

The process we use for building ECCC matrices builds on the process used to create coset code matrices as discussed in Section 2.3.1. ECCC matrices are still used to perform coset coding but they produce coset reprsentatives that can be checked for errors. To check for errors, an ECCC incorporates an error correcting code (ECC). Previously developed ECCs include Hamming codes [25].

Figure 4 shows the ECCC encoding and decoding process. Each step in each respective process is labeled in italics and underlined. Each matrix is shown in white text on a black background. The ECCC encoding step is identical to the non-error correcting coset encoding process shown in Figure 3, where **H**$^{\#}$ converts the dataword into a coset, and **Z** generates zero coset representatives that are used as part of the coset representative selection process (described in Section 2.4). We add a fourth *Parity Check Matrix* (**H'**) in the decoding process to check a coset representative produced by an ECCC matrix for errors and to provide repair information. During decoding, **H'** checks the coset representative for errors, the decoder corrects as many errors as possible, and **H** converts the coset representative to the corresponding dataword.

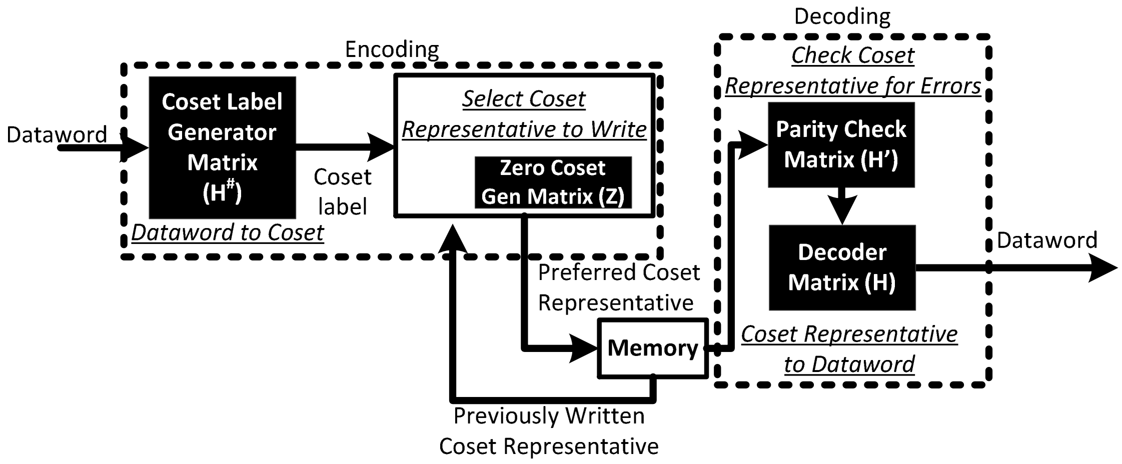

Encoding
Dataword
Coset Label Generator Matrix (H#)
Dataword to Coset
Coset label
Select Coset Representative to Write
Zero Coset Gen Matrix (Z)
Preferred Coset Representative
Memory
Previously Written Coset Representative
Decoding
Check Coset Representative for Errors
Parity Check Matrix (H')
Decoder Matrix (H)
Coset Representative to Dataword
Dataword

Figure 4: ECCC Encoding and Decoding Proceses

ECCC matrices meet the requirements of both an ECC and a coset code. A single matrix can fit the requirements of multiple codes. Since both **Z** and **H**$^{\#}$ the requirements of a coset code and of an ECC, coset representatives generated from them are also ECC codewords that can be checked for errors. Section 2.3.2.1 discusses the requirements for the ECC matrices. Section 2.3.2.2 discusses how we created **Z**, **H**$^{\#}$ and **H** so that they meet the requirements of both the ECC we used and the coset code we used.

2.3.2.1 ECC Matrices

The ECCs we used are defined by three matrices: *the generator matrix* (**G**), *the ECC Decoder Matrix* (**G**$^{\#}$), and *the parity check matrix* (**H'**) [38]. Figure 5 shows how each matrix is used during the ECC encoding/decoding processes. The ECC encoding/decoding process differs from the ECCC encoding/decoding process (shown for illustrative purposes only in Figure 5). During the encoding process, **G** maps the dataword to the corresponding ECC codeword that is then written to memory. During the two-step decode process, the parity check matrix (**H'**) checks the read-out ECC codeword for errors, corrects them, and **G**$^{\#}$ is then used to convert the corrected ECC codeword back into the corresponding dataword.

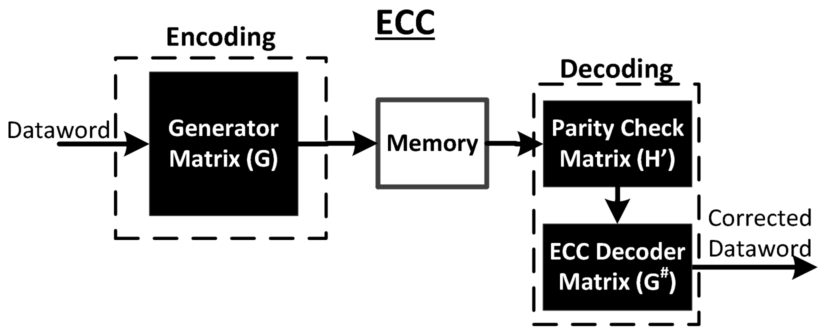


Figure 5: Encoding/Decoding ECC

Requirements for the Generator Matrix (G). In Chapter 4, we use a Hamming code as our ECC. Creating a Hamming ECC **G** matrix requires that two conditions are met [42]:

> **Condition 1.** *All rows in **G** are linearly independent over GF(2).*
>
> **Condition 2.** *All columns in **G** are different*

Requirements for the ECC Decoder Matrix ($\mathbf{G}^{\#}$). $\mathbf{G}^{\#}$ is the pseudo-inverse of **G** over GF(2) [28].

Generating the Parity Check Matrix (H'). A Hamming parity check matrix **H'** requires that $\mathbf{GH'}^{\mathbf{T}} = \mathbf{0}$, where "T" is the transpose operator [45]. We used a Hamming code for ECC in Chapter 4 and used the method presented in Lin and Costello Jr. [42] to generate **H'**.

2.3.2.2 Z (Zero Coset Generator Matrix), H# (Coset Generator Label Matrix), and H (Decoder Matrix) for Error Correction

This section presents our construction of the ECCC matrices that do not have an ECC. These matrices are **Z** (Zero Coset Generator Matrix), $\mathbf{H}^{\#}$ (Coset Generator Label Matrix), and **H** (Decoder Matrix).

Generating the Zero Coset Generator Matrix (Z) and the Coset Label Generator Matrix ($\mathbf{H}^{\#}$). **G** is the combination of **Z** and $\mathbf{H}^{\#}$. If **Z** and $\mathbf{H}^{\#}$ combine to form **G** then all coset representatives are also ECC codewords. Combining **Z** and $\mathbf{H}^{\#}$ to form **G** requires meeting the following two conditions in addition to those listed in Section 2.3.2.1**:**

> **Condition 1.** *The number of columns in $\mathbf{H}^{\#}$, **Z** and **G** must be the same*. If the number of columns were different, then **G**, $\mathbf{H}^{\#}$, and **Z** would produce different length codewords. For **Z** to be within **G**, the generated codeword length of both matrices and the number of columns of both matrices must be the same.
>
> **Condition 2.** *$\mathbf{H}^{\#}$ and **Z** vertically concatenated must form **G**.*

In our experiments, we selected **Z** from a previously developed coset code that has requirements compatible with **G**. We used a **Z** where all columns differ to ensure that all columns in **G** differ (Since **Z** and $\mathbf{H}^{\#}$ are vertically concatenated, all of the columns in **G** by definition are different). We generated $\mathbf{H}^{\#}$ based on the requirements listed in Section 2.3.1.2 and the requirements for **G**. We made the number of columns in $\mathbf{H}^{\#}$ the same as **Z** to satisfy Condition 1 for $\mathbf{H}^{\#}$. To satisfy condition 2, we randomly generated linearly independent rows for $\mathbf{H}^{\#}$ over GF(2) until **Z** and $\mathbf{H}^{\#}$ had sufficient rows to form **G**.

Generating the Decoder Matrix (H). Since coset representatives are also ECC codewords, the ECC Decoder Matrix ($\mathbf{G}^{\#}$) is the same as **H**. To generate the decoder matrix **H,** we took the pseudo-inverse of **G** over GF(2) [44]. Since both **Z** and $\mathbf{H}^{\#}$ combine to form **G**, and $\mathbf{H}^{\#}$ produces a coset representative, then **H** as the pseudo-inverse of **G** produces the corresponding dataword.

ECCC matrices can be used solely for error correction as presented in Section 2.3.2.1. Figure 6 shows the ECC encode/decode processes using the ECCC matrices. The ECC matrix names are shown in black text and the ECCC matrices are shown in white text on a black background. The generator matrix **G** is composed of $\mathbf{H}^{\#}$ and **Z**. Any dataword multiplied by the combination of these two matrices produces a coset representative that is also an ECC codeword. **H'** is used to check the ECC codeword/coset representative for errors. Finally, $\mathbf{G}^{\#}$ (a.k.a **H)** is used to decode the ECC codeword/coset representative back into the corrected dataword.

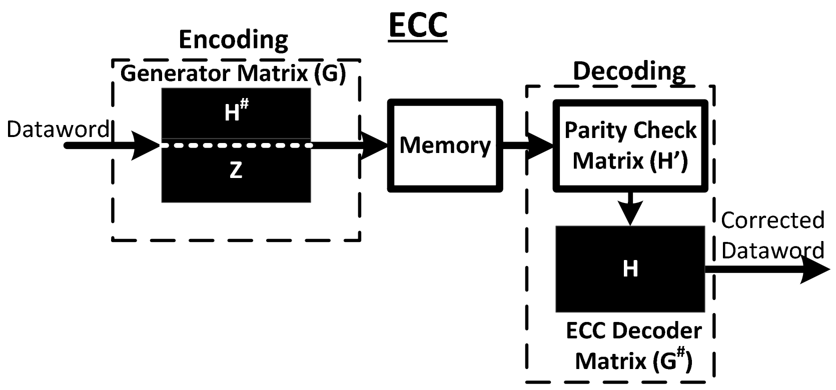


Figure 6: ECC Encoding/Decoding Process with ECCC Matrices

2.4 *Selecting a Coset Representative to Write from a Coset*

The coset rep selector selects a coset representative from a coset to write as part of the coset code encoding process. Figure 7 shows the location in the encoding process of the coset rep selector. Given a coset label and the previously written coset representative, the coset rep selector picks a coset representative from the *dataword coset*—the coset that represents the dataword—to write to memory.

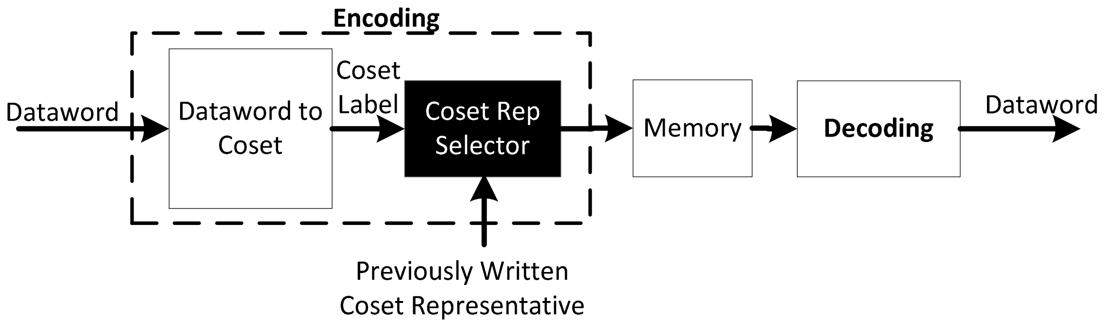


Figure 7: Coset Rep Selector Highlighted in the Coset Coding Encoding/Decoding Process

The coset rep selector requires the following four pieces of information: 1) the zero coset, 2) the *translate coset*—a coset that provides information on how writing the given coset will change a the value in a given memory location (Section 2.4.1); 3) a *metric function*—a function that provides a lexicographic ordering to a set (Section 2.4.2); and 4) a *search function*—an algorithm for applying a metric function (Section 2.4.3, exhaustive search and Section 2.4.4, the Viterbi algorithm). The search function determines how the coset rep selector uses each of the four components listed above to find a coset representative to write.

2.4.1 Translate Coset

This section discusses the translate coset and its part in the coset code encoding process. A translate coset represents how writing a given coset will change the value in a given memory location. The coset rep selector uses the translate coset to select a coset representative to write.

Figure 8 shows the different types of data needed to form a translate coset. Forming a given translate coset requires both the dataword coset and the previously written coset representative. The coset representatives in this example come from the extended Hamming (8,4) coset code (shown for illustrative purposes only in Figure 8). The three vectors in the memory box in the figure have been written to Flash memory in addresses “A”, “B” and “C”. The right side of the figure shows the dataword coset to be written.

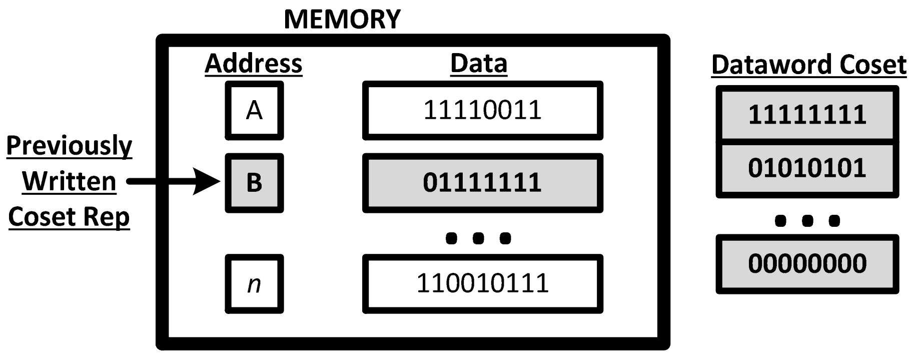


Figure 8: Example Setup for Forming a Translate Coset

Creating a translate coset consists of XORing the coset representatives in the dataword coset with the previously written coset representative. Figure 9 shows the formation of a translate coset representative. The encoder XORs the previously written data (01111111) and a coset representative from the dataword coset (11111111) to produce the translate coset representative 10000000. For translate coset representatives, a "1" indicates a bit will be flipped during the write while a "0" indicates a bit will not be flipped. For example, writing the dataword coset representative corresponding to the translate coset representative 10000000 will flip the 1st bit from the left. Performing this operation for all of the dataword coset representatives forms the translate coset.

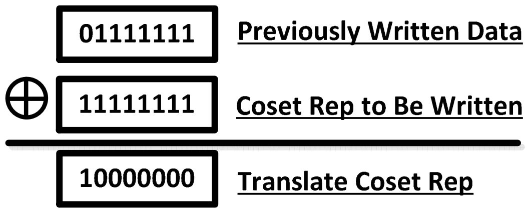


Figure 9: Formation of a Translate Coset Representative

2.4.2 The Metric Function

A metric function assigns a single numerical value to a coset representative. These values are used to order representatives within a coset. An exemplary metric function is counting the number of 1s in a coset representative. For example, if we had the following coset representatives, {0000,0001,0010,0011}, applying this metric function results in metric values {0,1,1,2}. If we wanted to pick coset representatives with the lowest metric value to write, we would pick 0000 because it has metric value 0. Other metric functions have been proposed in prior work [16].

2.4.3 Exhaustive Search

Exhaustive search consists of *enumerating*—finding all of the elements in a given set—each coset and then choosing the best coset representative. Figure 10 shows the four steps in an exhaustive search-based coset rep selector which are also listed below:

1. Enumerating the zero coset;
2. Finding a translate coset representative;
3. Enumerating the translate coset; and
4. Using a metric function to select a coset representative.

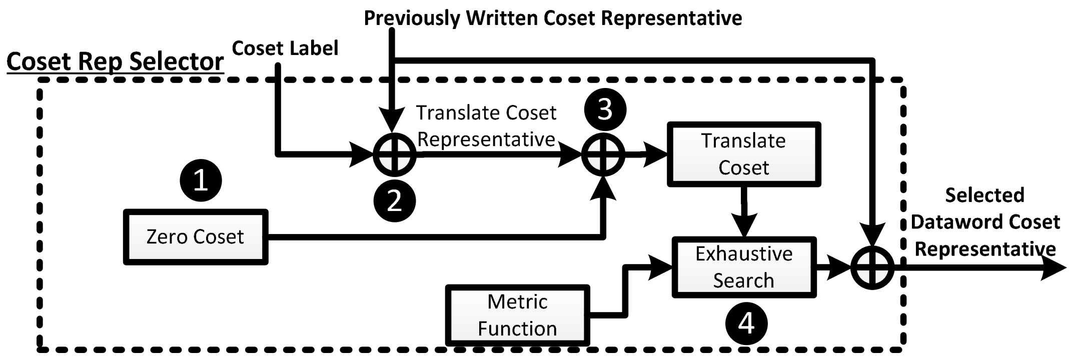

Previously Written Coset Representative
Coset Rep Selector
Coset Label
Translate Coset Representative
3
Translate Coset
1
2
Zero Coset
Selected Dataword Coset Representative
Exhaustive Search
Metric Function
4

Figure 10: Exhaustive Search-Based Coset Rep Selector

Step 1: Enumerate the Zero Coset (Figure 11). We first enumerate the coset representatives in the zero coset. To obtain the zero coset, we multiply all possible inputs by **G.** The results of these multiplications are the zero coset representatives.

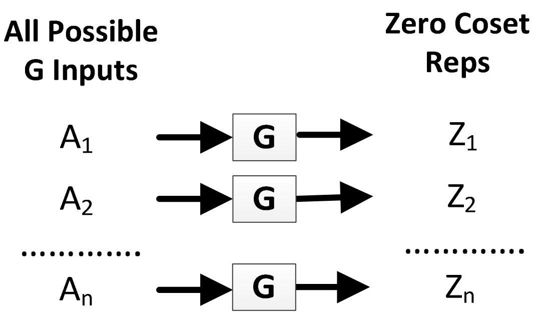


Figure 11: Enumerating the Zero Coset

Step 2: Find a Translate Coset Representative (Figure 12). We find coset representative in the translate coset by XORing the coset label and the previously written data.

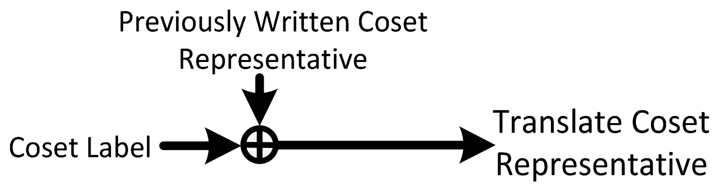

Previously Written Coset Representative
Coset Label
Translate Coset Representative

Figure 12: Finding a Translate Coset Representative

Step 3: Enumerate the Translate Coset (Figure 13). We enumerate the translate coset using the dataword coset by XORing the coset representatives in the zero coset with a translate coset representative. This process produces all the translate coset representatives.

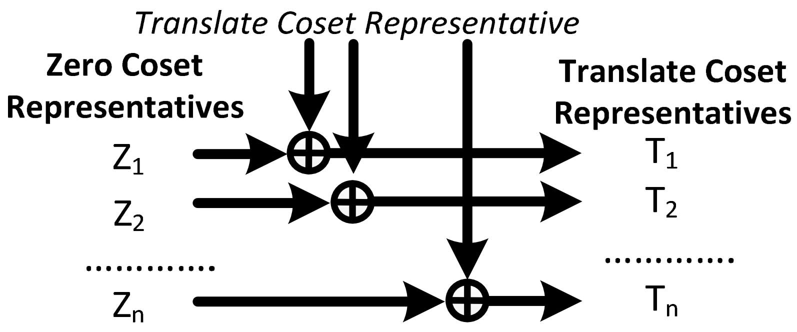

Translate Coset Representative
Zero Coset Representatives
Translate Coset Representatives
Z1
Z2
Zn
T1
T2
Tn

Figure 13: Enumerating the Translate Coset

Step 4: Use a Metric Function to Select a Coset Representative. We use a metric function on the translate coset to find the coset representative to write in the dataword coset. Exhaustive search applies the metric function to translate coset representatives to generate a corresponding metric value for each representative. These metrics are then used to select a translate coset representative.

2.4.4 The Viterbi Algorithm

As an alternative to exhaustive search, we can design the coset rep selector using the Viterbi algorithm (Viterbi) [20]. Viterbi is well suited for searching cosets with a large number of coset representatives such as with the codes presented in Chapter 4. For large coset sizes, Viterbi requires less power and less area than exhaustive search. For small coset sizes, exhaustive search may be preferable due to its lower base overheads.

Figure 14 shows how we can use Viterbi as part of the coset rep selector. Inputs and outputs are shown in bold, and components that are built into the coset rep selector are shown with white boxes with black text. The two inputs to the coset rep selector are used to form the translate coset representative. The metric function, translate coset representative, and zero coset are fed into the Viterbi algorithm which produces a zero coset representative. To convert back to a dataword coset representative the zero coset representative is first XORed with the translate coset representative and then XORed with the previously written coset representative to produce the selected dataword coset representative.

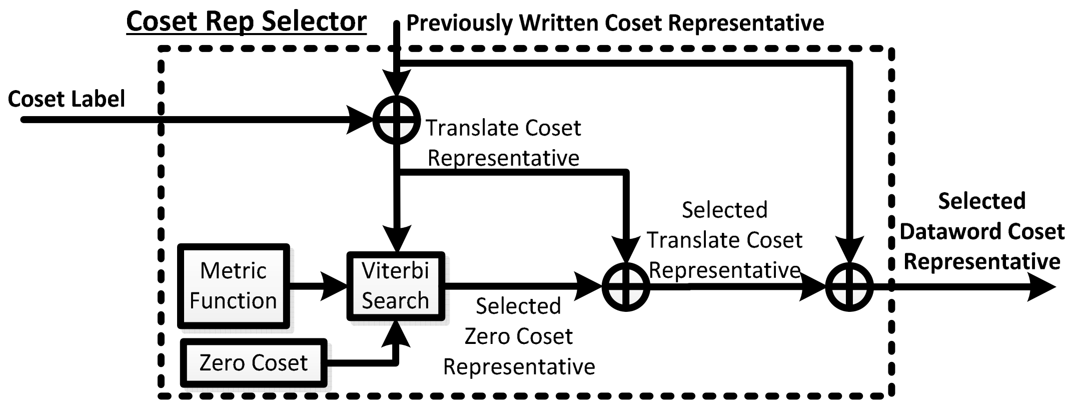

Coset Rep Selector
Previously Written Coset Representative
Coset Label
Translate Coset Representative
Selected Translate Coset Representative
Selected Dataword Coset Representative
Metric Function
Viterbi Search
Zero Coset
Selected Zero Coset Representative

Figure 14: Viterbi-Based Coset Rep Selector

Viterbi has two conditions that exhaustive search does not have. The metric function for Viterbi must produce non-negative numbers and the zero coset must be representable as a finite state machine (FSM). These conditions allow Viterbi to use dynamic programming to calculate the path metrics inductively. We designed our metric functions when using Viterbi so that the produced metric values are non-negative. Both binary convolutional codes and binary block codes [38] have a zero coset that can be represented as an FSM.

3. Extending the Lifetime of Phase Change Memory

This chapter presents our application of coset coding to extend the lifetime of memory locations composed of PCM cells. We apply the coset coding technique presented in Chapter 2 to extend PCM memory lifetime by flipping fewer bits per write compared to writing un-coded datawords. *Flipping fewer bits results in writing less and thus extends lifetime of PCM memory.*

Section 3.1 provides background on PCM. Section 3.2 discusses the PCM failure modes. Section 3.3 discusses other work on extending the lifetime PCM. Sections 3.4 and 3.5 introduce *FlipMin*, our proposed application of coset coding to extend the lifetime of memory locations composed of PCM cells, and provides an evaluation of how well FlipMin reduces the number of bit flips per write. Section 3.6 presents exemplary hardware implementations of a FlipMin encoder and decoder. Section 3.7 presents the area, energy, and latency costs from using our hardware implementations of a FlipMin encoder and decoder. Sections 3.8 and 3.9 discuss how we setup our experiments and present the results of our evaluation of FlipMin for both random and program inputs. Section 3.10 concludes the chapter.

This chapter makes the following contributions:

- Shows how to use coset coding to extend the lifetime of PCM memory by minimizing bit flips per write,
- Presents a method for synergistically combining FlipMin with the ability to tolerate erasures,
- Presents and evaluates exemplary hardware implementations of both a FlipMin encoder and decoder, and
- Presents an evaluation of how well FlipMin extends the lifetime of memory locations composed of PCM cells for both random and program inputs.

3.1 Background

PCM is an emerging memory technology that is poised to either replace or complement Flash memory in computing systems. PCM has faster program/erase times than Flash as well as greater endurance. A PCM cell has three parts: an electrode on top that is used to sense the cell level; the PCM chalcogenide material that changes state; and a heater to change the state of the material. A PCM cell stores a single bit using two states, each of which has a different resistance.

A PCM cell has a low-resistance and a high resistance state. Figure 15 depicts both resistance states of a given PCM cell. The left side of the figure shows the low-resistance state of a PCM cell and the right side of the figure shows the high-resistance state of a PCM cell. To write a PCM cell, a heater melts part of the chalcogenide material and cools it at a given rate. A current is then passed through the cell to read the resistance value. When cooled at a fast rate, the chalcogenide turns amorphous and has a high resistance. When cooled at a slower rate, the chalcogenide turns crystalline and has a low resistance. When the resistance of a PCM cell is low (i.e., the chalcogenide is mostly crystalline), one bit value is read out; when the resistance of a PCM cell is high (i.e., the chalcogenide is mostly amorphous), the other bit value is read out.

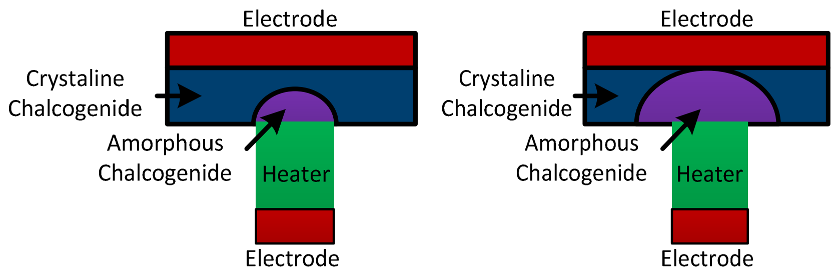


Figure 15: PCM Cell in Low and High Resistance States

Although PCM holds promise as an emerging non-volatile storage technology, it is a *write limited memory,* i.e., repeatedly writing to PCM cell eventually results in the cell unable to change state.

3.2 Failure Modes

PCM cells have been shown to exhibit two different failure modes [60] due to repeated writes: the observed failure modes are stuck in low resistance (i.e., closed) and stuck in high resistance (i.e., open). A PCM stuck-at closed failure is due to elemental segregation in the chalcogenide material. A PCM stuck-at open failure is due to voids between the chalcogenide material and the electrode. A PCM cell that cannot change resistances cannot change value and is referred to as a "stuck-at" cell. A stuck-at PCM cell can be read but cannot be written.

Stuck-at closed and stuck-at open PCM failure modes result from repeated cycling between states. A PCM cell can change state a fixed number of times before failing. We extend the lifetime of the memory location by reducing the number of PCM cells that change states per write compared to writing un-coded data directly.

3.3 Related Work

We group existing schemes for extending the lifetime of memory locations composed of PCM cell into four categories. We list prior schemes in Table 3; shaded schemes in the table represent those that we will not quantitatively compare against in this chapter. With respect to the non-shaded schemes in the table, Section 3.3.1 discusses schemes that use bit flip reduction (BFR). Section 3.3.2 discusses schemes that use error correction based techniques. Section 3.3.3 discusses the technique of adding additional memory cells. Section 3.3.4 discusses wear-leveling techniques.

Table 3: PCM lifetime extension schemes. We quantitatively compare to un-shaded rows.

Approach	Scheme	Instantiation	Granularity	Overhead	Why No Quant. Comparison
bit flip reduction	Flip-N-Write (FnW) [16]	FnW per-byte	8 bits	1 bit=12.5%	
		FnW per-word	64 bits	1 bit=1.56%	subsumed by FnW per-byte
	Coset Coding	*discussed in paper*	64 bits	*tunable*	
error/erasure Correction	ECC	Hamming (72,64)	64 bits	8 bits=12.5%	
	ECP [60]	ECP_6	block ~ 512 bits	61 bits=11.9%	
		ECP_{12}	block ~ 512 bits	121 bits=23.6%	
		ECP-ideal	block ~ 512 bits	0	
	Pay-As-You-Go [55]		entire memory	*tunable*	subsumed by ECP-ideal
	SAFER [62]	SAFER8	block ~ 512 bits	22 bits=4%	subsumed by ECP-ideal
		SAFER32	block ~ 512 bits	55 bits=10.7%	subsumed by ECP-ideal
	RDIS [46]	RDIS3	block ~ 512 bits	*see †*	subsumed by ECP-ideal
	FREE-P [71]		block ~ 512 bits	64 bits=12.5%	requires OS support
	DRM [31]		page ~4KB	*see ǂ*	requires OS support
adding memory cells	DoubleMem		64 bits	64 bits*=100%	

† *Overhead is listed as 18%, but RDIS does not account for overheads to track erasures.*
ǂ *12.5% to track erasures plus 100% for paired pages plus a single 1KB "ready table"*
* *Actual overhead is greater than 64 bits due to extra state bits to track which copy of the location is being used.*

3.3.1 Postponing Wear-out: BFR

One can extend the lifetime of memory locations composed of PCM cells by writing a codeword that flips fewer bits per write compared to writing an un-coded dataword. All else being equal, flipping fewer bits when writing to a memory location results in that location lasting for a greater number of writes. To the best of our knowledge, the only prior work in this area is Flip-N-Write [14]. At each write, Flip-N-Write chooses to write either the dataword or its inverse, depending on which requires fewer bit flips. Flip-N-Write adds a single bit per location to indicate whether the data is inverted or not. Flip-N-Write is coset coding with the repetition code with a metric function that minimizes bit flips. In this dissertation, we explore the repetition code and other coset codes for reducing the number of bit flips per write to postpone wear-out of a memory location composed of PCM cells.

A non-coding method for minimizing bit flips is to combine multiple writes in a buffer composed of memory that is not write-limited [37] before writing to a memory location composed of PCM cells. Write coalescing is a useful technique that is orthogonal to all prior work and to our work.

3.3.2 Tolerating Wear-out: Error Correction

One can extend the lifetime of memory locations composed of PCM cells by tolerating bit errors after wear-out occurs. Tolerating wear-out is effective when a minority of memory cells at some location granularity (e.g., byte, line, etc.) fail far earlier than average making the entire location unusable. Example schemes include ECC and some techniques developed for PCM [27][40][49][56][63]. One prominent scheme is Error Correcting Pointers (ECP) [54]. The ECP scheme tolerates errors in known bit positions (i.e., *erasures*) in memory locations by maintaining pointers to these bit positions and adding bits to be used as replacements. For example, ECP_6 operates at a 512-bit location granularity, and it keeps six 9-bit pointers ($\log_2(512) = 9$) and 6 replacement bits for tolerating up to 6 erasures in the 512-bit location. There has been a large amount of work that extends and optimizes ECP, including Pay-As-You-Go [49], SAFER [56], and RDIS [40]. We both compare to and combine our coset coding based scheme with ECP with the intent of extending the lifetime of memory locations composed of PCM cells.

3.3.3 Adding Memory Cells

One can extend the lifetime of memory locations composed of PCM cells by adding more memory cells and using these memory cells for purposes of extending memory location lifetime rather than increasing memory capacity. For example, if a memory location is logically 64-bits, we can use 128 physical bits in a scheme we call DoubleMem. With DoubleMem, initially the first 64 bits of the physical location are used to store data. When the first 64-bit physical location fails, the second 64-bit physical location is used to store data. There has been research into more sophisticated methods for adding memory cells to improve lifetime, including Waterfall codes and hypercells [36], but they all share the same idea. These techniques are complementary to our work.

3.3.4 Wear-leveling

One can extend the lifetime of memory locations composed of PCM cells by evenly wearing memory cells in a memory module. There are two kinds of wear-leveling: intra-location wear-leveling and inter-location wear-leveling. *Intra*-location wear-level schemes level out the wear in a given location uniformly (e.g., by remapping logical bit positions) [33][64]). These schemes require state to track the current bit position mappings for each location and sophisticated heuristics to decide when and how to remap bit positions. *Inter*-location wear leveling schemes seek to avoid writing to some locations more frequently than others. These schemes [33][50][51][55][64] avoid these situations by dynamically mapping from logical locations to physical locations in ways that are similar to but simpler than virtual memory translations from virtual pages to physical pages. This work is complementary to our work.

3.4 FlipMin

This section presents *FlipMin*, our coset coding based technique for extending the lifetime of memory locations composed of PCM cells. FlipMin extends the lifetime of memory locations composed of PCM cells by writing coset representatives that flip fewer bits than writing datawords directly. Section 3.4.1 discusses our assumptions when developing FlipMin for PCM. Section 3.4.2 discusses how FlipMin selects a coset representative to write to a memory location. Section 3.4.3 discusses how to use FlipMin to write to memory locations with stuck-at PCM cells.

3.4.1 System Model Assumptions

We assumed when designing FlipMin for PCM cells that we can 1) write both transitions 0-to-1 and 1-to-0 with equal cost (a formal model for this is known as the Write Efficient Memory (WEM) model [3]), and 2) read a memory location before writing to it (a requirement of using the translate coset method for determining which coset representative to write as discussed in Section 2.4.1). Both these assumptions about PCM have been made in prior work [9][14].

3.4.2 FlipMin Coset Representative Selection

The coset representative selection process begins with the dataword and ends with the coset representative that is written to memory. Section 3.4.2.1 lists each of the steps and provides a description of the actions taken during each step. Section 3.4.2.2 presents an example of FlipMin coset representative selection using a repetition coset code.

3.4.2.1 Steps

In this section we present three steps, illustrated in Figure 16 and described below, that describe how FlipMin selects a coset representative to write that maximizes BFR.

Step 1: Generate the Coset Label. The coset label (defined in Section 2.1) uniquely identifies the coset corresponding to the dataword. The dataword is multiplied by $\mathbf{H}^{\#}$ (defined in Section 2.3.1.2) to generate the coset label.

Step 2: Generate the Translate Coset. FlipMin selects a coset representative to write based on information from the translate coset (discussed in Section 2.4.1). A translate coset representative is first generated by XORing the previously written coset representative and the coset label. The entire translate coset is then enumerated using the zero coset. The coset representatives in the translate coset indicate which cells will flip when writing each coset representative in the dataword coset.

Step 3: Generate the Coset Representative that is Written to Memory. To generate the coset representative that will be written to memory, the translate *coset leader*—the coset representative with the fewest number of 1s—is determined using exhaustive search (described in Section 2.4.3). We then use the translate coset leader to find the coset representative that will flip the fewest number of bits and XOR the translate coset leader with the previously written coset representative to produce a coset representative in the dataword coset. When written, this coset representative will flip the fewest number of bits of all the coset representatives in the dataword coset.

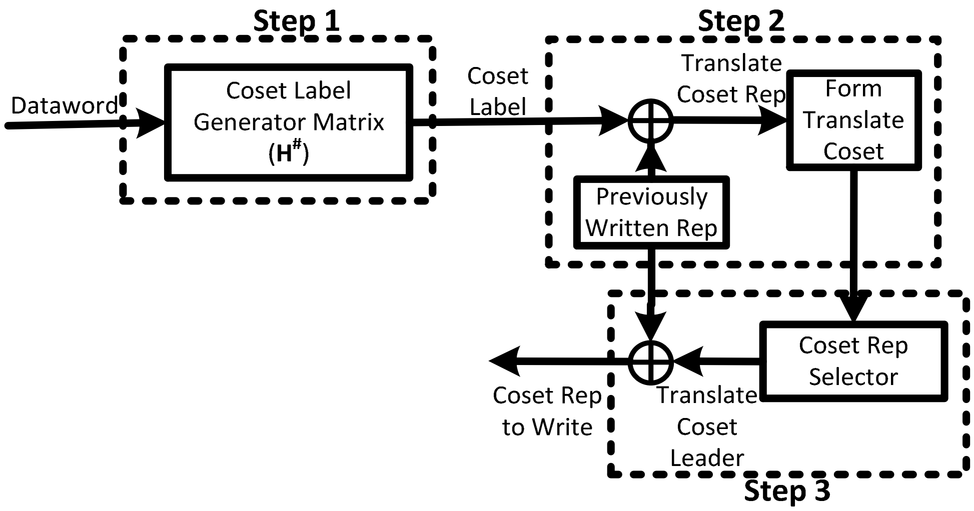


Figure 16: Selecting the Preferred Coset Rep to Write Using FlipMin

Whether or not writing a coset representative using FlipMin results in a BFR depends on the coset code that is used. We use the FlipMin metric function with coset codes that have coset representatives in each coset with an equal or lower weight than the corresponding un-coded dataword. Selecting coset representative to write using FlipMin from one of these coset code results in a bit flip reduction when compared to writing the un-coded dataword directly.

3.4.2.2 Example

In this section, we provide an example illustrating the FlipMin coset representative selection process using the repetition code shown in Table 4. This repetition code has two coset representatives per dataword. For example, the coset representatives for 01 are 010 and 101.

Table 4: 2-Bit to 3-Bit Repetition Coset Code

Dataword	Coset Representative 1	Coset Representative 2
00	000	111
01	010	101
10	100	011
11	110	001

Figure 17 shows each of the three steps to FlipMin coset representative selection using a repetition code. For this example, the previously written data Is 111 and the dataword is 01. Step 1 consists of converting from the dataword (01) to the coset label (010). For the repetition code, we simply append a “0”. In Step 2, the coset label (010) is converted into a translate coset representative (101) by XORing it with the previously written data (111). The other translate coset representative (010) in the translate coset is then generated by flipping all of the bits in the first translate coset representative (101). In Step 3, the translate coset representative with the fewest number of 1s (010) is selected by the coset rep selector and XORed with the previously written data (111) to produce the coset representative to write to memory (101).

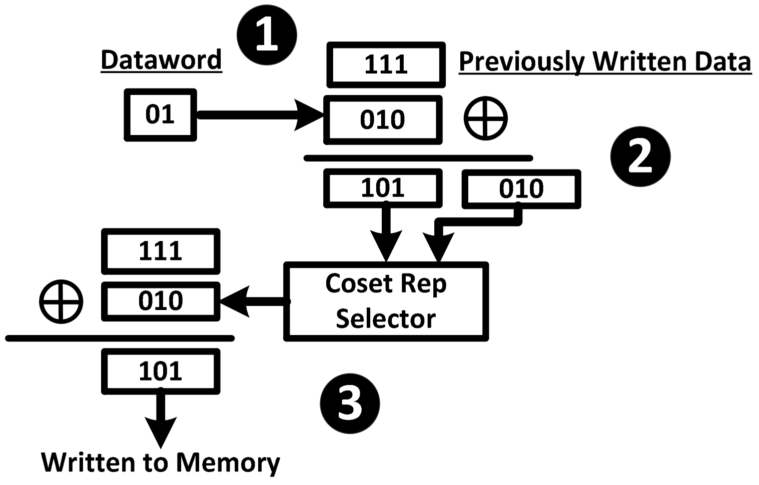


Figure 17: FlipMin Coset Representative Selection Process with the Repetition Coset Code in Table 4

3.4.3 Coset Erasure Matching (CEM)

In addition to reducing bit flips, FlipMin tolerates stuck-at cells, i.e., FlipMin allows for writes to a PCM memory location with stuck-at cells; we call the FlipMin process to write to memory locations with stuck-at memory cells *coset erasure matching* (CEM). CEM tolerates stuck-at PCM cells by selecting a coset representative to write that matches the values of the stuck-at PCM cells in a given memory location. CEM first reads out a *fault mask* (a vector that indicating which bits are stuck-at) that corresponds to a given memory location and then uses the memory location fault mask to determine which bits cannot change when selecting a coset representative to write.

The following example as illustrated in Figure 18 demonstrates the CEM write process when one or more stuck-at PCM cells are present in a memory location. The figure contains an example translate coset along with a *fault mask,* defined as a vector where a “1” indicates a memory cell that cannot be changed. The fault mask here has a “1” in the third position from the left so this memory cell cannot change value during a write. The translate coset for this code is shown on the figure below the fault mask vector with the values in black with white text indicating translate coset representatives that cannot be written and values in white with black text indicating translate coset representatives that can be written. In the figure, for example, the coset representative 11101110 cannot be written because it would flip the third bit, while the coset representative 00010001 can be written since it would not flip the third bit. Even though a bit value cannot be changed, a valid coset representative can still be written when using CEM.

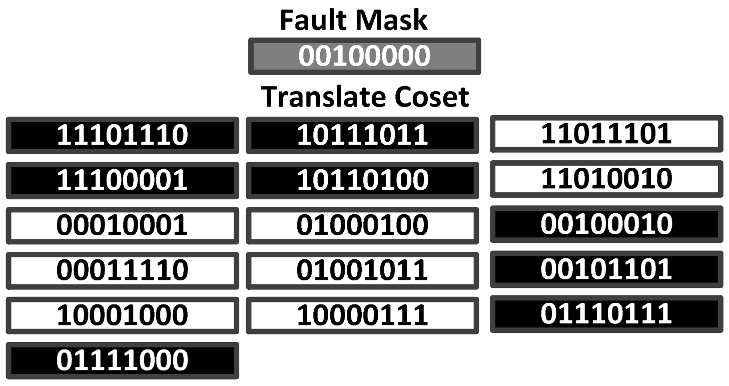


Figure 18: Coset Erasure Matching Example

3.5 FlipMin BFR Evaluation

We ran experiments to measure bit-flip reduction properties of coset codes with FlipMin. We analyzed FlipMin with the following coset codes: Parity(72,64) (equivalent to a previously developed scheme known as Flip-N-Write [14]), two Reed-Muller [52] codes FM-RM(1,3), and a truncated version of RM(1,7) (FM-RM(1,7)T). For each coset code, we used the dual of the code listed (i.e., FM-RM(1,3) is the dual of RM(1,3)). Section 3.5.1 presents our methodology for determining the BFR of each code, and Section 3.5.2 presents our results.

3.5.1 Methodology

To simulate the average BFR, we used an in-house numerical tool that writes to a 64B cache line. We ran simulations for each of the three coset codes until the BFR converged to a constant value, and calculated the BFR for each coset code with FlipMin using the following equation:

$$BFR = 1 - \frac{\#\ Bit\ Flips\ in\ FlipMin}{\#\ Bit\ Flips\ in\ Uncoded}$$

3.5.2 Results

We found that for our simulations the BFR of FM-RM(1,3) and FM-RM(1,7)T is higher than that of FM-Parity(72,64). Table 5 presents the BFR for FM-Parity(72,64), FM-RM(1,3), and FM-RM(1,7)T. We found that the BFR for FM-RM(1,3), FM-RM(1,7)T, and FM-Parity(72,64) codes is 31.2%, 24.5%, and 15.8%, respectively.

Table 5: Bit Flip Reduction

Coset Code	Bit Flip Reduction
FM-Parity(72,64)	15.8%
FM-RM(1,7)T	24.5%
FM-RM(1,3)	31.2%

The BFR of FM-RM(1,3) is higher than that of FM-RM(1,7)T because of the *covering radius*—the maximum number of bits a vector can be different from the coset representatives in the zero coset. The smaller the covering radius for a given codeword length, the larger the BFR [7][10].

Shorter Reed-Muller codes have a smaller covering radius than longer Reed-Muller codes. Since shorter Reed-Muller codes have a smaller covering radius, they have a higher BFR than longer Reed-Muller codes. The smaller covering radius of FM-RM(1,3) results in a 31.2% BFR as compared to 24.5% for FM-RM(1,7)T.

3.6 Implementation

We preset an exemplary hardware implementation of a FlipMin encoder and decoder that consists of an encoder for translating from input to coset representative and a decoder for translating from a coset representative back to a dataword. Section 3.6.1 describes the system model and where FlipMin can be implemented in a computer memory hierarchy; Section 3.6.2 presents a hardware implementation of a FlipMin encoder; and Section 3.6.3 presents a hardware implementation of a FlipMin decoder.

3.6.1 System Model

A hardware implementation of FlipMin can be integrated into any of the existing hardware units on a PCM daughter board. Ideally, any implementation of FlipMin is physically located as close to the PCM cells as possible since FlipMin requires a read before a write. Figure 19 shows one location a hardware implementation of FlipMin can be integrated into an existing computer memory system. In this figure, PCM is attached to a processor as its main memory in the form of a DIMM (possibly similar to [4]) that has PCM chips on both sides of a daughter board. If a chip were present to coordinate activities between the PCM chips and provide inter-location wear-leveling as illustrated by “Chip Ctrl” in Figure 19, a hardware implementation of FlipMin could be integrated into this chip; alternatively, we could integrate FlipMin into the write controller located inside the PCM chips.

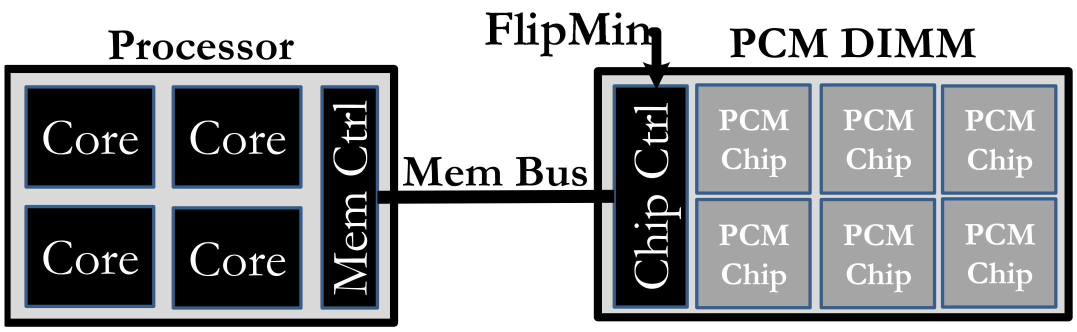


Figure 19: Example PCM Implementation that Integrates FlipMin into PCM Chips

3.6.2 Encoder

This section presents the hardware for a FlipMin encoder. First, we list the different encoding steps. Then, for each step, we present our hardware implementation.

The Encoding Steps for our Hardware Implementation of FlipMin:

1. Generate the coset label (defined previously in Section 2.5.2.1 as a coset representative inside a coset that uniquely identifies the coset) from a dataword
2. Generate a translate coset representative
3. Generate the translate coset
4. Find the minimum weight element in the translate coset
5. Compute the coset representative to write from the minimum weight element in the translate coset.

Step 1: Generate a coset label from a dataword (Figure 20). To generate the coset label from a dataword, we multiply the dataword by $\mathbf{H}^{\#}$ (defined in Section 2.3.1.2) over GF(2). Figure 20 depicts a hardware block diagram for implementing GF(2) matrix multiplication. We define the dataword to be *k* bits long and we define the coset label to be *n* bits long. Since we are doing a matrix multiply over GF(2), we do not use the standard addition and multiplication operations; rather, we add using XOR gates and multiply using AND gates.

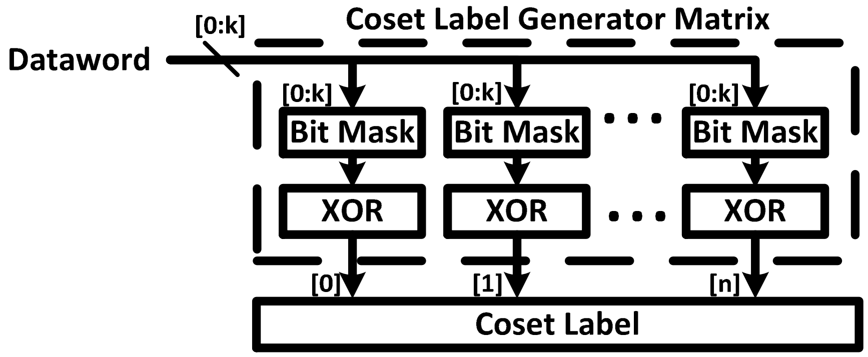


Figure 20: Hardware Block Diagram for Coset Label Generation

Two hardware units are used per column to perform matrix multiply over GF(2): a unit for the multiplication and a unit for the addition. The multiplication unit consists of a *bit mask*—a unit that selects bits from a given coset representative. The bits selected correspond to the 1s in the corresponding column of the coset representative generator matrix. For example, if the matrix column has the value 101, only the first and final bits of the input dataword would be selected. The addition unit consists of a *k* bit XOR gate. The different bits selected by the bit mask are XORed together to produce a bit in the coset label. The bits from the different columns are concatenated to form the coset label.

Step 2: Generate the translate coset label (Figure 21). Generating the translate coset label consists of a single XOR operation between the previously written data and the data to write.

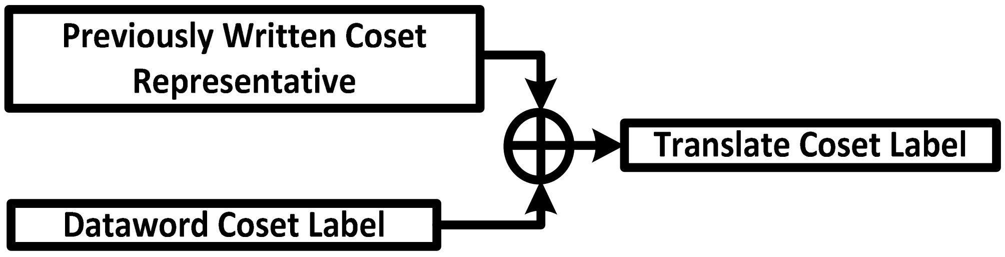


Figure 21: Generating the Translate Coset Label

Step 3: Generate the translate coset (Figure 22). To obtain the translate coset, we XOR the translate coset label with the zero coset stored in a ROM. Generating the translate coset requires a single XOR operation per coset representative in the coset. The resulting vectors are the coset representatives of the translate coset.

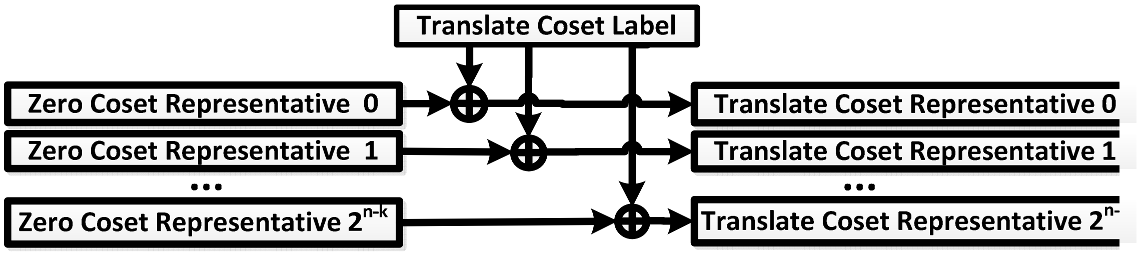

Translate Coset Label
Zero Coset Representative 0
Zero Coset Representative 1
Zero Coset Representative 2^n-k
Translate Coset Representative 0
Translate Coset Representative 1
Translate Coset Representative 2^n-

Figure 22: Generating the Translate Coset

Step 4: Find the minimum weight translate coset representative (Figure 23). To find the translate coset representative with the fewest 1s, we input all coset representatives in the translate coset into the “Find Minimum Weight Rep” module. We implemented the “Find Minimum Weight Rep” module using the implementation of exhaustive search described in Section 2.4.3.

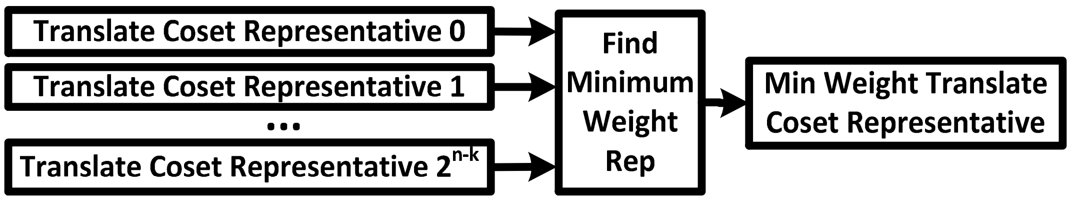

Translate Coset Representative 0
Translate Coset Representative 1
•••
Translate Coset Representative 2^{n-k}
Find Minimum Weight Rep
Min Weight Translate Coset Representative

Figure 23: Coset Metric Calculation and Selection Logic

Step 5: Compute the coset representative to write from the minimum weight translate coset representative (Figure 24). The final step is finding and writing the coset representative in the original coset corresponding to the selected translate coset representative. We XOR the translate coset representative with the previously written data to produce the coset representative in the original coset. This coset representative is then written to the memory location.

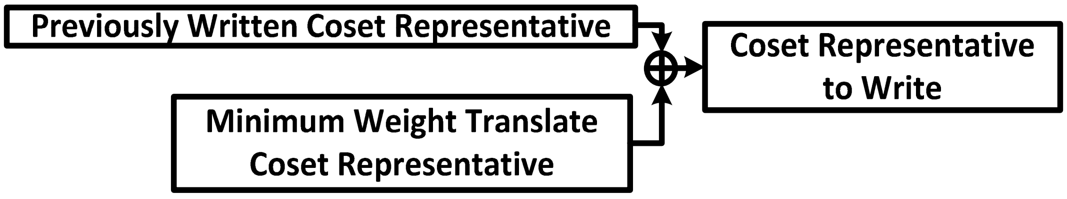

Previously Written Coset Representative
Minimum Weight Translate Coset Representative
Coset Representative to Write

Figure 24: Determining the Coset Rep to Write from the Translate Coset Leader and Previously Written Coset Representative

3.6.3 Decoder

Decoding FlipMin consists of a GF(2) matrix multiply. An example decoder hardware implementation is depicted in Figure 25. The process for the GF(2) matrix multiply is the same as in Figure 20 except that the generator matrix of the code is used, the input is of length *n,* and the output is of length *k.* The output of this matrix multiply is the dataword that was originally encoded using FlipMIn.

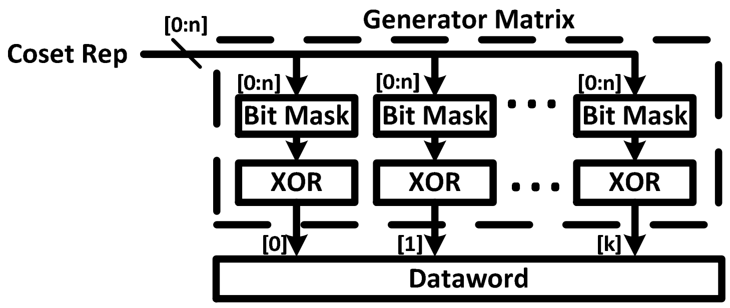


Figure 25. Hardware Block Diagram for Decoding

3.7 Hardware Costs

We determined the costs of using our exemplary hardware implementations of the FlipMin encoder and decoder for a set of coset codes. We compared the costs of our FlipMin encoder and decoder hardware designs to a PCM chip data sheet [41]. The PCM chip in this data sheet has 16-bit I/O width, so we multiplied the area and power costs from the data sheet by 4. Section 3.7.1 lists the different coset codes for which we designed FlipMin encoders and decoders; Section 3.7.2 presents the process we used to determine costs; Section 3.7.3 presents the costs of the encoder; and Section 3.7.4 presents the costs of the decoder.

3.7.1 Coset Codes Evaluated

We evaluated the hardware implementation of FlipMin example encoders and decoders for a set of Reed-Muller codes [52] and a parity code. Other codes could be used as well provided that the code divides up a space into cosets. Table 6 lists the duals of the different Reed-Muller and parity codes used for FlipMin for which we designed encoders and decoders. When we refer to these codes, we will be referring to the dual.

We designed our coset coding encoder and decoder to be the size of a standard word in a computing system (64-bits). For codes that accept inputs smaller than 64-bits, we divide the dataword into sub-vectors and encode each sub-vector independently. The coset codes we implemented a hardware encoder and a hardware decoder for are listed in Table 6. For the FM-Parity(72,64) and the FM-RM(1,7)T (truncated RM(1,7)) coset code, 1.125x PCM cells are required to store data compared to un-coded datawords written directly. For the FM-RM(1,3) coset code, 2x PCM cells are required to store data compared to writing un-coded datawords directly.

Table 6: FlipMin with Different Block Codes

Coset Code	Storage Overhead (per 64-bit Dataword)	Comments
FM-Parity(72,64)	8/64=0.125x	Performed on 8-bit sub-vector
FM-RM(1,7)T	8/64=0.125x	
FM-RM(1,3)	64/64=1x	Performed on 4-bit sub-vector

3.7.2 Process Used to Evaluate Hardware Costs

We used tools from the Synopsys suite of programs to determine the costs of our exemplary hardware implementations of a FlipMin encoder and decoder. First, we used Synopsys Design Compiler (DC) to synthesize from RTL to a gate-level netlist using gates from the Nangate 45nm semi-custom library [46]. Second, we used Synopsys VCS with 1,000 randomly generated inputs to determine the switching activity factor for the wires in the design. Third, we used Synopsys IC to layout and floor plan the design. Finally, we used Synopsys Design Compiler topological mode to determine the area, latency, and energy costs.

3.7.3 Encoder Costs

We found the cost of adding a FlipMin encoder to a PCM chip to be negligible. Table 7 lists the simulation latency, energy, and area costs for our implementations of FlipMin encoders for the codes listed in Table 6. According to our PCM datasheet [41], the write time for our PCM chip is 60-120us. The maximum delay for encoding using the RM coset codes is 12.86ns, or 0.021% of the PCM chip write latency. The worst case energy consumption for the encoder is 63.4 pJ. The typical idle current of 4 PCM devices is about 320µA and the minimum rail voltage is 1.7V, so the minimum idle power is approximately 544µW. If a block is encoded every 60µS, our largest encoder would take 1.06µW which represents 0.19% of the idle power of the PCM chip. The worst case area cost for encoding is 48,344 µm^2, which we believe to be negligible given the typical size of a PCM chip and a DIMM.

Table 7: Coset Coding Encoder Costs

Coset Code	Delay (ns)	Avg Energy (pJ)	Max Energy (pJ)	Area (µm^2)
FM-RM(1,3)	4.09	8.4	10.1	1,160
FM-RM(1,7)T	12.86	56.1	63.4	48,344
FM-Parity(72,64)	0.84	0.4	0.6	503

3.7.4 Decoder Costs

We found the cost of adding a FlipMin decoder to a PCM chip to be negligible. Table 8 lists the simulation latency, energy, and area costs for our implementations of FlipMin decoders for the codes listed in Table 6. From the same PCM datasheet used for the encoding evaluation, the read time is 115ns + 25ns per 16-bit entry. This is over an order of magnitude greater than our worst case decode delay of 0.59ns. The worst case decode energy is 0.4 pJ which if the chip is read at a rate of 60µS, our largest decoder would take 6.67nW (0.001% the idle power of the PCM chip). The worst case area cost for the decoder was 344µm^2 which is significantly less than the encoder size and we believe to be negligible compared to the PCM chip and DIMM size.

Table 8: Coset Coding Decoder Costs

Coset Code	Delay (ns)	Avg Energy (pJ)	Max Energy (pJ)	Area (µm^2)
FM-RM(1,3)	0.38	0.3	0.3	344
FM-RM(1,7)T	0.59	0.3	0.4	221
FM-Parity(72,64)	0.12	0.1	0.2	141

3.8 Experimental Methodology

This section presents our experimental methodology to determine how well FlipMin extends the lifetime of memory locations composed of PCM cells. We used the coset codes listed in Section 3.7.1 for our evaluation of FlipMin. Section 3.8.1 lists different techniques against we compared, and Section 3.8.2 discusses how we modeled wear-out of PCM cells.

3.8.1 Techniques Compared Against

We evaluated and compared the approaches listed in Table 9 for extending the lifetime of memory locations composed of PCM cells. We performed our experiments using an in-house simulator. The storage overhead is the additional percentage of memory cells needed to implement the scheme (e.g., implementing ECP$_6$ requires 11.9% additional overhead). For our comparison, the three shaded schemes in Table 9 had similar storage overheads so we did not normalize them. BFR indicates the bit-flip reduction of each scheme. Erasure correction denotes the number of stuck-at bits each scheme can tolerate.

We compared FlipMin to the following schemes:

- **Bit-Flip Reduction Schemes:** FM-Parity(72,64) on a per-byte granularity (equivalent to Flip-N-Write [16])
- **Error/Erasure Tolerance Schemes:** Hamming(71,64) and several variants of ECP [60].
- **Hybrids:** We combined FlipMin with both ECP and CEM and compared it to combinations of bit-flip reduction schemes (Flip-N-Write and FlipMin) and erasure tolerance schemes (ECP).
- **Additional Memory Locations:** We looked at a technique we call *DoubleMem*—using two memory locations to store a single memory locations worth of data.

Table 9: Schemes to Extend Memory Lifetime

Scheme	Storage Overhead	BFR	Erasure Correction
ECC-Hamming(71,64)	10.9%	0%	8 bits
ECP_6	11.9%	0%	6 bits
ECP_{12}	19.7%	0%	12 bits
ECP_{12}-ideal	0%	0%	12 bits
FM-Parity(72,64)	12.5%	15.8%	0 bits
FM-Parity(72,64)+ECP_6	25.6%	15.8%	6 bits
FM-RM(1,7)T	12.5%	24.5%	0 bits
FM-RM(1,7)T+ECP_6	25.6%	24.5%	6 bits
FM-RM(1,7)T+CEM	24.4%	24.5%	6 bits
FM-RM(1,3)	100%	31.2%	0 bits
DoubleMem	100%	0%	0 bits

3.8.2 Modeling Wear-Out

This section describes how we set up our experiments to evaluate the lifetime gains from using FlipMin compared to prior work.

We considered a memory location unusable after the number of unusable PCM cells in the location exceeded a given threshold. We used the same threshold for all schemes to fairly compare different schemes. For our experiments, we set this threshold to 0.9N where N is the number of memory locations at time 0 for the highest overhead scheme.

We sized each memory location to be the size of a cache block (64B). If a given cache block failed, we assumed that it could not use bits from other blocks. We also assumed that a mechanism such as FREE-P [63] could be used to remap failed blocks.

At time zero, we assumed PCM cells could be written, and over time, PCM cells wear out and become un-writable. We modeled PCM cell lifetime using a Gaussian distribution as in prior work [54]. We set the mean PCM cell lifetime to 10^8 based on published data [41][65]. Changing the mean lifetime of PCM cells did not change our results; rather, it changed the absolute lifetime numbers. As with prior work [54], we modeled variability in PCM cell lifetimes by changing the coefficient of variation (CV) of the PCM cell lifetime distribution. The higher the CV, the more dispersed the distribution. We set the CV to 0.05 to model PCM cell lifetimes with low manufacturing tolerances, and CV to 0.2 to model PCM cell lifetimes with high manufacturing tolerances.

3.9 Results

We ran simulations to determine how effectively FlipMin extends the lifetime of memory locations of PCM cells compared to prior schemes. Sections 3.9.1 and 3.9.2 present our simulation results for random and benchmark inputs, respectively.

3.9.1 Random Input Results

We experimentally evaluated FlipMin to determine how well it extends memory lifetime and to compare it to prior work. The number of cache lines still usable after a given number of writes is the metric we use to evaluate the effectiveness of each memory location lifetime extension scheme.

Each line and table graph presented in this section has the same format. The x-axis is the number of writes performed and the y-axis is the number of cache lines usable after a given number of writes. The number of usable cache lines at a given number of writes is normalized to FM-RM(1,3) which has 100% overhead. We state that a memory location has failed when 0.9N (the number of FM-RM(1,3) memory locations at time zero) locations remain. Each table presented in this section lists the number of writes before 0.9N and the percent improvement over our baseline runs.

FlipMin compared to prior BFR schemes. We compared the lifetime gains of FlipMin with Reed-Muller codes to a previously developed scheme Flip-N-Write [14] (equivalent to FM-Parity(72,64)).

FM-RM(1,7)T has higher memory location lifetime gains compared to FM-Parity(72,64) for random inputs at a CV of 0.05. Figure 26 shows the results for the BFR schemes with memory cell lifetime CV of 0.05. Table 10 lists results for a CV of 0.05. FM-RM(1,7)T has a lifetime gain of 46% while FM-Parity(72,64) has a lifetime gain of 12%. FM-RM(1,3) extends the lifetime of PCM longer than both FM-Parity(72,64) and FM-RM(1,7) with a lifetime gain of 178%.

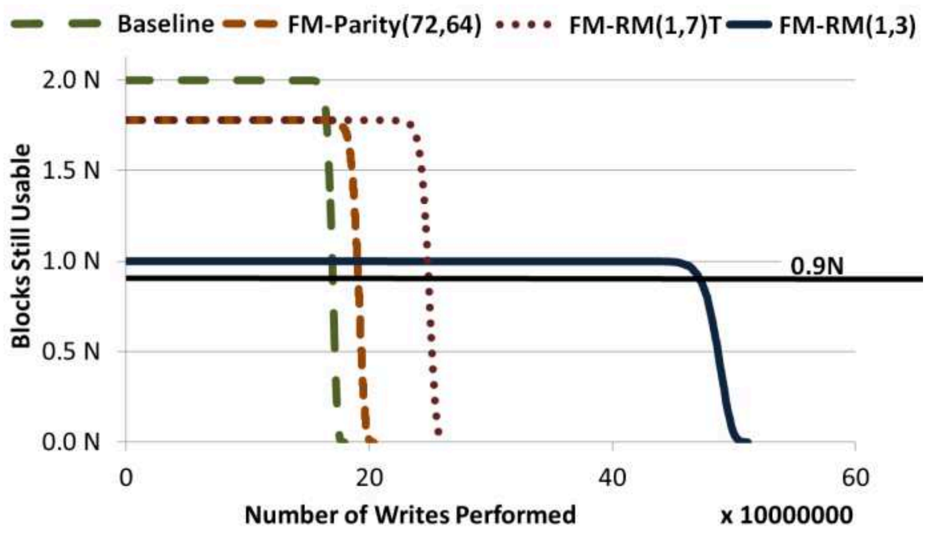


Figure 26. BFR Schemes (CV 0.05)

Table 10: BFR schemes compared (CV 0.05)

Scheme	**CV 0.05**	
	Writes Before 0.9N	**Percent Improvement Over Baseline**
Baseline	1.70e8	0%
FM-Parity(72,64)	1.91e8	12%
FM-RM(1,7)T	2.49e8	46%
FM-RM(1,3)	4.72e8	178%

At a CV of 0.2, the FM-RM schemes have a higher memory location lifetime gain than FM-Parity(72,64). Figure 27 shows these schemes with a memory cell lifetime distribution CV of 0.2. Table 11 lists the results of our comparison. FM-RM(1,7)T has a lifetime gain over baseline of 41% compared to FM-Parity(72,64)'s gain of 23%. FM-RM(1,3) has a lifetime gain of 82%.

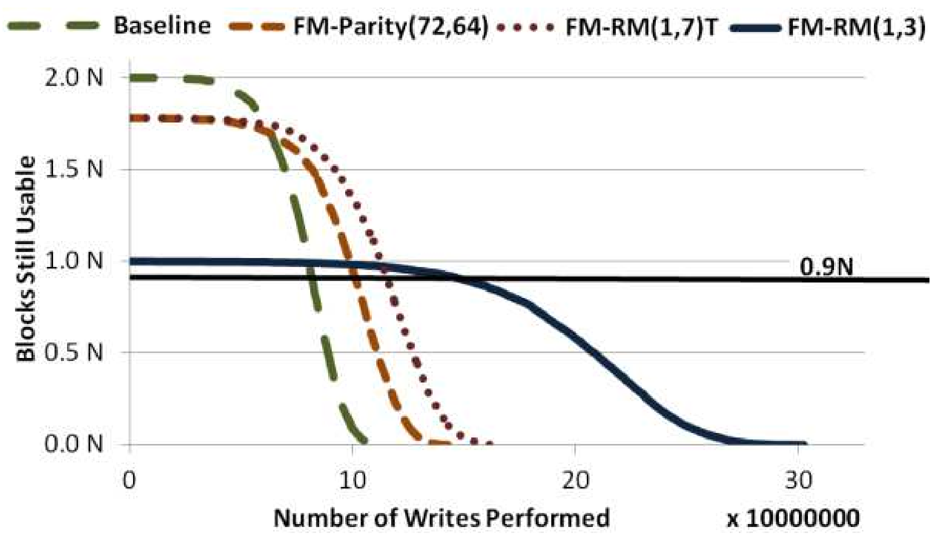


Figure 27. BFR Schemes (CV 0.2)

Table 11: BFR schemes compared (CV 0.2)

Scheme	**CV 0.2**	
	Writes Before 0.9N	**Percent Improvement Over Baseline**
Baseline	8.20e7	0%
FM-Parity(72,64)	1.01e8	23%
FM-RM(1,7)T	1.16e8	41%
FM-RM(1,3)	1.49e8	82%

FlipMin compared to stuck-at tolerance schemes. Since stuck-at tolerance also can extend the lifetime of PCM, we compared FlipMin to the following four stuck-at tolerance schemes: one ECC scheme (Hamming(72,64)) and three ECP [53] schemes (ECP_6, ECP_{12}, and ECP_{12}-Ideal).

For a CV of 0.05, using FlipMin results in higher lifetime gains than stuck-at tolerance. Figure 28 shows the stuck-at tolerance schemes compared to FlipMin for a CV for 0.05, and Table 12 lists the results. FlipMin extends PCM lifetime by 46% while stuck-at tolerance schemes ECC and ECP the cache line lifetime by only 3% and 6%, respectively, for the same overhead.

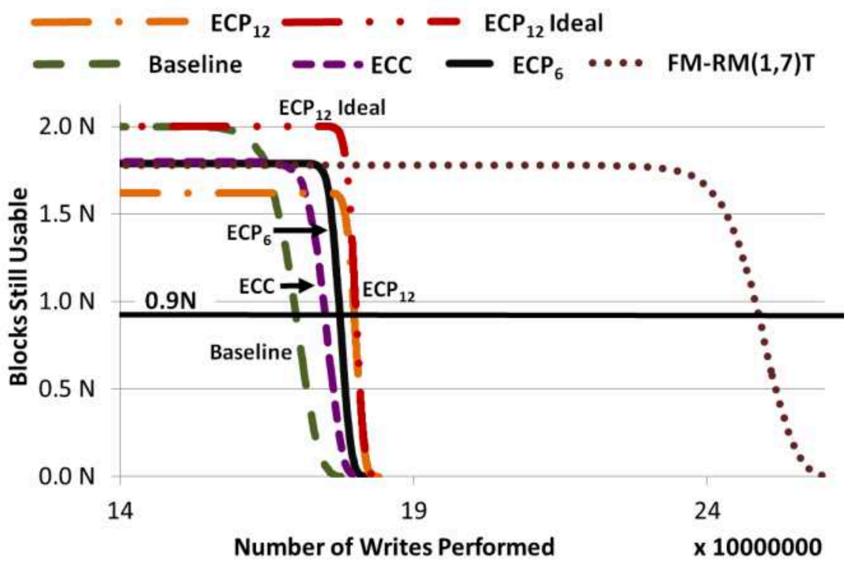


Figure 28. Stuck-at Tolerance Compared to FlipMin (CV 0.05)

Table 12: FlipMin compared to error/stuck-at tolerance schemes (CV 0.05)

Scheme	**CV 0.05**	
	Writes Before 0.9N	**Percent Improvement Over Baseline**
Baseline	1.70e8	0%
ECC-Hamming(71,64)	1.75e8	3%
ECP$_6$	1.78e8	5%
ECP$_{12}$	1.80e8	6%
ECP$_{12}$-ideal	1.80e8	6%
FM-RM(1,7)T	2.49e8	46%

For a CV of 0.2, stuck-at tolerance schemes show higher lifetime improvements while FlipMin shows lower lifetime improvements compared to a CV of 0.05. Figure 29 shows the same schemes for a CV of 0.2, and Table 13 presents the number of writes before 0.9N as well as the percent improvement over the baseline for each scheme. The lifetime gains of FM-RM(1,7)T are

41% at a CV of 0.2 from 46% at a CV of 0.05. A CV of 0.2 increases ECC lifetime improvement to 23% and ECP_{12}-ideal lifetime improvement to 48% from 6%.

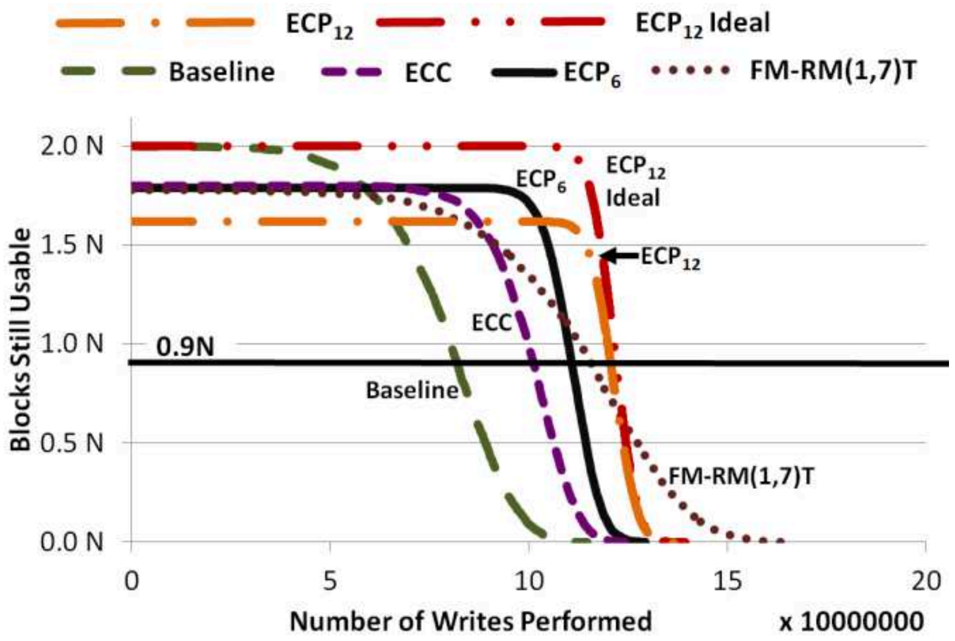


Figure 29. Stuck-at Tolerance Compared to FlipMin (CV 0.2)

Table 13. FlipMin compared to error/stuck-at tolerance schemes (CV 0.2)

Scheme	**CV 0.2**	
	Writes Before 0.9N	**Percent Improvement Over Baseline**
Baseline	8.20e7	0%
ECC-Hamming(71,64)	1.01e8	23%
ECP_6	1.11e8	35%
ECP_{12}	1.20e8	46%
ECP_{12}-ideal	1.21e8	48%
FM-RM(1,7)T	1.16e8	41%

Stuck-at tolerance is more important for higher CV values because the weakest cell in a memory location with a higher CV is weaker than in a lower CV. For example, given a memory location with a CV of 0.05 and a mean cell lifetime of 1e8, the weakest cell lifetime is, for the average of 100,000 different seed values, 85,354,250 bit flips. For a CV of 0.2 the weakest cell lifetime is 41,417,000 bit flips, almost half as many as a CV of 0.05. For random inputs, the lifetime of the weakest cell limits the lifetime of the location. Stuck-at tolerance removes the lifetime limiting effects of the bottom *n* cells on lifetime by providing *n* replacement cells. Stuck-at tolerance is superior to coset coding when the lifetime gains from replacing the *n* weakest cells exceeds that of the coset coding gains. We can also combine both techniques to gain the lifetime extension gains of coset coding and the weak-cell tolerance of stuck-at tolerance schemes.

Combining stuck-at tolerance with FlipMin. To assess the effectiveness of combining stuck-at tolerance with FlipMin, we evaluated the lifetime gains for a set of schemes combining stuck at tolerance with FlipMin (FM-Parity(72,64)+ECP$_6$, FM-RM(1,7)+ECP$_6$, FM-RM(1,7)T+CEM) and a scheme that solely performs stuck-at tolerance (ECP$_{12}$-ideal) with the same area overhead. We evaluated these schemes to determine how well stuck-at tolerance alone does compared to FlipMin combined with stuck-at tolerance at extending memory location lifetime.

We found that for the schemes listed above, FM-RM(1,7)T combined with either ECP$_6$ or CEM had the highest lifetime extension at a CV of 0.05. Figure 30 shows results for the schemes listed above for a CV of 0.05. Table 14 presents results for writes before 0.9N and percent improvement over baseline. FM-Parity(72,64)+ECP$_6$ has a lower lifetime gain than FM-RM(1,7)T+CEM and FM-RM(1,7)T+ECP$_6$ alone. FM-RM(1,7)T combined with a stuck-at tolerance scheme had a lifetime extension of 53% while FM-Parity(72,64) has a lifetime extension of only 19%. ECP$_{12}$-ideal did the worst at only 6%. This shows how FlipMin combined with stuck-at tolerance is more effective than stuck-at tolerance alone for the same area overhead.

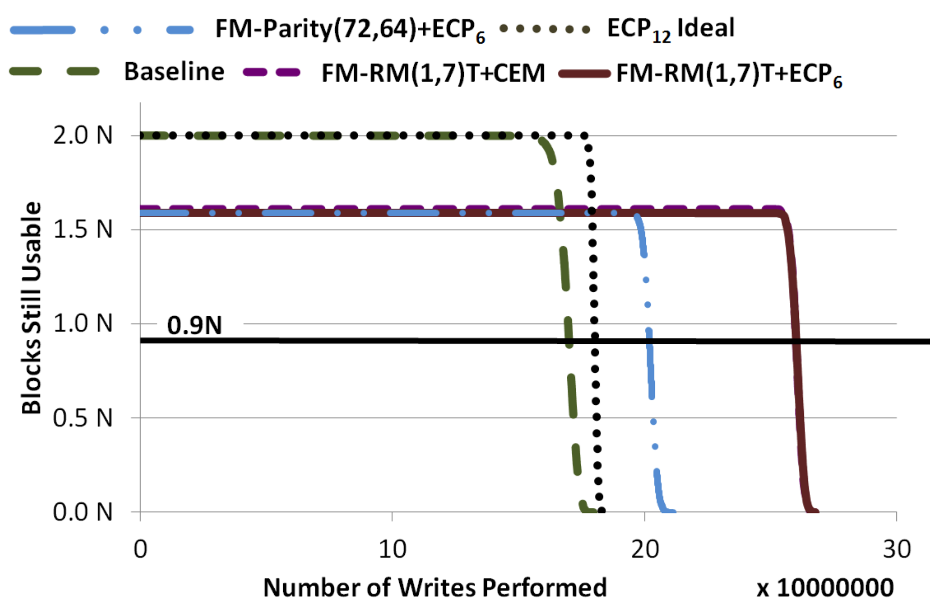


Figure 30. FlipMin Combined With Stuck-at Tolerance (CV 0.05)

Table 14: BFR + stuck-at tolerance compared to error/stuck-at tolerance alone (CV 0.05)

Scheme	**CV 0.05**	
	Writes Before 0.9N	**Percent Improvement Over Baseline**
Baseline	1.70e8	0%
ECP_{12}-ideal	1.80e8	6%
FM-Parity(72,64) + ECP_6	2.02e8	19%
FM-RM(1,7)T + ECP_6	2.60e8	53%
FM-RM(1,7)T + CEM	2.60e8	53%

We found that of the schemes listed in Table 14, FM-RM(1,7)T combined with stuck-at tolerance has the highest lifetime extension at a CV of 0.2. Figure 31 lists these schemes compared at a CV of 0.2. Table 15 lists the writes before 0.9 and percent improvement over baseline. FM-Parity(72,64) again has a lower lifetime extension of 71% while FM-RM(1,7)T combined with either ECP_6 or CEM has a lifetime extension of 95%. ECP_{12}-ideal does better with a lifetime extension of 48%, but is still less than FM-Parity(72,64) or FM-RM(1,7)T when combined with stuck-at tolerance. At a CV of 0.2, FlipMin is still superior to stuck-at tolerance alone.

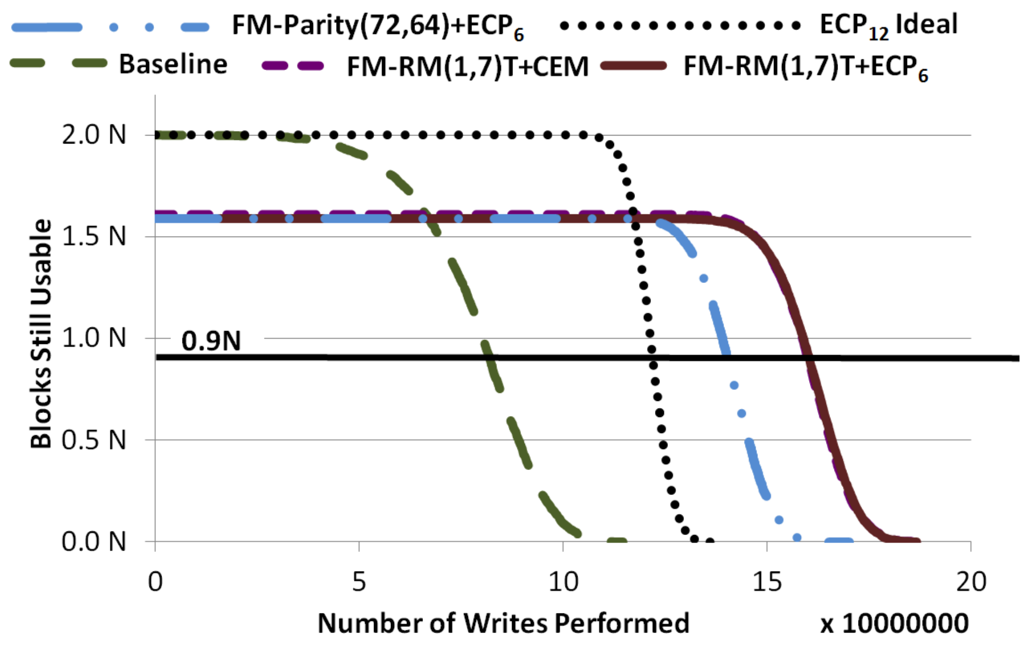


Figure 31. FlipMin Combined with Stuck-at Tolerance (CV 0.2)

Table 15. BFR + stuck-at tolerance compared to error/stuck-at tolerance alone (CV 0.2)

Scheme	**CV 0.2**	
	Writes Before 0.9N	**Percent Improvement Over Baseline**
Baseline	8.20e7	0%
ECP$_{12}$-ideal	1.21e8	48%
FM-Parity(72,64) + ECP$_6$	1.40e8	71%
FM-RM(1,7)T + ECP$_6$	1.60e8	95%
FM-RM(1,7)T + CEM	1.60e8	95%

FM-RM(1,3) compared to DoubleMen. Although the prior schemes improve the lifetime of PCM, they may not provide lifetime gains required for a specific target usage. For these cases, FM-RM(1,3) provides substantial lifetime gains at the cost of 100% area overhead (as shown in Table 9). To provide a fair comparison, we compared FM-RM(1,3) to an un-coded scheme that uses the same area overhead as FM-RM(1,3): DoubleMem (described in Section 3.8.1) . We gave DoubleMem an unrealistic advantage by ignoring the overheads require to track which location was being used.

We compared the lifetime gains for both DoubleMem and FM-RM(1,3) and found that FM-RM(1,3) performed better than DoubleMem for the same overhead at a CV of 0.05. Figure 32 shows DoubleMem and FM-RM(1,3) for a CV of 0.05 and Table 16 lists results. At a CV of 0.05 FM-RM(1,3) extended the lifetime of PCM by 178%, while DoubleMem gave a gain of only 95%.

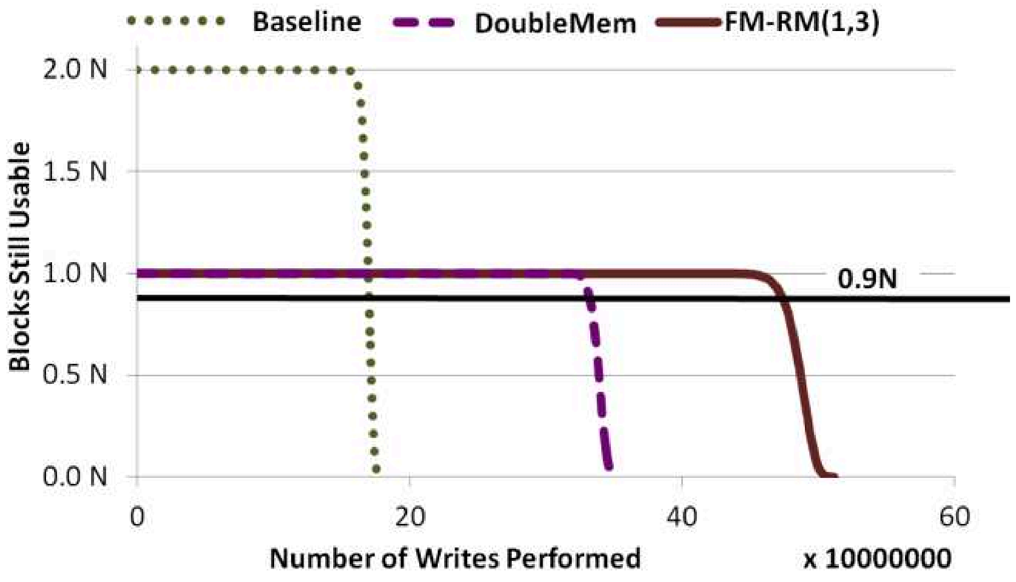


Figure 32. FM-RM(1,3) Compared to DoubleMem (CV 0.05)

Table 16: FM-RM(1,3) Compared to DoubleMem (CV 0.05)

Scheme	CV 0.05	
	Writes Before 0.9N	Percent Improvement Over Baseline
Baseline	1.70e8	0%
DoubleMem	3.31e8	95%
FM-RM(1,3)	4.72e8	178%

At the higher CV of 0.2, there are smaller percent improvements for both DoubleMem and FM-RM(1,3), but FM-RM(1,3) is still better than DoubleMem at extending memory location lifetime. Figure 33 and Table 17 show results for a memory lifetime CV of 0.2. FM-RM(1,3) has a lifetime gain of 82% and DoubleMem has a lifetime gain of 56%.

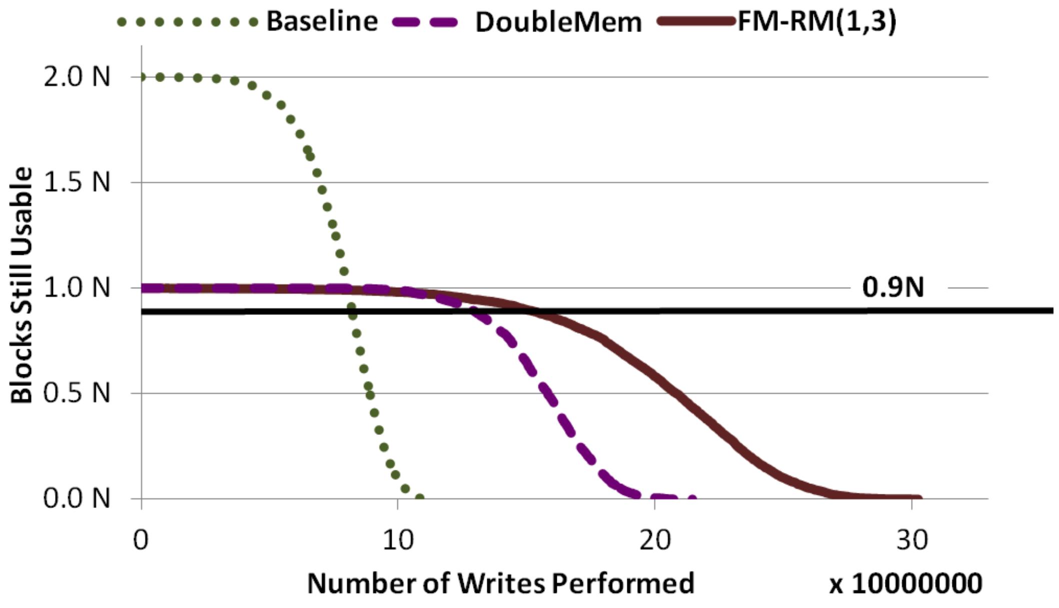


Figure 33. FM-RM(1,3) Compared to DoubleMem (CV 0.2)

Table 17. FM-RM(1,3) Compared to DoubleMem (CV 0.2)

Scheme	CV 0.2	
	Writes Before 0.9N	Percent Improvement Over Baseline
Baseline	8.20e7	0%
DoubleMem	1.28e8	56%
FM-RM(1,3)	1.49e8	82%

3.9.2 Benchmark Results

We evaluated memory location lifetime extension for a suite of benchmarks that write to PCM to assess how well FlipMin extends lifetime when PCM is used as part of the memory sub-system in a computing system. Section 3.9.2.1 presents the methodology we used to evaluate benchmark inputs, and Section 3.9.2.2 presents our results.

3.9.2.1 Methodology

We assessed the effectiveness of the lifetime extension schemes listed in Table 9 for the following Hadoop benchmarks that come with the Apache Hadoop distribution [24]. We used the Gem5 simulator [8] to produce memory traces that we fed into an in-house simulator. Table 18 lists the parameter we used with the Gem5 simulator. We simulated a single in-order core with L1D and L1I caches and a single L2 cache. In terms of memory, our system was configured to have 128MB of memory and 2GB of swap.

Table 18: System Configuration

CPU	X86-64 in-order core at 2.0GHz
L1 D-Cache	64kB, 2-way associative, 64B Line Size
L1 I-Cache	32kB, 2-way associative, 64B Line Size
L2 Cache	2MB, 8-way associative, 64B Line Size
Memory	128MB, 2GB Swap

To obtain a large number of writes to a large number of lines in a reasonable amount of time, we produced traces of writes to each line and then projected these traces onto a single target page of memory with lines numbered *1* to *B.* This projection was performed by mapping line *X* to line *X mod B.* The trace of writes to the target page first line became the concatenation of the traces for lines *1, B+1, 2B+1, and so forth.*

3.9.2.2 Results

We evaluated the lifetime of memory locations using benchmark inputs for PCM cell lifetime distributions of CV of 0.05 and 0.2.

CV of 0.05 Results. Figure 34 shows our results for benchmark inputs at a CV of 0.05. We plot the lifetime gain over the baseline. We determined that the memory location failed after 0.9N memory locations remained. FM-RM(1,7)T and FM-RM(1,7)T + stuck-at tolerance extend the lifetime of PCM by 20.25% and 64% respectively. Stuck-at tolerance alone had little impact on memory location lifetime and in some cases decreased the lifetime of a memory location with benchmark inputs at a CV of 0.05 On average, Hamming(71,64) decreased the lifetime of the memory location by 5.33%. ECP_6 provided a lifetime increase of only 9.18% as compared to 32.37% for FM-RM(1,7)T at the same overhead.

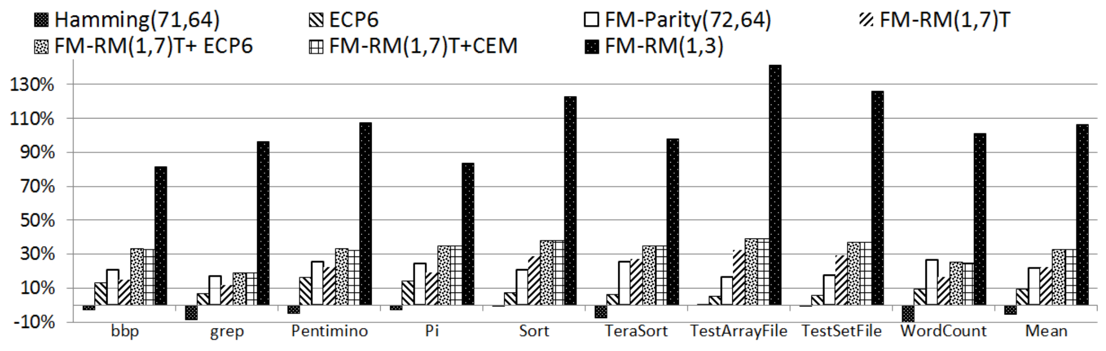


Figure 34. Lifetime extension schemes compared using Hadoop inputs. CV 0.05

CV of 0.2 Results. Figure 35 shows our results for benchmark inputs with a CV of 0.2. The lifetime gains of the FlipMin + stuck-at tolerance schemes were on average higher for a CV of 0.2 (~60%) than with a CV of 0.05 (~30%). FM-RM(1,7)T and stuck-at tolerance combined are on par with RM(1,3) with an average lifetime improvement of about 60% at a CV of 0.2. FM-RM(1,7)T alone does slightly worse on average with a 20.25% lifetime gain compared to 22.27% at a CV of 0.05. Stuck-at tolerance schemes do better at a CV of 0.2 than a CV of 0.05 with benchmark inputs.

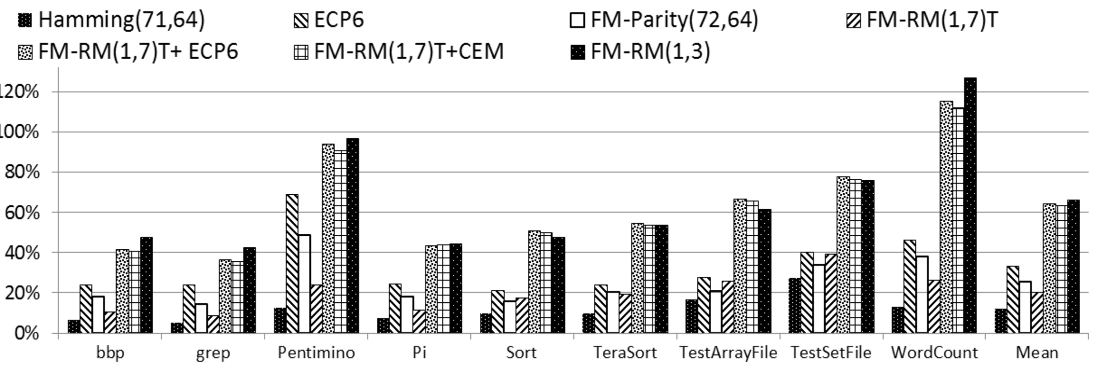


Figure 35. Lifetime extension schemes compared using Hadoop inputs. CV 0.2

3.10 Conclusion

We have shown that coset coding enables us to optimize writing memory structures. In particular, we have shown how to use coset coding to avoid wear-out—by reducing bit flips—and to tolerate the bits that eventually wear out. We have not exhausted the possible coset coding techniques that we can use – there are many more possible codes and metrics that we hope to explore in future work.

4. Extending the Lifetime of NAND Flash Memory

This chapter presents our coset code design and implementation to increase the lifetime of memory locations composed of NAND Flash cells. Flash is used in multiple products sold today such as memory cards [66], cell phones [67], and solid state drives (SSDs) [1]. One drawback of current Flash-based systems is that Flash cells become unable to change state after a certain number of writes. In this chapter, we present our coset coding technique to increase the number of Flash writes before failure and evaluate the effectiveness of coset coding in extending the lifetime of a single memory location. We also provide and evaluate a modified SSD design that incorporates coset coding during normal drive operation without requiring modifications to the host-drive interface.

This chapter presents our coset coding design, simulation results applying our coset coding technique to Flash, and our design/implementation of coset coding for Flash SSDs. Sections 4.1, 4.2, and 4.3 provide background on Flash, present related work, and present our design of coset coding for Flash. Section 4.4 discusses a system-level technique (stuck-at cell pointers [SCPs]) that we developed for use with coset coding to enhance its effectiveness. Section 4.5 presents our coset coding evaluation results. Using coset coding in a device that incorporates Flash chips can require changes to how the device operates on the Flash chips. An example of a class of devices that would require modifications to use coset coding is Flash SSDs sold today. Sections 4.6 and 4.7 present our modifications to an existing SSD design along with our simulation results for using coset coding to extend the lifetime of a Flash SSD. Section 4.8 concludes the chapter.

This chapter makes the following contributions:

- We demonstrate how to use coset coding to extend the lifetime of memory locations composed of Flash cells;
- We present two different metric functions designed for use with coset coding to extend the lifetime of Flash;
- We introduce stuck-at-cell pointers (SCPs), a scheme based on error correcting pointers (ECPs) [60] to futher extend the lifetime of memory locations composed of Flash cells when used with coset coding; and
- We present a design for implementing coset coding in a Flash SSD.

4.1 Flash Background

Flash cells become unable to change state after changing states a given number of times. Each new generation of Flash cells has supported fewer cell state changes than previous generations. Future generations of Flash cells are projected to support even fewer state changes before becoming unusable [22]. Sections 4.1.1 and 4.1.2 provide an overview of the organization of Flash cells in chips sold today and how data are stored in Flash cells. Section 4.1.3 discusses error correcting codes (ECCs) which are required when using Flash chips. Section 4.1.4 discusses how we extend the lifetime of memory locations using coset coding.

4.1.1 Flash Organization

As discussed in Chapter 3, memory locations with PCM cells can be changed to any value at the bit granularity; in contrast, memory locations with NAND Flash cells are read/written at the page granularity. Writing a page of Flash cells uses the *program* operation. After a Flash page is programmed and before a given cell can be re-written, all cells in a page must be *erased* (reset to a pre-defined level). Erases are performed at the *block* granularity (a block is defined as a unit composed of multiple pages, typically on the order of 64 pages (i.e., 256KB for a 4KB page) [45]). A program/erase (P/E) cycle consists of programming all of the pages in a block and then erasing the same block. Flash cells wear out due to P/E cycles and are eventually unable to change value [43].

4.1.2 Storing Data in Flash Cells

During a write, digital values (one or more bits) are converted into an amount of charge (an analog value) which is then stored in a Flash cell. Flipping a bit more times before an erase will result in more programs before an erase. The number of times a given bit can change value before an erase is required depends on how each digital value is mapped to an amount of charge.

The number of states in a Flash cell depends on the number of charge *levels* (discrete amounts of charge) assigned to the cell. The design of the Flash cell write controller determines the number of charge levels in a given Flash cell. Cells begin at level 0. During a write, the write controller increments the Flash cell an arbitrary number of levels using the program operation. During an erase, the write controller decrements the Flash cell back to level 0 using the erase operation.

Charge levels in a Flash cell can be mapped to any binary value. Figure 36 depicts two example mappings between a 4-level Flash cell and four different binary values. In the first mapping (Mapping 1), the cell stores two bits. In the second mapping (Mapping 2), the cell stores only a single bit. For example, with Mapping 1, L0 represents the value “00” while with Mapping 2, L0 represents the value “0”. In both mappings, the cell level is the same and only the interpretation of the cell level when read out changes.

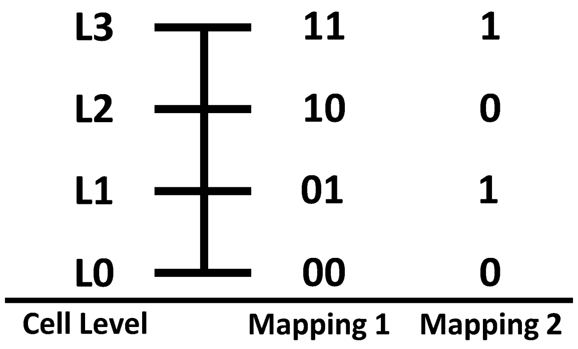


Figure 36: Example Mapping Between Cell Levels and Bits

Mappings with 2-bits per cells are currently used in Flash chips sold today as they offer the highest possible density of bit storage per Flash cell. Mapping 2 (a.k.a. W*aterfall coding* [36]) stores only a single bit per cell resulting in lower storage density. For example, Waterfall coding has half the storage density of Mapping 1 for a 4-level cell. The advantage of using Waterfall coding compared to Mapping 1 is that Waterfall coding has a higher ratio of re-programs to area overhead compared to Mapping 1. As explained in Section 4.5, Waterfall coding combined with coset coding is more area efficient at extending the lifetime of memory locations composed of Flash cells than Mapping 1 combined with coset coding.

We use *f* to denote the number of times a bit is guaranteed to be able to be flipped before the block the cell is in must be erased. In Figure 36, the bits in Mapping 1 have an $f = 1$, since each bit is guaranteed to only be able to flip once. The left bit flips between L1 and L2, and the right bit flips between any two levels. The right bit could flip up to 3 times if the left bit does not flip, but it is only guaranteed to be able to flip once before an erase (so $f = 1$). Waterfall coded bits for a 4-level cell each have $f = 3$ because each bit is guaranteed to be able to flip three times without an erase.

We can create bits where $f > 1$ from bits where $f = 1$ by combining multiple $f = 1$ bits into a single $f > 1$ logical bit. This is useful as Flash chips manufactured today have $f = 1$. Constructing logical bits in this way reduces storage density. For example, three $f = 1$ can hold 3 bits worth of data, but only 1 bit of data when used as a $f = 3$ bit. This trade-off can be worthwhile when writing un-coded data or combined with coset coding. High f bits allow for a more efficient implementation of coset coding with the trade-off of reducing storage density. Future SSDs have been proposed to support high f high cells using Waterfall coding [30] as it is more efficient at implementing bits with f > 1 than creating logical cells.

4.1.3 Flash SSDs and ECC

Devices that use Flash memory must provide a mechanism for tolerating errors in data stored in Flash cells [39]. Current Flash research shows that Flash cells require the ability to correct at least one error per 1024 cells [6]. The same research concluded that stronger error correction will most likely be required in the future.

Our technique provides both error correction and endurance benefits by incorporating an ECC into a coset code. Section 2.3.2 discussed how to create and use an *error correcting coset code* (ECCC). The key idea in an ECCC is to ensure that cosets consist solely of ECC codewords.

There are two considerations when choosing the specific ECC to use with coset coding. First, the choice of ECC determines how many errors can be corrected (e.g., SECDED) and how much storage overhead is required for error correction. Correcting more errors requires more storage overhead. Second, the ECC must be compatible with the code used for coset generation (Section

2.3.2). In the results presented in this chapter, we use a Hamming code as our ECC (which is compatible with the convolutional coset codes we evaluate [28]).

4.1.4 Wear-out Mechanism

Flash cells wear out due to P/E cycles. The objective in this chapter is to delay cell wear-out by re-programming memory locations composed of Flash cells without first erasing the memory location [30]. Re-programming pages without an intervening erase reduces the number of erases required for a given number of programs. The calculations in this section assume uniform random inputs where a write to any given cell is independent of the other cells and has the same write distribution as the other cells. This is the case in many modern SSDs due to the use of a scrambler [12].

Using current methods to write to Flash without coset coding, the probability of successfully re-programming one cell (P_{SROC}) is high. P_{SROC} is calculated using Equation (1), the ratio of the number of allowed transitions to the number of possible transitions*:*

$$P_{SROC} = \frac{\text{Number of Allowed Transitions}}{\text{Number of Possible Transitions}} \tag{1}$$

Table 19 lists the possible two-write combinations for a given Flash memory. There are four possible write combinations with the probability of each two-write combination calculated assuming uniform random inputs and two possible values (“0” or “1”). For random inputs, each two-write combination has a 25% chance of occurring. Of the four possible two-write combinations when re-programming Flash, only 1-to-0 is not allowed. Using Equation (1),

$P_{SROC} = \frac{3}{4}$ or 75%, i.e., there are three allowed transitions of four possible transitions for re-programming a cell once.

Table 19: Random Data Input Two-Write Probabilities For a Single Cell

Write 1	Write 2	Allowed?	Probability
0	0	Yes	25%
0	1	Yes	25%
1	0	No	25%
1	1	Yes	25%

In contrast to P_{SROC} for a single rewrite (75%), P_{SROP} without an erase operation is very small for un-coded data written to a 4KB page, the standard page size used in Flash chips produced today [42]. P_{SROP} is calculated by multiplying together P_{RSOC} for all memory cells in a page (Equation (2)):

$$P_{SROP} = P_{SROC}^{\{\text{Page Size in Bits}\}} \tag{2}$$

Using Equation (2), we calculate the P_{SROP} for successfully writing twice to a 4KB page is $(0.75)^{32,768}$ (a very small number). Coset coding increases P_{SROP} by providing multiple coset representatives to represent a single dataword and searching for the best coset representative to write using a metric function as discussed in Chapter 2.

4.2 Related Work

There has been prior work on both extending the lifetime of a single Flash memory location and extending the lifetime of Flash SSDs. Section 4.2.1 presents prior work on extending the lifetime of a single memory location. Section 4.2.2 presents prior work on system-level techniques for extending the lifetime of Flash SSDs.

4.2.1 Flash Location Lifetime Extension Prior Work

This section summarizes prior work on mitigating and tolerating the effects of Flash wear-out that is comparable to our work. Coset coding allows for re-writing of a page multiple times before an erase is required. Coding techniques that enable multiple writes to a page before having to erase are known as "multi-write codes" or "rewriting codes." We use coset coding as a multi-write code. In this section, we discuss previously developed multi-write codes.

Previously developed multi-write codes share the same high-level feature of coset coding of mapping a dataword to multiple possible codewords. *Enumerative coding* [29] maps a dataword to a set of codewords using the lexicographical ordering of the codewords (e.g., a dataword maps to all codewords with a given number of "1s").

A *linear write-once-memory (WOM) code* [15] decodes a codeword by taking its modulus which permits multiple codeword representations of the same dataword. (For example, if the modulus is 7, then codewords 3 and 10 both represent the dataword 3).

Floating codes [30] are algorithms for mapping *k*-bits of information onto *n* *q*-level cells. Floating codes guarantee *t*-bit rewrites for each set of *n*-bits. Many different floating codes have been proposed in prior literature [13][30][31][34]. An illustrative floating code configuration is k=2,n=4,q=2. This code maps two bits of information onto four Flash cells. With this configuration, a floating code by Jiang et al. [30] guarantees three re-programs of the 2-bits in any sequence of writes. Table 20 shows two possible 3-write sequences. Bit sequence 1 has an invalid Flash transition between writes 2 and 3. The right-most bit flips from 1->0 without an intervening erase. Jiang et al.'s floating code allows this transition by mapping the data onto the Flash cells so that only 0->1 transitions are performed even though the data itself has a 1->0 transition but still reading out the correct sequence. The second sequence flips only the right-most bit; again, this would not be possible with un-coded Flash. Jiang et al.'s floating code allows this sequence of writes.

Table 20: Floating Code Write Sequences

Write	Bit Sequence 1	Bit Sequence 2
Initial	00	00
1	01	01
2	11	00
3	10	01

What distinguishes our version of coset coding from prior multi-write codes is the ease of coset coding to implement and its quantitative superiority at achieving multiple writes per erase. Floating codes require a complex lookup process to perform both reads and writes. Linear WOM codes require calculations using integers as well as a very large modulo operation, both of which can be expensive to implement in hardware. Implementing coset coding requires only XOR operations and Viterbi search (for which many hardware designs have been proposed [32]). Enumerative codes only guarantee a single page re-write. Coset codes can re-write a given page multiple times.

4.2.2 Flash SSD Lifetime Extension Prior Work

There are a number of system-level techniques that are orthogonal and complementary to our work that can be used to extend SSD lifetime. We discuss our modifications to incorporate coset coding into an SSD in Section 4.6. Write buffering [26] consists of adding a write buffer in front of the SSD to coalesce multiple writes to the same datum and thus reduce the number of writes to the SSD itself. Deduplication [23] reduces the number pages that need to be written to the

SSD by storing data that is identical across multiple addresses only once. Reads to any of these addresses are performed from the same page.

4.3 Coset Code Design

We designed an ECCC to extend the lifetime of Flash. As discussed in Chapter 2, designing an ECCC consists of three parts: selecting the coset code, selecting the ECC, and determining how to select a coset representative to write.

This section discusses the coding aspects of this design. We use a convolutional code [38] as our coset code and a Hamming code [25] at the 1024-bit granularity as our ECC. We used the Viterbi-based coset rep selector discussed in Section 2.4.4 to search a given coset for a coset representative to write. Convolutional codes are easily searched by the Viterbi algorithm which makes the the combination of a convolutional coset code and a Viterbi-based coset rep selector well suited for coset coding. Section 4.3.1 presents the metric functions we used with coset coding to extend the lifetime of memory locations. Section 4.3.2 presents the write and read processes we developed for using coset coding with memory locations composed of Flash cells.

4.3.1 Metric Functions

We developed two metric functions to extend the lifetime of Flash memory for use with our Viterbi-based coset rep selector (Section 2.4.4). Sections 4.3.1.1, 4.3.1.2, and 4.3.1.3 present, respectively, the difference pieces of information used in the metric functions, a metric function previously used for PCM memory modified to work with Viterbi, and the metric function we designed specifically for Flash.

4.3.1.1 Flash Write and Cell Information Used in Our Metric Functions

Table 21 lists the information we used in our metric functions. We calculate the metric for a given coset representative one bit at a time. The index of the bit for which the metric is being calculated is indicated using the symbol i. We consider the following four pieces of information when constructing the metric functions: the coset label (c_i), the zero coset elements ($Z_{e,i}$), the number of writes done to a given Flash cell (w_i), and the number of times a bit can change value (flip) without an erase to the memory location (f).

Table 21: Information Used in the Metric Functions

Notation	Description
i	Bit index
c_i	Coset label bit i
e	Coset representative index
$z_{e,i}$	Zero coset element e bit i
w_i	Number of writes performed to Flash cell i since last erase
f	Number of times a given bit can change value without an erase

The coset representatives in the zero coset are defined by the coset code that is used. As shown in Table 21, we use the symbol e to denote the index of the coset representative in the zero coset.

We used two pieces of information on the cells themselves in our metric functions: the number of writes to each cell, and the number of times each bit can flip (f). The number of writes to a given Flash cell is obtained by performing a read before a write. The information on f is hard-wired into the encoder during manufacture time.

4.3.1.2 Metric Function BFR

Metric Function Bit Flip Reduction (BFR) reduces the number of cells that flip for each write; we used this metric function with PCM in Chapter 3 and then modified it to work with our Viterbi-based coset rep selector (Section 2.4.4) and an ECCC for Flash. BFR extends the number of re-programs to a given Flash page by reducing the impact of writing a coset representative compared to the un-coded dataword it represents. In the context of PCM, reducing the impact of a write consists of reducing the number of cells that flip per write. In the context of Flash, reducing the number of bit flips increases the likelihood of being able to re-write the page. We use BFR to reduce the number of cell flips per write compared to writing the corresponding un-coded dataword to maximize the number of page re-writes.

Metric Function BFR consists of the sum of $\delta(c_i, z_{e,i})$ functions as shown in Equation (3). The input to each $\delta(c_i, z_{e,i})$ function is the coset label and the bits of the zero coset representative(s) that are being searched. The sum of these delta functions assigns the lowest metric to the coset representative with the fewest number of "1s". When used with the translate coset, the selected coset representative will require the fewest bit flips to write.

$$\text{Metric} = \sum_{i=1}^{N} \delta(c_i, z_{e,i})$$

$$\delta(c_i, z_{e,i}) = \begin{cases} 1, c_i \neq z_{e,i} \\ 0, c_i = z_{e,i} \end{cases} \tag{3}$$

4.3.1.3 Metric Function BFR+SCI+WL

Metric Function BFR+SCI+WL is the metric function we developed specifically for re-programming pages of Flash memory without an intervening erase. This metric function uses three different techniques to maximize page re-writes: BFR, stuck-at cell interpolation (SCI), and wear-leveling (WL). We first discuss SCI and WL and then describe how Metric Function BFR+SCI+WL implements all three techniques.

SCI allows for additional page re-programs even when a subset of the bits in the page cannot change. After programming a page one or more times, cells become stuck-at and are no longer able to change value without an erase to the page. Since there are multiple options to choose from in a coset, we can select a coset representative to write that incorporates values of bits that no longer can change without an erase (*stuck-at bits*). SCI is the process of incorporating previous cell information from stuck-at bits into a new write.

WL is the process of selecting coset representatives so that the wear due to writes is evenly spread across a given page. Spreading out wear prevents writes from clustering in a given group of cells and reduces the rate at which any cell becomes stuck. We perform WL by preferring to write bits that have not been written over those that have been written. WL is the same as SCI when a written bit is also a stuck-at bit (i.e., when $f = 1$) but differs when $f > 1$.

There are trade-offs between WL and BFR. Maximizing WL may result in reduced BFR and vice versa. For example, writing each single cell in the page results in ideal WL, but no BFR. Alternatively, the highest possible BFR may result from writing only the first five cells, resulting in changes to each cell on every write (flipping the minimum number of bits). There is high BFR in this scenario but poor WL. We considered this trade-off when designing the most effective metric functions for Flash.

Metric Function BFR+SCI+WL, shown in Equation (4), uses both SCI and WL in addition to BFR to select a coset representative to write to memory. Metric Function BFR+SCI+WL uses the number of previous writes to memory (w) to perform both SCI and WL. SCI is performed by checking if $w_i = f$. If $w_i = f$, the cell cannot be re-written and we assign an infinite metric value "∞" to the cell. An infinite metric value prevents the search algorithm from selecting an un-writable coset representative unless there are no writable coset representatives left. WL is performed by reading the number of writes done previously to each cell and adding it to the metric for the coset representative. We select the coset representative with the lowest metric so a higher number of writes adds a higher value to the coset representative metric, making the coset representative metric less likely to be chosen. The addition of the "+ 1" to w_i ensures that when a coset representative is selected, cells that have been written to are less likely to be written than cells that have not previously been written to. Without the "+1", both a cell that has not been written to and a cell that will not be written to both have a metric value of "0". With the "+1", a cell that has not been written to previously has a metric value of "1", and not writing a cell has a metric value of "0".

$$\text{Metric} = \sum_{i=1}^{N} \delta\left(c_i, z_{e,i}\right)(\alpha_i + 1)$$
$$\alpha_i = \begin{cases} \infty, & w_i = f \\ w_i, & w_i < f \end{cases} \tag{4}$$

4.3.2 Write and Read Processes

Figure 37 and Figure 38 compare the write and read processes of the method currently in use (labeled as "standard write/read" on the figures) in Flash chips to the technique using coset coding (labeled as "coset coding write/read"). In these figures, elements of the read/write process for the current method are depicted in white boxes with black text, and coset coding elements are shown using black boxes with white text.

Figure 37 depicts both the standard and coset coding write processes. The standard write process has only one stage ("ECC Gen") before writing to memory that consists of generating the ECC codeword from the un-coded dataword and then writing the ECC codeword to disk. Writing with coset coding consists of four steps. After generating an ECC codeword from the dataword, the coset coding encoder produces a coset representative to write (discussed in Section 2.4). SCPs are then generated following the coset coding encode (technique discussed in subsequent Section 4.4). During SCP generation, cells are checked to assess whether any cells will be written beyond the maximum number of levels. If a cell is going to be written to too high of a level, it is replaced with an SCP. Future reads and writes will use the SCP instead of the original cell until the page is erased. Finally, data is stored to the cell using Waterfall coding. If the cell has two levels, there is no difference between what is stored now and what is stored using Waterfall coding. For cells with more levels, Waterfall coding stores only a single bit instead of multiple bits per cell.

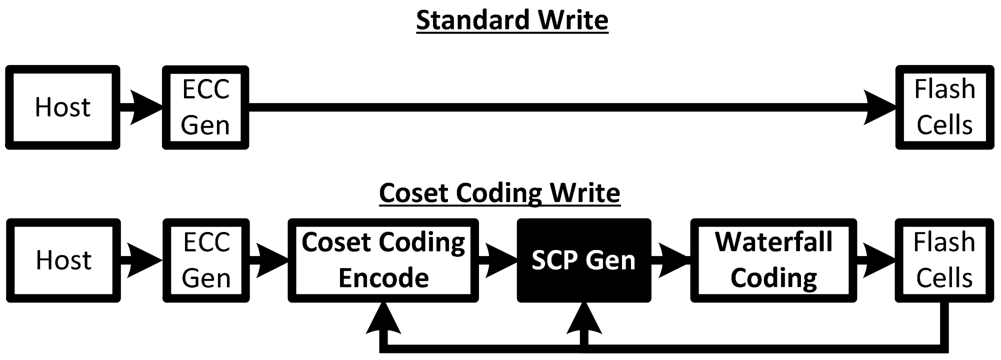


Figure 37: Coset Coding Write Process Compared to Write Process Used Today

Part of the coset coding write process requires searching the coset for the best coset representative to write. Our PCM coset sizes were relatively small allowing exhaustive search to provide a relatively fast method for searching the coset. For Flash, we opted to use the Viterbi search technique described in Section 2.4.4 since we used codes where the number of coset representatives is higher (the largest number of coset representatives we had for our PCM experiments was 256 in contrast to Flash where our experiments had 2^{512} coset representatives). For large cosets, Viterbi-based coset representative selection has lower hardware costs and faster search times than exhaustive search.

Figure 38 depicts both the standard and coset coding read processes. The standard read process has only one stage ("ECC Check") before reading out from memory. The standard read process first checks the read data for errors and then sends the data to the host. Reading with coset coding consists of three steps. First, the analog voltage levels are converted into binary values by reading each value modulo 2. Second, the read unit replaces values marked with an SCP pointer with SCP values. Finally, the coset coding read process checks the read data for errors and converts a coset coded representative (decodes) back into an un-coded dataword.

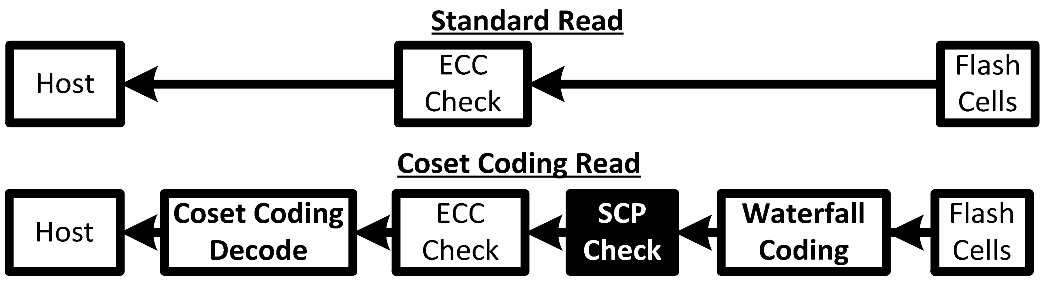


Figure 38: Coset Coding Read Process Compared to Standard Read Process Used Today

4.4 Stuck-At Cell Pointers (SCPs)

SCPs is a technique we developed that is similar to Error Correcting Pointers (ECPs) [54] to gain more writes from before an erase is required by providing replacement cells for a page. SCPs, illustrated in Figure 39, have two parts: an un-coded value used to indicate the location of the replaced Flash cell(s), and one or more replacement Flash cells. Each set of SCPs is stored with its corresponding page. SCPs replace the bit-value(s) of a given cell with bit value(s) of the replaced cell. Replaced bit values must be located consecutively in the Flash page for a given SCP. For example, 100 1-bit SCPs can be used to replace 100 randomly located 1-bit cells in a page. 100 2-bit SCPs can be used to replace 200 Flash cells, located in pairs of 2 in the page.

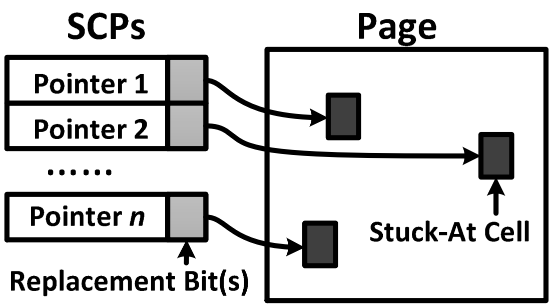


Figure 39: Stuck-At Cell Pointers

The area overhead of a SCP is log(*n*) + C, where *n* is the number of cells in a given page, and *C* is the number of replaced cells per SCP. Log(*n*) memory cells are used to store the value that points at the replacement cell. *C* cells are used as replacement cells for the stuck-at cells.

4.5 Flash Memory Location Lifetime Extension Evaluation

This section presents our experimental results for re-programming Flash pages using coset coding with random inputs and compares our results to related work. Section 4.5.1 presents the experimental methodology we used to obtain results. Section 4.5.2 lists the techniques we compared against, and Section 4.5.3 presents our results.

4.5.1 Evaluation Methodology

We performed coset coding experiments using parameters summarized in Table 22 for each metric function evaluated using a custom in-house simulator. We ran 100,000 different random inputs each with a distinct seed to simulate a distribution of writes to a 4KB page. We added ECC capabilities to our coset code by embedding a 128-state rate ½ convolutional coset code inside a Hamming code as described in Section 2.3.2. Each simulation run wrote to a single page and ended when the page was no longer able to be re-programmed.

Table 22: Experiment Setup

Simulator	In-House
Inputs	Random
Un-coded Page Size	4,096 Bytes
Error Correction	Hamming Code
Code Type	Convolutional
Code Rate	½
Number of States	128
Coded Page Size	8,448 Bytes

4.5.2 Techniques Compared

Table 23 lists the different techniques we compared in our evaluation of Flash multi-write codes. All evaluated techniques require the same number of Flash cells. We compared our coset coding technique with our two metric functions to uncoded writes and two prior work schemes: enumerative codes [29] and floating codes [30], both of which are described in Section 4.2.

Table 23: Page Re-Writing Techniques We Compared

Page Re-Writing Technique
Uncoded with Spare Cells
Coset Coding – Metric Function BFR
Coset Coding – Metric Function BFR+SCI+WL
Enumerative Coding
Floating Coding

For the uncoded write scheme, each cell is paired with a spare cell after the original cell is written the spare is used so that all schemes evaluated have the same area overhead. Therefore, the *f* value for bits with un-coded writes is double that of the other schemes (i.e., at *f* = 2, the bits for un-coded writes are *f* = 4).

We did not evaluate previously developed coding schemes where the writing scheme is a function of the number of writes. One example of a class of write-dependent coding schemes are *concatenated WOM codes* [35]. As with coset coding, a system implementation of concatenated WOM codes require that the number of writes to a given Flash cell or block of Flash cells be available at the encoder. For coset coding, this information is read from the Flash cell state whereas concatenated WOM codes cannot obtain this information from the previously written data. Using concatenated WOM coding requires recording the number of writes to a given Flash cell or block of Flash cells separately, resulting in a significant expansion of memory overhead. This cost is not considered in [35] and storing this information would obviate any gains from using the scheme. For these reasons, we have not included the performance of concatenated WOM coding in our evaluation.

Of the schemes we compared against, only our design of coset coding and 2x cells are designed to incorporate ECC. To the best of our knowledge, previously developed ECC techniques are not compatible with floating codes or enumerative coding. Writing to Flash requires error correction to protect against error causing phenomenon during Flash cell operation [6][21]. Since Flash requires error correction and neither floating or enumerative codes have an existing mechanism for correcting errors, new error correction techniques would need to be developed to use floating and enumerative codes with Flash.

4.5.3 Results

This section presents our lifetime extension evaluation of page re-writing schemes for a single page. We first provide our results and an analysis of each re-writing scheme without SCPs, and then provide results and analysis of each re-writing scheme when 100 SCPs are used.

Figure 40 presents results comparing the re-program gains of un-coded writes and floating codes (a previously developed technique described in Section 4.2.1) to coset coding with both Metric Function BFR and Metric Function BFR+SCI+WL. The x-axis has the number of cell levels. The y-axis has the number of page re-programs. We do not plot enumerative coding (another previously developed scheme described in Section 4.2.1) as enumerative coding is designed only for f = 1. Enumerative coding allows a single re-write of a given Flash page regardless of input.

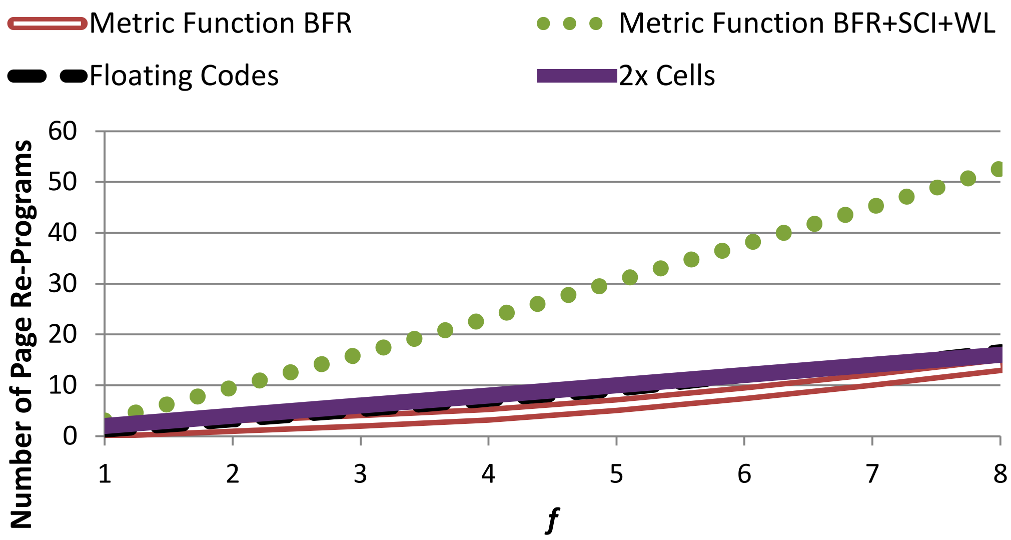


Figure 40: Single Page Re-Write Count for Random Data Writes Using Re-Write Schemes

***f* = 1.** We analyzed *f* = 1 separately from f > 1 as bits stored in cells manufactured today all have *f* = 1. We discuss below how coset coding with Metric Function BFR performed compared to prior work and present how well coset coding with Metric Function BFR+SCI+WL performed against prior work.

Coset coding with Metric Function BFR+SCI+WL resulted in a mean of 3 page re-programs before an erase was required. In comparison, enumerative coding provides 2 page-rewrites and floating codes only allow a single write to the page. Coset coding has a gain of 1.5x (3/2) over enumerative coding and a gain of 3x (3/1) over floating codes. Coset coding is superior to both enumerative and floating codes when *f* = 1 are used.

Metric Function BFR applied to Flash has lower lifetime than 2x cells with only a single page write. Metric Function BFR treats both the 0-to-1 and 1-to-0 transitions equally when determining which coset representative to write. PCM allows both transitions during the write process. When re-programming a page of Flash cells, changing the cell value from 0 to 1 is allowed, but changing the cell value from 1 to 0 is not allowed. We found that when re-writing the page using Metric Function BFR with *f* = 1, a coset representative was selected to be written that required on average 1.4% of the cells in the memory location to perform a 1-to-0 transition. Since 1-to-0 transitions were required, these coset representatives could not be written to the memory location without first erasing the page.

We found that not all the cells in the page were stuck-at after coset coding was unable to re-program the page, suggesting that there may be more effective metric functions. Figure 41 shows the number of writes to each cell when the page was unable to be re-written. Each bin of cells (i.e., cells with zero writes) was generated taking the average of 10,000 runs each with a distinct seed value. The number of writes to each Flash cell is on the x-axis, and the percentage of the Flash cells in the page that each bin contains is on the y-axis. For this metric function, 44% of cells could still be written, so there may be more possible page re-programs with a different metric function. Exploring improved metric functions for bits with *f* = 1 to increase page re-programs is a topic for future work.

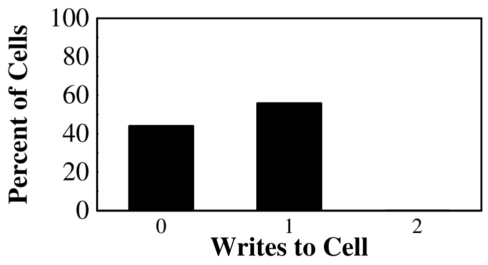


Figure 41: Cell Wear Distribution Before Erase Using Metric Function BFR+SCI+WL With *f* = 1

***f* > 1.** Metric Function BFR+SCI+WL achieves a 2.4x gain in re-programs at *f* = 2 and a 3.3x gain in re-programs at *f* = 8 over using 2x Flash cells. Metric Function BFR+SCI+WL achieves between a 3x and 3.4x gain over floating codes. Floating codes perform about the same as 2x cells with this given range of *f* values. With higher values of *f*, floating codes do show gains. With *f* = 23, floating codes provide 52-62 page re-programs compared to only 46 with 2x cells. Metric Fucntion BFR+SCI+WL has a lifetime gain between 3x and 5.7x higher than Metric Function BFR.

Figure 42 compares the re-programming gains of coset coding to 2x cells and floating codes when using SCPs. We do not plot enumerative coding as enumerative coding only guarantees two writes to a Flash page and does not benefit from SCPs. The x-axis has the *f* value for the bits in the page. The y-axis has the number of page re-programs.

We evaluated both coset coding and floating codes with 100 SCPs per page. These SCPs allow for a few more writes before having to erase a page. Figure 42 shows the results from these experiments. For coset coding, we used Metric Function BFR+SCI+WL. We do not show Metric Function BFR as we have demonstrated that Metric Function BFR+SCI+WL provides more page re-writes than Metric Function BFR at the same *f*. We did not evaluate enumerative coding with SCPs as enumerative coding is fixed at a single re-program. We also did not use SCPs for 2x cells as 100 SCPs are insufficient to improve the number of Flash re-programs.

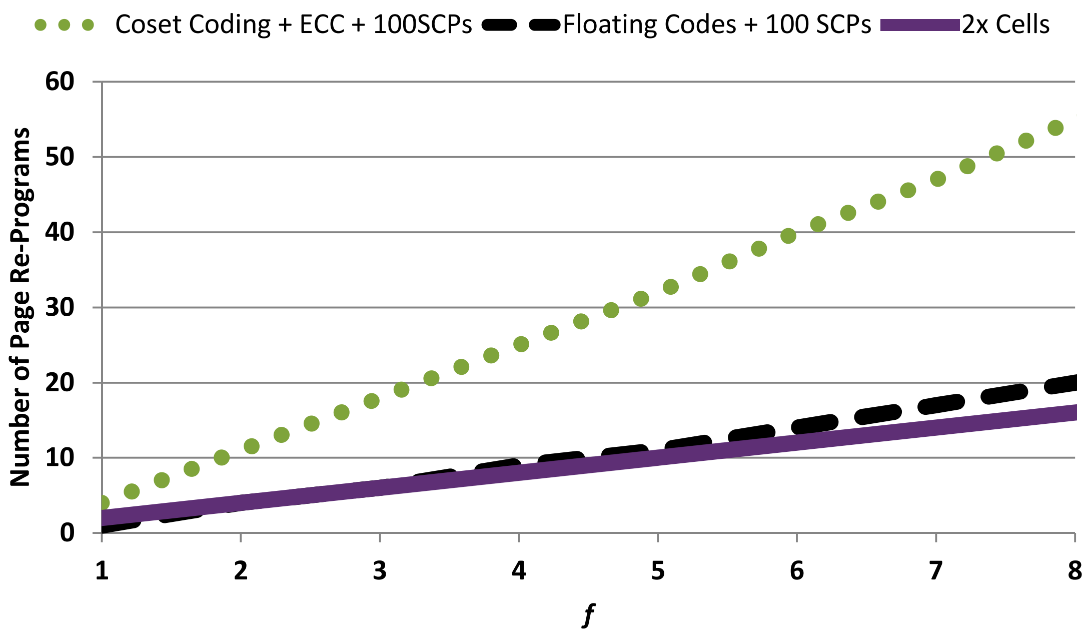


Figure 42: Single Page Re-Write Count for Random Data Writes Using Re-Write Schemes + 100 SCPs

Coset coding does better than both 2x cells and floating codes when combined with 100 SCPs. Floating codes with SCPs perform worse with for bits with *f* = 1, but do the same or better than 2x Flash cells for a larger number of cell levels. Coset coding re-program gains over un-coded writes and floating codes increase with the number of levels in the Flash cells. Coset coding with 100 SCPs has lifetime gains between 2x and 3.4x over un-coded, and 2.66x and 4x over floating codes with 100 SCPs. Coset coding will have the highest lifetime gains of the three schemes presented here when used in an SSD.

4.6 *Flash SSD Implementation Design*

We incorporated coset coding into a previously designed model of a SSD *Flash Translation Layer* (FTL) [2], a program that mediates the interactions between host commands and Flash cells.

Hard drives are being augmented and/or replaced by Flash SSDs as Flash SSDs are faster, more reliable, and use less power. In the following sections, we present the following proposed coset coding modifications to implement coset coding in the modules that are part of a typical FTL design:

1) *map table* that stores mappings from the input block numbers to the physical page locations on the drive (Section 4.6.1);
2) *garbage collector* that controls the erasing of Flash blocks (Section 4.6.2); and
3) *write controller* performs writes to pages in a solid state drive (Section 4.6.3).

4.6.1 Map Table

We modified the map table to store the metadata required to decode our coset encoded data. A map table in an SSD stores mappings from the addresses sent by the host to the physical page locations on the drive. To decode written data, the coset code decoder requires the *start state* of the convolutional code [38] used as a coset code. The number of bits in the start state depends upon the convolutional code that is used. Our coset encode/decode method requires storing the start state at the encode/decode granularity which in our experiments is 501 un-coded bits and 1024 coded bits. For our experiments, we used a convolutional code with a constraint length of 7 [32]. For a 4KB page, we have 66 independent encode/decodes. Storing the start states for all 66 encodes takes 462-bits or 0.6% of the coded page size (calculated as {number of independent encode/decodes}*{constraint length} = 66*7 = 462). Theoretically, we can reduce the number of start states required to record data down to a single start state. Storing only one start state would reduce the start state storage overhead to 0.01% of the coded page size. Reducing the number of start states required to be stored to decode data is future work.

Table 24 shows the three fields in a coset code enabled SSD map table entry: the *logical block address* (LBA), the *physical page number* (PPN), and start state. The LBA (first field) is the number indicating the memory location on which the host operates. The PPN (second field) is the *physical page number* (PPN) that uniquely identifies the physical page where the SSD stores data. The LBA and PPN fields are also present in SSDs manufactured today. The start state (third field, described above) is a binary value generated during the coset coding write process and used during the read process (7 bits for our experiments).

Table 24: Fields in a Map Table Entry

LBA	PPN	Start State

The FTL accesses the map table during reads and writes to the SSD by the host. SSDs sold today use the map table to translate between the LBA and the PPN. A write command consists of the host sending an LBA to the SSD along with the data to write. Internally, the SSD stores data in a location indicated by the PPN of the page. After the piece of data is written, the PPN where the piece of data is stored is recorded along with the LBA in the map table. With coset coding, the start state is also stored. A read command consists of the host sending an LBA to the SSD. The map table is used to locate the PPN where the data at the LBA is stored then the data is read and returned the host. With coset coding, the start state is read during the read command and used to decode the read-out data before sending it back to the host.

4.6.2 Garbage Collector

In this section, we describe how we modified the garbage collector to support coset coding. We present both the design of a standard FTL garbage collector and our modifications to support coset coding.

Standard FTL garbage collector. Figure 43 shows the cycle that blocks go through in a standard garbage collector. Each block is classified as either *Clean* (no pages have been written since the block was last erased), *Active* (currently being used for writes), or *Sealed* (holds data and cannot be written). Each page is classified as either *Clean* (unwritten), *Valid* (holds the current value of a datum), or *Stale* (holds an old value for a datum that was subsequently re-programmed). Each write presented to the SSD is written to a Clean page in the Active block so that the page becomes Valid.

Initially in an SSD, a Clean block is selected to become Active. Data from write commands sent by the host to the SSD are then stored in the Active block in what is termed “out-of-place” writing. If there was a previously written data, the page with that data becomes Stale. When every page in a block is written (either Valid or Stale), the block is marked as Sealed and can no longer be written. At this point, the SSD selects a Clean block to be the new Active block. To make a block Clean, the block is erased. Blocks are erased when the size of the free pool drops below a pre-defined threshold. When a block is erased, it and all the pages in the block are marked as Clean.

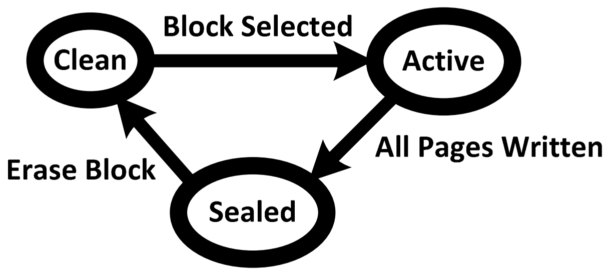


Figure 43: Standard Garbage Collection

Coset coding modified FTL garbage collector. We modified the garbage collector so that it does not automatically erase a block when transitioning a block from Sealed to Clean. Instead, the block is only erased when a given fraction of the pages in the block have failed to be re-programmed by the write controller. Below this threshold, the FTL *eraselessly cleans* the block, i.e., the FTL marks the block as Clean without altering the contents of the block. The FTL can eraselessly clean a block instead of the erasing the block because coset coding allows for multiple re-writes of a page without erasing the page first. Eraselessly cleaning a block does not require page moves or an erase of the block. We maximize eraseless cleans and minimize full cleans of a block to minimize write amplification and maximize block lifetime. We present and discuss write amplification for our experiments using both standard and coset coded writes in Section 4.7.3.

Figure 44 shows the garbage collection process when using coset coding in an SSD. Two of the transitions are the same as a standard SSD. Blocks are marked as Active from Clean when selected to be written to by the drive controller. Once all pages in the block are written to, the block transitions from Active to Sealed.

We modified the process of transitioning from Sealed to Clean to use coset coding to re-write pages without an intervening erase. An un-writeable page threshold is used to determine whether the block is erased or eraselessly cleaned. Above the threshold the block is erased, below the threshold the block is eraselessly cleaned. The lower the un-rewritable page threshold, the fewer times a block is eraselessly cleaned before being fully cleaned. The higher the un-rewritable page threshold, the less capacity the block holds and the write controller has a higher likelihood of having to spend longer to find a page to write. After a block is completely erased, all pages became writable again.

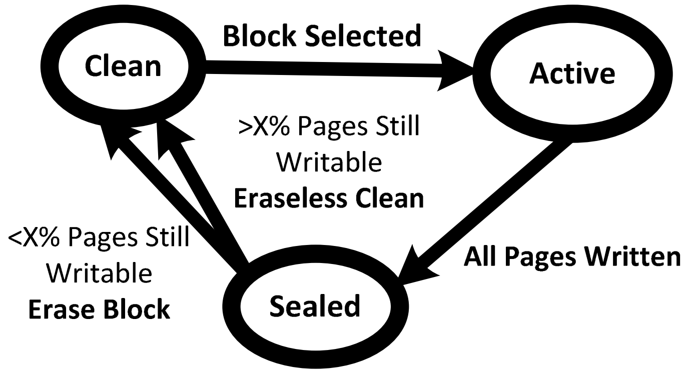


Figure 44: Coset Coding Garbage Collection

Because eraselessly cleaning blocks does not reclaim un-writable pages, a fraction of the pages in a block can be un-writeable resulting in lower capacity of a SSD until these pages are erased. When designing a coset code enabled SSD, the SSD must have the minimum advertised capacity at all times to ensure data are written successfully to the Flash. We designed the process shown on Figure 45 to meet this requirement.

After writing to a page, we check if the drive capacity is below the advertised capacity. If the drive capacity is below the advertised capacity after a write to a page, the garbage collector erases blocks until the drive is equal to or greater than the advertised drive capacity and the free pool is back to its maximum size. During this erase process, the garbage collector selects blocks to erase that maximize the number of Sealed pages per erase (versus minimizing the number of Valid pages per erase which is what it does when refilling the free pool normally).

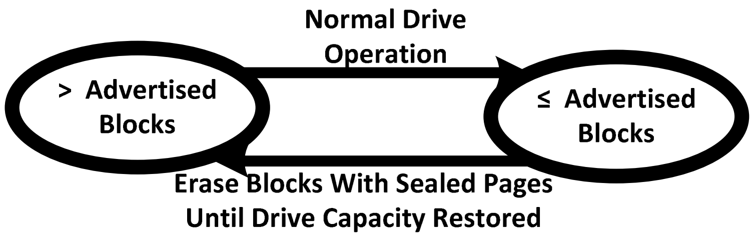

Normal Drive Operation
> Advertised Blocks
≤ Advertised Blocks
Erase Blocks With Sealed Pages Until Drive Capacity Restored

Figure 45: Ensuring Sufficient Blocks are Available in the Drive

To reduce *write amplification* (wear due to internal page moves for a write operation), we do not move Valid pages when eraselessly cleaning a block. Instead, Valid pages are kept in the block. The only exception is if a block contains only used pages (Sealed or Valid). In this case, the block selected to be cleaned is fully erased. If the best block selected to erase has only used pages (i.e., Valid or Sealed), it is necessary to reclaim those pages. Eraseless cleans do not reclaim Sealed pages or move out Valid pages. Rather, if it is necessary to reclaim Valid and Sealed pages, the garbage collector fully erases the block. Fully erasing the block moves all Valid data out of the block resulting in all pages in the block reclaimed.

4.6.3 Write Controller

A traditional SSD write does not require circuitry to retry after a failed write because the block is erased before being written; when using coset coding, a block is not erased before re-programming and write failure can occur. To address these failures, coset coding requires adding circuitry to attempt writing data until a successful write occurs. The following compares standard SSD writes to a write controller that uses coset coding.

Standard SSD writes. As shown in Figure 46, a standard SSD writes to pages left-to-right top-to-bottom across a block. In this example, all pages are either written all zeroes or all ones. As discussed previously, Flash cannot execute a 1-to-0 transition for a given Flash cell during a program operation. The figure depicts four page writes to a block in sequence 1,1,0,1. Since the block is erased before the sequence of writes, all four writes succeed. Writes proceed in the following order: the first write is to the upper left page, then the upper right, then the lower left, and finally the lower right.

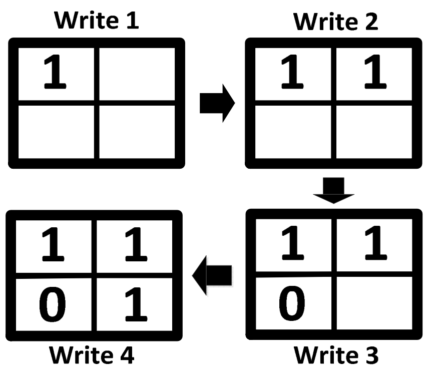


Figure 46: Standard Write Process

Coset enabled FTL. We modified the write controller to tolerate failed writes and mark a page when it is no longer writable. After failed write, the write controller then retries writing to Flash until the write succeeds. Figure 47 depicts the write process for a coset coding enabled drive. Since the block is not erased before each write, there is "stale" data in the block (depicted in the figure in light grey). Writing new data (1,1,0) consists of over-writing stale data as shown in sequence across the block from left, to upper right, and to bottom. Since the first two page write transitions are 0-to-1 the first two page writes are successful. The third page write requires a 1-to-0 transition. Since this transition is not allowed, the third write fails. The un-writeable page is marked as Sealed and the same write is attempted on the lower-right page. Since writing the lower-right requires a 0-to-0 transition, and a 0-to-0 transition can be written without an erase, the write is allowed and the page of data is written. Once the lower-right page is written, the block is marked as Sealed and a new block is selected to be written to for future writes.

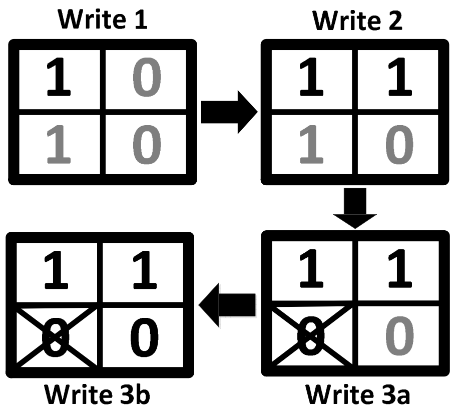


Figure 47: Coset Coding Write Process

4.7 *Flash SSD Implementation Evaluation*

This section presents our evaluation of the system level effects when using coset coding in an SSD. Section 4.7.1 presents the methodology we used to evaluate coset coding in the context of

SSDs. Section 4.7.2 discusses over-provisioning, a technique for increasing drive lifetime by adding additional blocks to a drive. Section 4.7.3 presents our results.

4.7.1 Methodology

Our experimental methodology consisted of selecting a suite of benchmarks to run, selecting a simulator and configuring it to emulate writing data to a Flash SSD, sizing the number blocks for each simulation, simulating data written, determining the timeframe for each simulation, and the metric we use to evaluate the results.

Benchmark Selection. We used the MSR Cambridge [47] set of benchmarks as inputs to simulate a variety of SSD write patterns. Table 25 lists the different benchmarks used as well as the function of the server from which each benchmark was generated. These functions represent possible workloads in which a Flash SSD that might be used. For each benchmark, we simulated the disk activity of the first volume of 12 of the workload types.

Table 25: Benchmark Information

Server	Function	Server	Function
usr	Home directories	src2	Source control
proj	Project directories	stg	Web staging
prn	Print server	ts	Terminal server
hm	HW monitoring	web	Web/SQL server
rsrch	Research projects	mds	Median server
prxy	Firewall/web proxy	wdev	Test web server

Simulator Selection/Configuration. We used an in-house functional simulator based on DiskSim to evaluate the system level effects of coset coding. Table 26 presents a summary of the simulator configuration. We configured the simulator to mimic a standard Flash SSD. We sized the drive to have a number of blocks in the drive based on the benchmark run (see Table 27).

We took the parameters for the number of pages per block and the size of each page from a Micron NAND Flash datasheet [42]. Each Flash block contains 256 pages and each page is 4KB. Blocks are cleaned when the number of clean blocks in the system drops below 5% and blocks are cleaned until 15% of the drive is free again. Our garbage collector randomly selects a block to clean from the set of blocks with the fewest active pages.

Table 26: Simulator Parameters

Parameter	Value
Number of Blocks	Based on Benchmark (See Below)
Pages Per Block	256
Un-coded Page Size	4,096 Bytes
Coded Page Size	8,448 Bytes
Free Pool Size	5% (Min) 15% (Max) of Drive
Garbage Collection	Random Selection from the Set of Blocks with the Fewest Active Pages

Simulating Data Written. We emulated data writes in our system simulation. We assumed all data that are written are random due to the use of a scrambler [12] and generated a distribution of the number of possible page re-writes to a given page by running page-level simulations as described in Section 4.5 for each bit *f* value. We simulated at least 100,000 random data page re-programs for each simulation and used the distribution of page re-programs in the system simulator to determine how many re-programs each page could sustain.

Timeframe for each Simulation. We ran each benchmark for writes over a three-year period. Each benchmark contains input representative of a week of disk activity. We assumed that the disk activity from week to week was the same. To simulate three years of activity, we simulated the same week of disk activity from a given benchmark until three years of disk activity had passed. After the simulator completed three years of writes, we took the maximum number of erases to the blocks in the drive as the lifetime required of all the blocks to sustain this many writes.

Table 27 lists the different benchmarks grouped by data size. We sized the drive differently for each benchmark grouping. The largest drive size was prn, then proj and hm together, and finally the remainder of the benchmarks.

Table 27: Benchmarks Grouped By Data Footprint Size

Large Drive		***Med. Drive***		***Small Drive***	
Drive Size	12.39GB	**Drive Size**	1.65GB	**Drive Size**	0.72GB
Benchmark	**Data Size (GB)**	**Benchmark**	**Data Size (GB)**	**Benchmark**	**Data Size (GB)**
prn	12.39	proj	1.65	web	0.72
		hm	1.63	prxy	0.71
				usr	0.65
				ts	0.54
				src2	0.50
				stg	0.39
				wdev	0.34
				Mds	0.33
				rsrch	0.29

Lifetime Metric. Our lifetime metric is the number of erases each cell must support at a given area overhead for the drive to last three years. To calculate this metric, we recorded the number of erases to each block during the operation of the drive. After three years of writes had occurred, we took the maximum number of erases to all blocks as the required number of erases that all cells must support.

4.7.2 Over-Provisioning

We used over-provisioning in conjunction with coset coding in our evaluation in to extend the lifetime of a SSD. Over-provisioning increases lifetime of a Flash SSD by adding more blocks in the drive than are externally visible to the user. Increasing the amount of over-provisioning consists of adding Flash cells without providing any additional capacity to the end-user. These extra blocks enable an SSD to perform more efficiently, tolerate failed blocks, and extend the lifetime of the SSD.

4.7.3 Results

The following presents our results that compare lifetime required when using SSDs for two different Flash cell types in production currently (2LCs and 4LCs). We compare writing methods used with a SSD produced today with our coset coding enhanced SSD writing methods in terms of lifetime required for a given amount of area overhead. We measure required lifetime in terms of the number of erases required from cells in the drive for a given amount of storage. Section 4.7.3.1 presents results for 2LCs, and Section 4.7.3.2 presents results for 4LCs.

4.7.3.1 2LCs

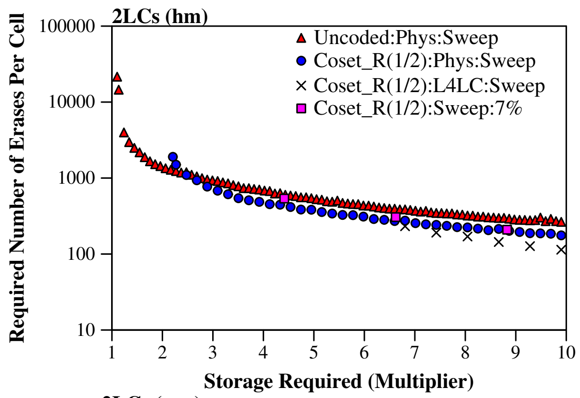
We evaluated four schemes listed in Table 28 on 2LC Flash cells to determine the effectiveness of each scheme in extending the lifetime of a SSD. All schemes were given a Hamming(1024,1013) [25] code for error correction. Each scheme is defined by three different parameters. For each scheme a tuple is given that identifies the scheme in the graphs below. The first parameter is the page re-write code. Since we showed that coset coding with Metric Function BFR+SCI+WL is better than prior work in terms of re-write gains, we only evaluated our coset coding page re-write scheme with this metric function. The second parameter is how bits are stored in the cell. For 2LCs, bits were stored either using the physical mapping used in Flash chips sold today or using logical LCs constructed using the method discussed in Section 4.1.2. Un-coded (U) and 1F both use the physical mapping, L3F uses Logical 4LCs, and FS sweeps this parameter. The third parameter is the amount of over-provisioning. U, 1F, and L3F all sweep this parameter while FS has it fixed at 7% [68].

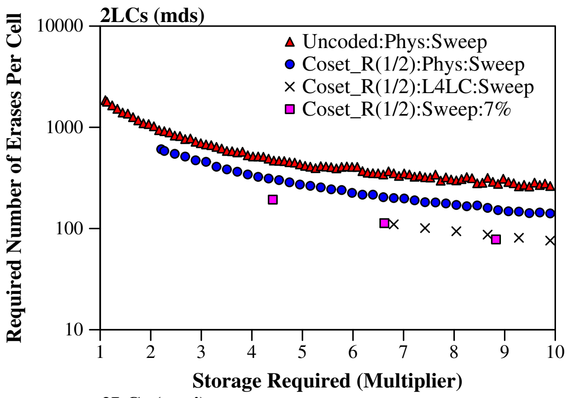
Table 28: 2LC Schemes Evaluated

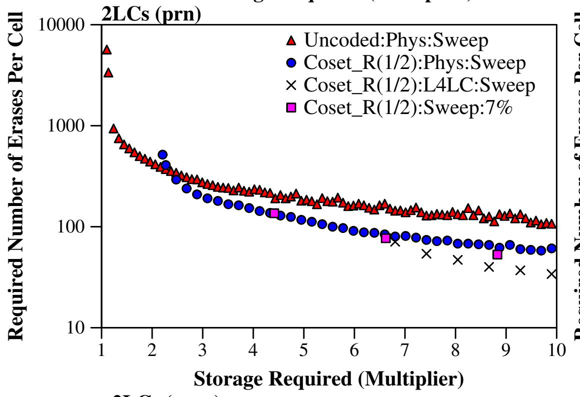
Abbrev	Page Re-Write Code	Bit Mapping	Over-Provisioning	Tuple
U	Un-coded	Physical (*f* = 1)	Sweep	Uncoded:Phys:Sweep
1F	Rate ½ Conv Code	Physical (*f* = 1)	Sweep	Coset_R(1/2):Phys:Sweep
L3F	Rate ½ Conv Code	Logical 4LC (*f* = 3)	Sweep	Coset_R(1/2):L4LC:Sweep
FS	Rate ½ Conv Code	Sweep	7%	Coset_R(1/2):Sweep:7%

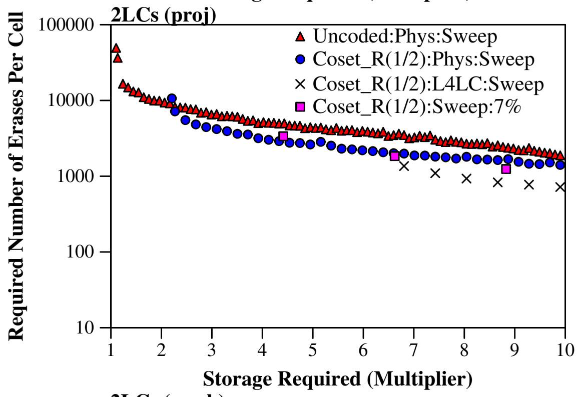
We evaluate lifetime extension for the schemes listed in Table 28 using two types of graphs. The first graph represents our lifetime metric (discussed in Section 4.7.1) where a lower number of required erases corresponds to longer drive lifetime. The second graph depicts how many erases each block must support for a given amount of drive storage to achieve three years of drive lifetime. This graph evaluates lifetime extension among the four schemes by graphing the storage multiplier (x-axis) against the number of erases each cell is required to support (y-axis). For the storage multiplier, a 1x storage multiplier equates to the drive having the advertised storage to the user, a 2x multiplier equates to the drive having twice the advertised storage, and so forth.

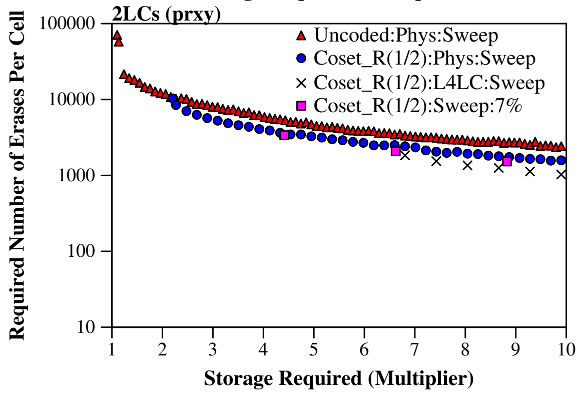
Figure 48 shows our lifetime extention graphs using 2LCs. Our lifetime extension graphs have y-axis ranges grouped into two categories in order to make it easier to distinguish between the different write methods plotted. Table 29 lists the benchmarks in each group. All benchmarks have a y-axis minimum of 10 erases required per block. prn, mds, rsrch, src1, src2, stg, ts, usr, and wdev have a y-axis maximum of 10,000 erases required per block; and hm, web, proj, and prxy has a y-axis maximum of 100,000 erases required per block.

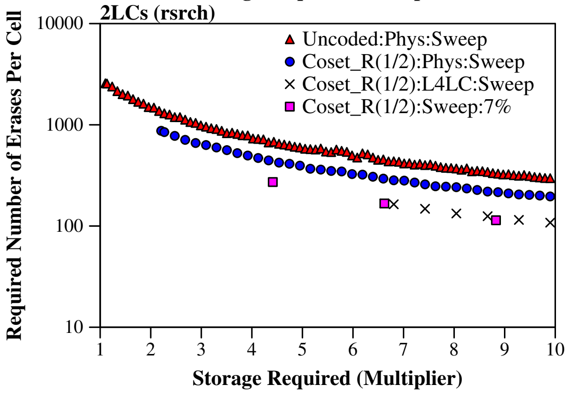












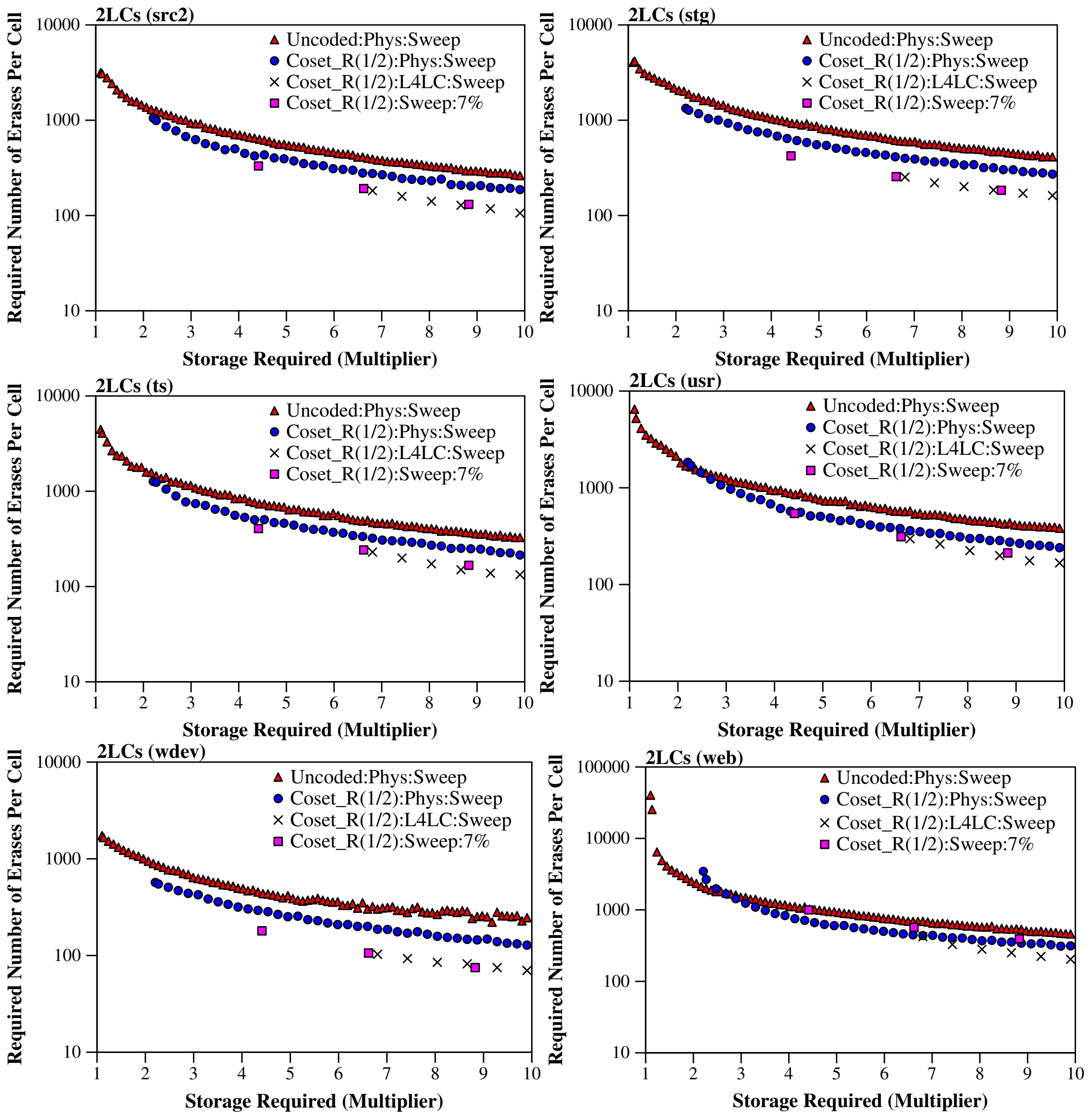
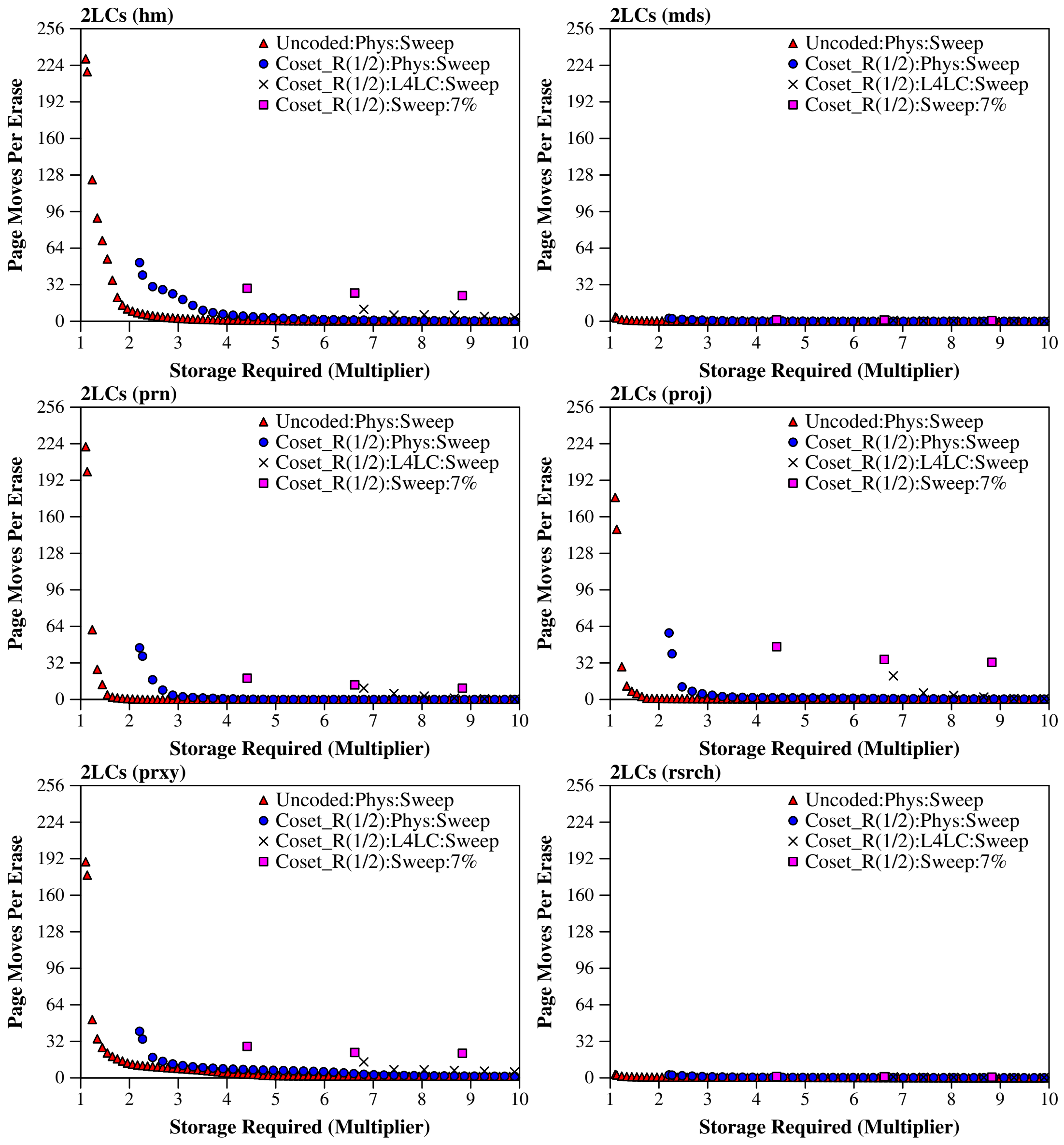


Figure 48: Required Number of Erases Per Cell (2LCs)

Table 29: Lifetime Graphs Y-Axis Min and Max Values (Erases Required Per Block)

Benchmarks	Y-Axis Min	Y-Axis Max
prn,mds,rsrch,src1,src2,stg,ts,usr,wdev	10	10,000
hm,web, proj,prxy	10	100,000

Figure 49 depicts for 2LCs the measured write amplification in the drive among the four schemes as the average number of page moves per erase by graphing the storage multiplier (x-axis) against the average number of page moves per erase operation (y-axis). As discussed in Section 4.6.2, our version of coset coding reduces write-amplification but does not always eliminate write amplification. Write amplification occurs due to pages that need to be moved out of blocks when the blocks are erased.



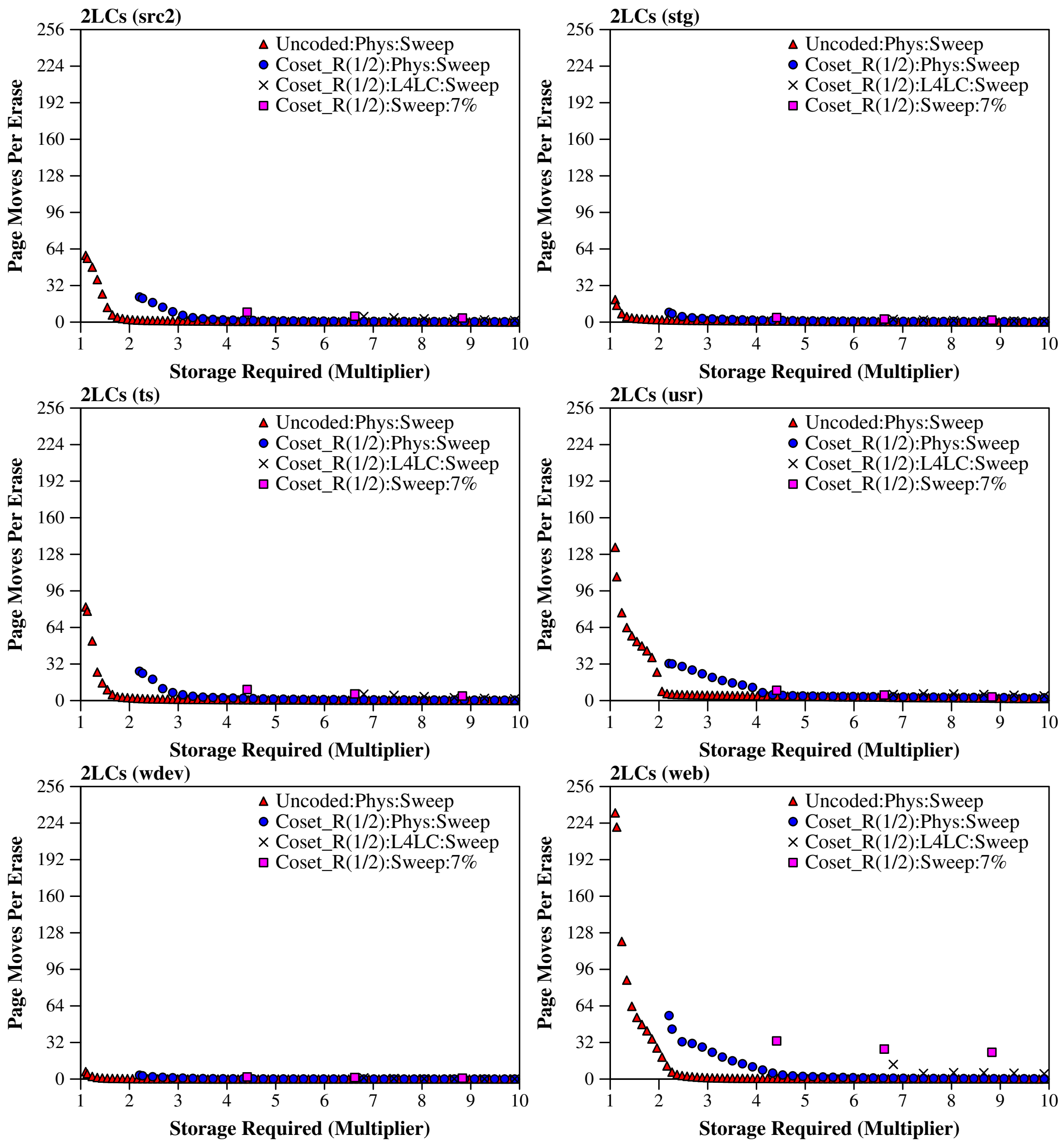


Figure 49: Average Number of Page Moves Per Erase (2LCs)

We will discuss results from two benchmarks, hm and mds, which are representative of the behavior of the twelve benchmarks.

hm benchmark. 1F for the hm benchmark performs worse than U when write amplification is over 40.4 page moves per erase (over 2.27 storage overhead) for 1F. At 2.27 storage overhead, the number of required erases is 1,236 for U and 1,499 for 1F. Due to the write amplification of

1F being 33.93 page moves per erase higher than that of U, there are more writes for 1F than U even though the number of writes performed by the host is the same for both. Since 1F has more writes, it also has more erases than U even though 1F requires fewer erases for a given number of writes. Once sufficient storage is provided for write amplification to decrease to 30.3 page moves per erase for 1F, 1F has a lower required erase count than U. L3F requires fewer erases per cell than U due to a combination of the high number of page re-writes (16.19) and low write amplification (maximum of 24.7 page moves per erase). L3F requires between 151 and 156 fewer erases per cell than U. FS only reduces write amplification from 28.8 at 4.41 overhead to 22.45 at 8.83 overhead. Even with this high write amplification, FS still requires fewer erases per cell as than U. FS requires 537 erases per cell at 4.41 overhead, while U requires 602 erases per cell at 4.43 overhead.

mds benchmark. Since write amplification is close to zero, 1F requires between 111 and 332 fewer erases per cell than U. Both L3F and FS also require fewer erases per cell than U as well due to the high number of page re-writes (on average 16.19 for L3F; see Figure 48 for FS) and the fact that there is no write amplification. FS performs on par with L3F with a range of 180-241 erases required per cell lower than U for L3F and 198-276 lower than U for FS[1].

4.7.3.2 4LCs

Table 30 lists the different evaluated schemes. Figure 50 and Figure 51 show the lifetime extension and write amplification for each scheme respectively. As with 2LCs, each scheme has three parameters as well as the tuple label used on the graphs. U and 1F are the same schemes

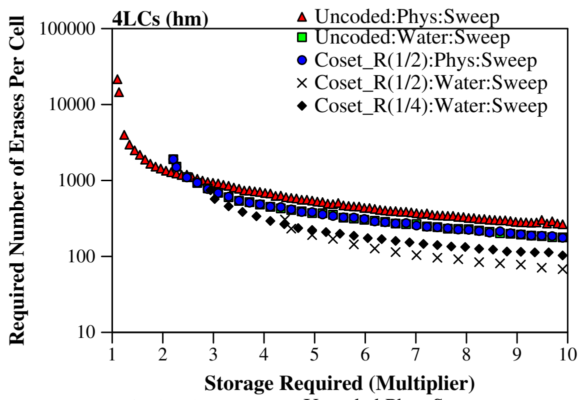
[1] Due to the discrete nature of our data points, the area overhead for each data point for FS and U respectively are not the same. We calculated the difference in erases required per cell between FS and U ±0.05 storage overhead (i.e., one difference was calculated with at 8.82 storage required for FS and 8.87 storage required for U).

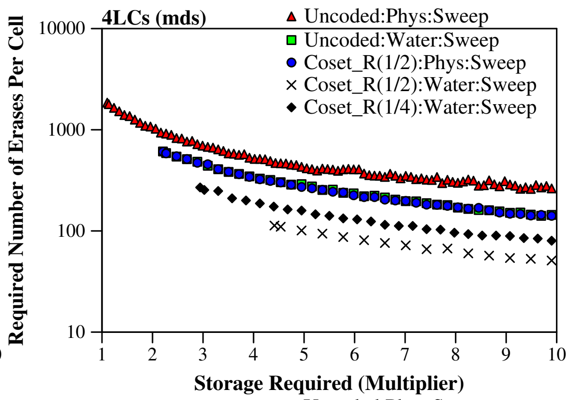
used for 2LCs as listed in Table 28 but with their area overheads adjusted for 4LCs. W uses Waterfall coding (discussed in Section 4.1.2) to store un-coded bit values. We used Waterfall coding to allow a given 4LC to store a single *f* = 3 bit. Both $3F_{R½}$ and $3F_{R¼}$ use Waterfall coding to store coset coded bit values.

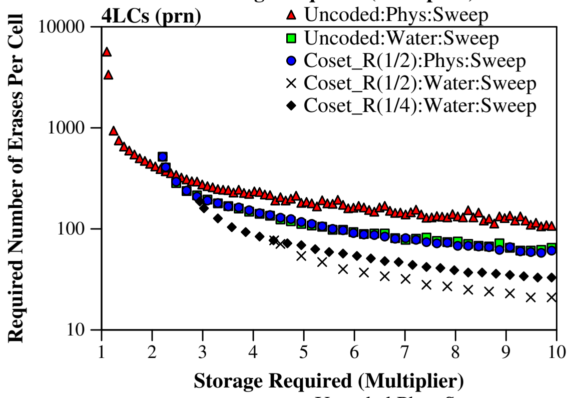
Table 30: 4LC Schemes Evaluated

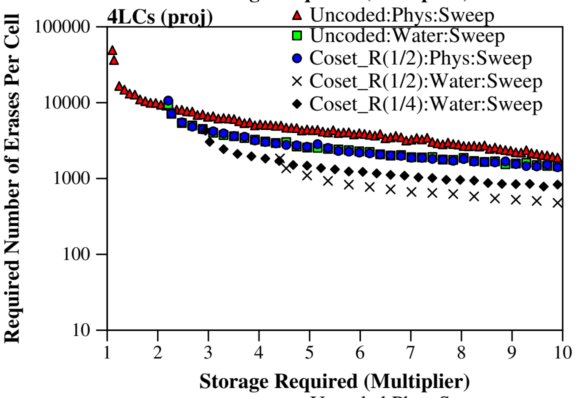
Abbrev	Writing Code	Bit Mapping	Over-Provisioning	Tuple
U	Un-coded	Physical (*f* = 1)	Sweep	Uncoded:Phys:Sweep
W	Un-coded	Waterfall (*f* = 3)	Sweep	Coset_R(1/2):Water:Sweep
1F	Rate ½ Conv Code	Physical (*f* = 1)	Sweep	Coset_R(1/2):Phys:Sweep
$3F_{R½}$	Rate ½ Conv Code	Waterfall (*f* = 3)	Sweep	Coset_R(1/2):Water:Sweep
$3F_{R¼}$	Rate ¼ Conv Code	Waterfall (*f* = 3)	Sweep	Coset_R(1/4):Water:Sweep

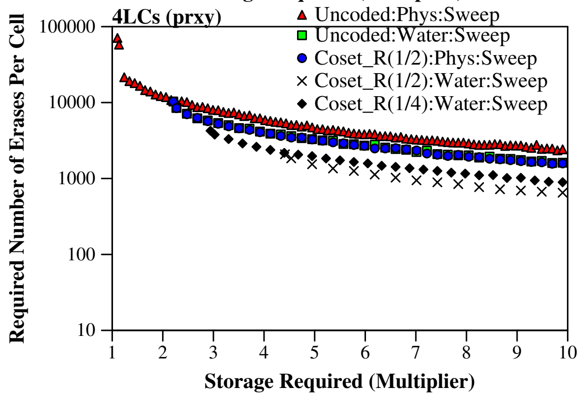
For our 4LC analysis below, we evaluate lifetime extension for two coset coding schemes (1F and $3F_{R½}$) and the U and W schemes by graphing the storage multiplier (x-axis) against the number of erases each cell is required to support (y-axis) as well as the write amplification for the hm and prxy benchmarks. As with 2LCs, we selected these benchmarks to depict two types of behavior of the different write schemes relative to each other. We present results using the same two types of graphs as our 2LC analysis (required number of erases per cell for the prxy benchmark and the write amplification).

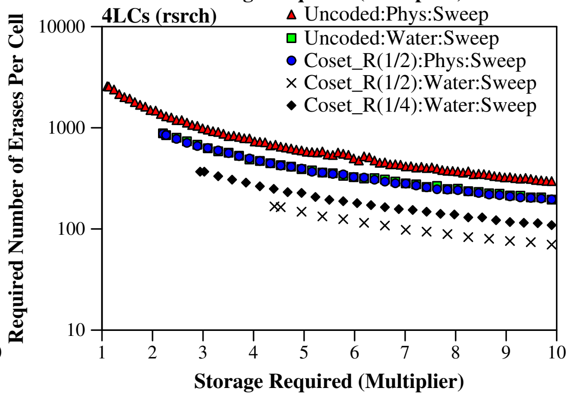

4LCs (hm)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Required Number of Erases Per Cell
Storage Required (Multiplier)
100000
10000
1000
100
10
1
2
3
4
5
6
7
8
9
10


4LCs (mds)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Required Number of Erases Per Cell
Storage Required (Multiplier)
10000
1000
100
10
1
2
3
4
5
6
7
8
9
10


4LCs (prn)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Required Number of Erases Per Cell
Storage Required (Multiplier)
10000
1000
100
10
1
2
3
4
5
6
7
8
9
10


4LCs (proj)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Required Number of Erases Per Cell
Storage Required (Multiplier)
100000
10000
1000
100
10
1
2
3
4
5
6
7
8
9
10


4LCs (prxy)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Required Number of Erases Per Cell
Storage Required (Multiplier)
100000
10000
1000
100
10
1
2
3
4
5
6
7
8
9
10


4LCs (rsrch)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Required Number of Erases Per Cell
Storage Required (Multiplier)
10000
1000
100
10
1
2
3
4
5
6
7
8
9
10

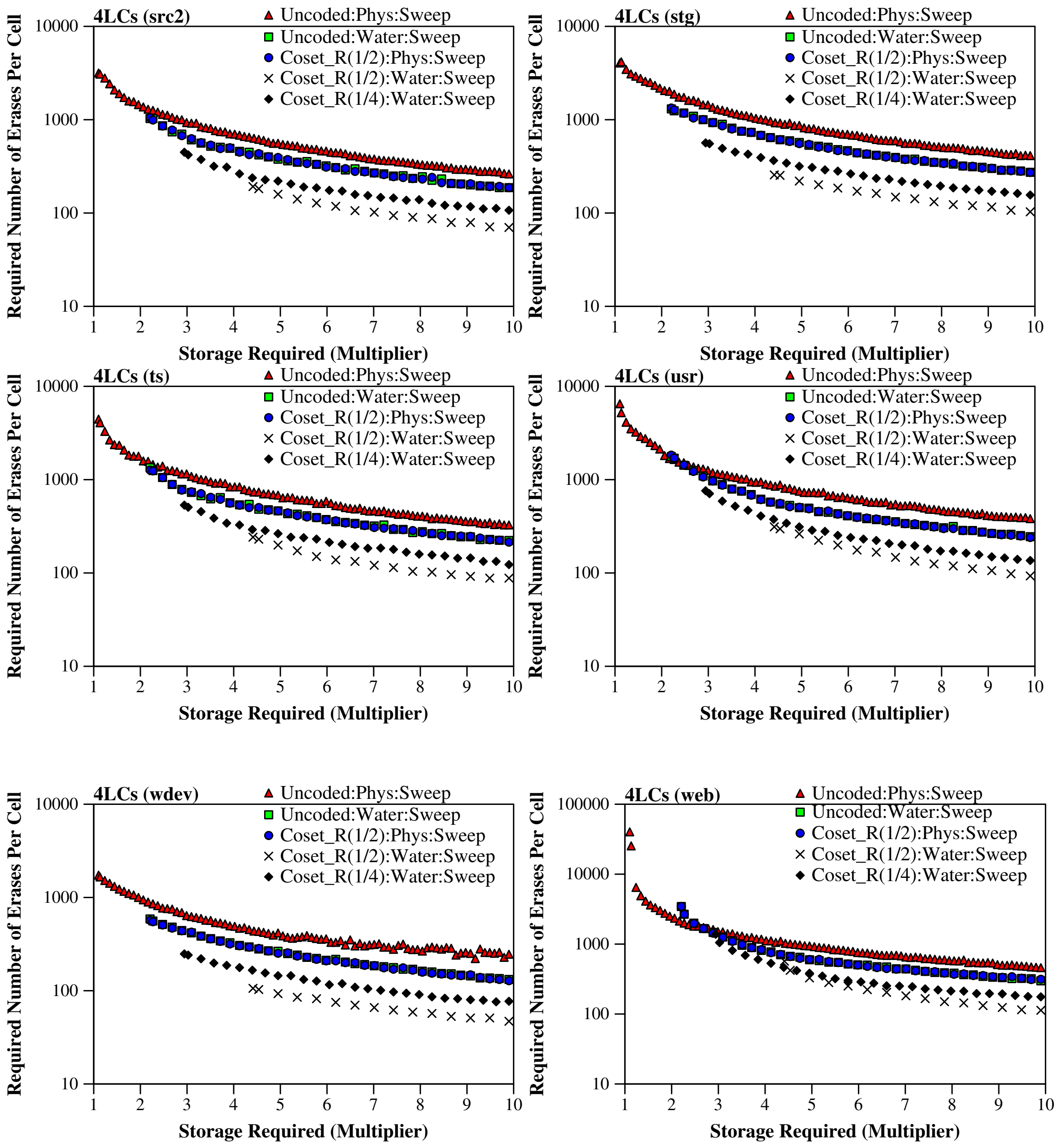
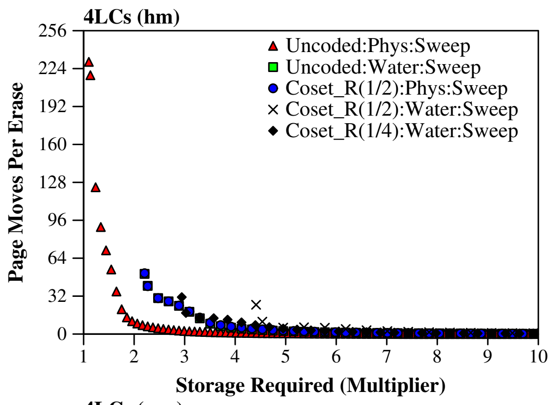
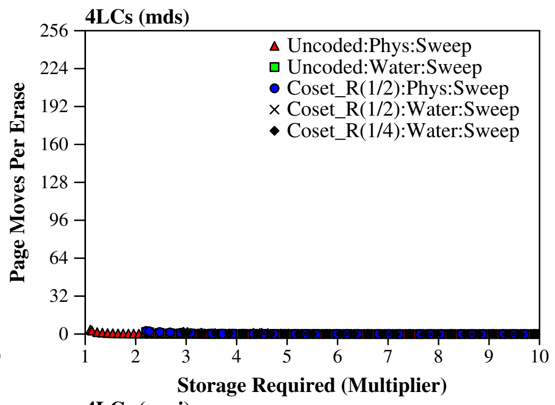
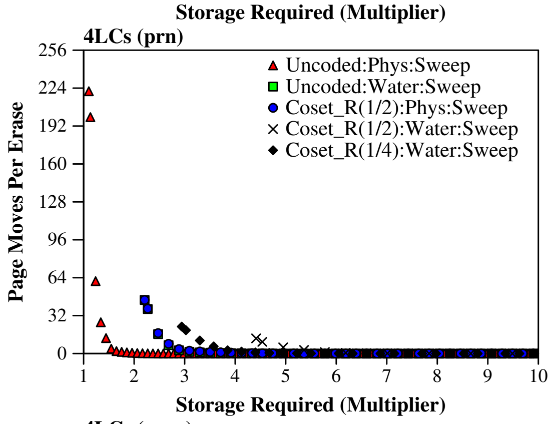
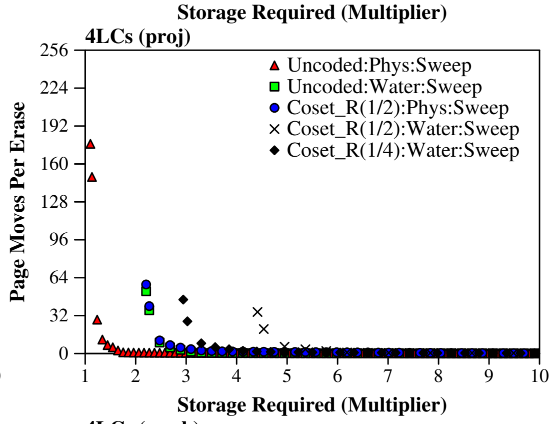


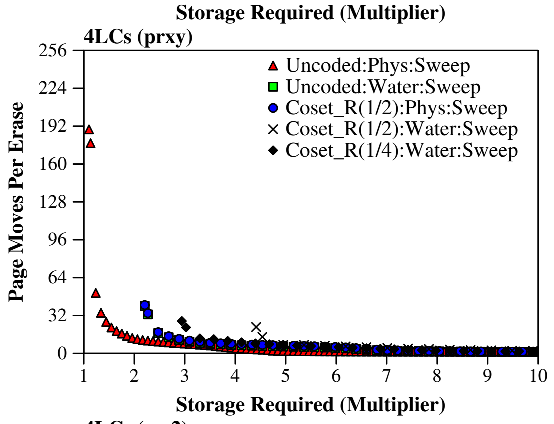
Figure 50: Required Number of Erases Per Cell (4LCs)

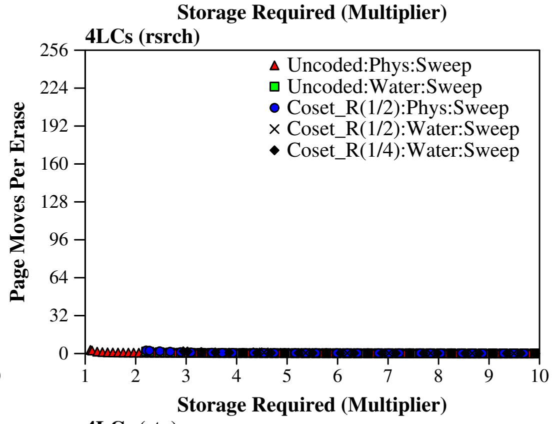

4LCs (hm)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)

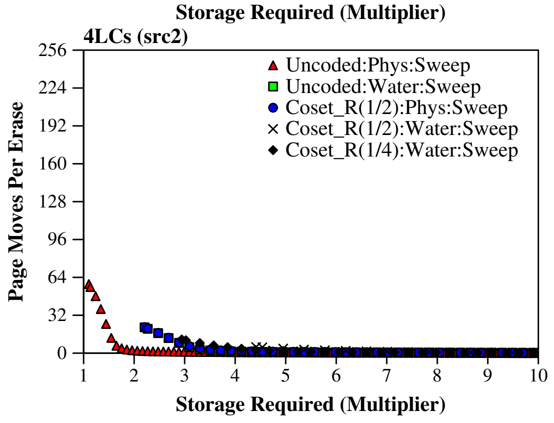

4LCs (mds)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)

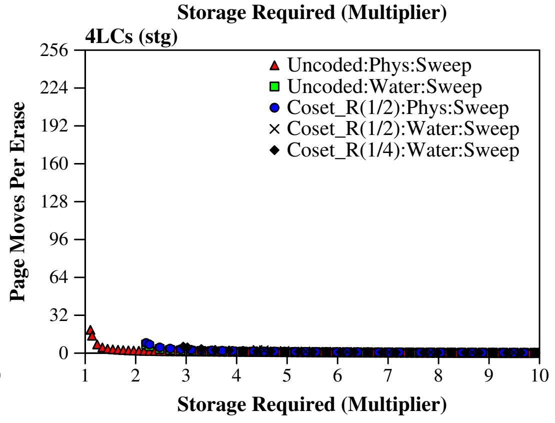

4LCs (prn)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)


4LCs (proj)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)


4LCs (prxy)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)


4LCs (rsrch)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)


4LCs (src2)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)


4LCs (stg)
Uncoded:Phys:Sweep
Uncoded:Water:Sweep
Coset_R(1/2):Phys:Sweep
Coset_R(1/2):Water:Sweep
Coset_R(1/4):Water:Sweep
Page Moves Per Erase
Storage Required (Multiplier)

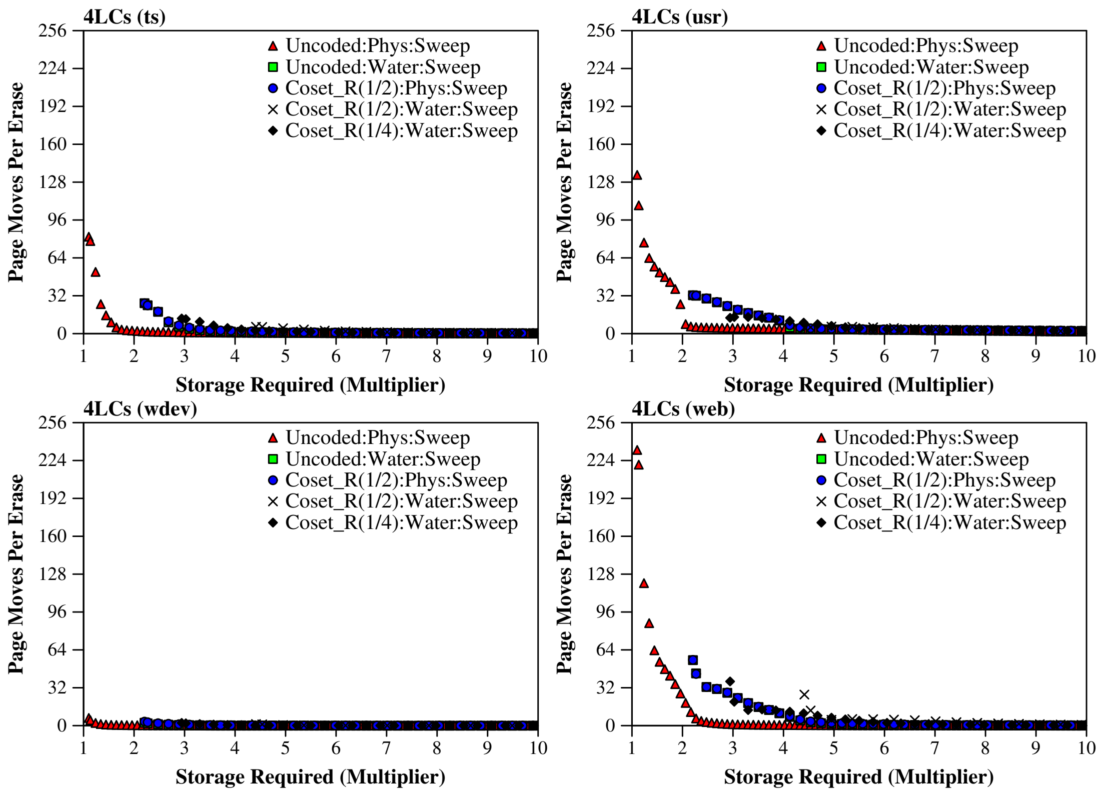


Figure 51: Average Number of Page Moves Per Erase (4LCs)

We will discuss results from two benchmarks, hm and prxy, that exhibit representative behavior of the benchmarks evaluated.

hm benchmark. The results for 1F and U are identical with 2LCs and 4LCs since in both cases bits have f = 1 and the relative overheads are the same. W and 1F perform similarly in terms of lifetime gain because both can perform approximately three page writes before requiring an erase. $3F_{R½}$ performs better than the other schemes but has the highest minimum required overhead of 4.41. $3F_{R½}$ requires between 197 and 357 fewer erases per cell than U. $3F_{R¼}$ requires fewer erases than 1F, W and U at the same area overhead, but more than $3F_{R½}$. $3F_{R¼}$ has a minimum storage multiplier 1.47 lower than $3F_{R½}$.

prxy benchmark. $3F_{R½}$ performs the best in terms of lifetime gain of the schemes evaluated for prxy. $3F_{R½}$ requires between 1,771 and 3,216 fewer erases than U. As with hm, W and 1F perform similarly in terms of lifetime gain for the evaluated benchmarks**.** Both schemes have about three re-programs before erase, and both schemes use the same mechanism for reducing write amplification. Both W and 1F are almost identical – the largest difference in the required number of erases is 297 at 6.19 storage overhead. 1F has between 817 and 3,128 fewer erases required per cell than U. W requires 748 to 3,012 fewer erases than U. $3F_{R¼}$ does better than 1F with 673 to 1567 fewer erases required for the same overhead, but worse than $3F_{R½,}$ with 243 to 429.

$3F_{R½}$ erase required range is 1,771-3,216 lower than U for 4LCs compared to 1,394-1,599 lower than U using L3F with 2LCs. This is due to the higher efficiency of using Waterfall coding compared to logical cells. $3F_{R½}$ requires three 2LCs since each bit stored in a 2LC is only f = 1. Each 4LC can store two f = 1 or one f = 3, a loss of one cell instead of two cells. Since $3F_{R½}$ on 4LCs is more efficient than L3F on 2LCs, the overhead for $3F_{R½}$ is lower at a given amount of over-provisioning than that of L3F, resulting in a higher difference in erases per cell required between $3F_{R½}$ and U than L3F and U.

4.8 Conclusion

Flash SSDs are quite attractive as replacing hard drives for enterprise use, but their limited lifetimes are a potential obstacle. In this chapter, we have shown how to apply theoretical work in coding theory to extend the lifetime of Flash SSDs. Furthermore, we show that the use of coset coding can help to mitigate or even eliminate the write amplification that causes excess wear to the underlying Flash cells in an SSD. Implementing coset coding requires only modest changes to SSDs currently in production, and these changes are isolated to the SSD controller.

5. Conclusions

Coset coding is a more efficient method of extending memory lifetime than writing uncoded data and adding redundant memory cells for both PCM and Flash. This dissertation presented coset coding-based techniques to extend the lifetime of PCM and Flash and provided examples of implementations for both memory types. Unlike current methods used to write PCM and Flash memory that only allow for a 1-to-1 mapping between a dataword and a set of codewords, our proposed coset coding writing technique allows writing a representative codeword from a number of possibilities. We use this flexibility to minimize wear due to writes to a given memory location and to extend the lifetime of memory locations composed of write-limited memory cells.

PCM. Our coset coding technique extends the lifetime of memory locations composed of PCM cells by reducing the number of bits that flip per write. The fewer bits that flip per write the more times we can write to the same memory location. We also presented a technique that allows writes to the memory location even after a subset of bits cannot change value, provided example hardware implementations of our coset code encoder and decoder, and evaluated hardware performance in terms of area, energy, and delay costs. The energy and delay costs for the encoder and decoder are under 0.2% for our PCM datasheet and we believe the area overheads to be negligible.

NAND Flash cells. We increased lifetime of memory locations composed of NAND Flash cells by using coset coding to re-write pages so that we reduce the number of times a page needs to be erased for a set number of writes. To re-write pages, we developed a metric function to use with coset coding that uses three different techniques (BFR, WL, and SCI) to maximize the likelihood after a given write that the next write to the page will be successful. We evaluated our technique against previously developed work for re-writing pages and found that we achieved between 1.5x and 3.3x more page re-writes than uncoded. To further increase the efficiency of coset coding at re-writing pages, we used SCPs (a technique based on ECPs) to replace a few cells in a page that can no longer be re-written. Coset coding with the experimental setup from Section 4.5.1 has lifetime gains between 2x and 3.4x over un-coded with 100 SCPs compared to 1.5x to 3.3x without SCPs for *f* values ranging from 1 to 8.

Integrating coset coding into Flash SSDs. We showed how to integrate coset coding into an FTL that can be used in Flash SSDs. Our implementation requires neither a significant number of changes to the existing SSD infrastructure nor changes to the external host interface. We modified the map table, garbage collector, and write controller of a previously developed SSD model to allow for eraselessly cleaning blocks and demonstrated in simulations that using coset coding increases the lifetime of both 2LC and 4LC SSD write schemes as compared with writing uncoded data.

Future work. Areas for future research include refining methods (e.g., coset code selection, search algorithms, and system integration) to optimally construct cosets and pick what to write using coset coding; work on applying coset coding to other areas of non-volatile memory design such as using coset coding as a scrambler or for security purposes; and layering multiple codes for optimal coset code design.

Coset code selection. We demonstrated that our methods compared well against current techniques, but we did not evaluate whether the the coset codes we used are optimal. Future research could evaluate new coset codes and/or using other existing codes to increase memory location lifetime. Future research might also evaluate using coset codes with less area overhead than the code that we used. We give results for a coset code with 100% area overhead; however, other codes exist that can be used for coset coding that have lower area overhead. Lower area overhead means fewer coset representatives per coset and therefore may result in lower lifetime gains.

Metric functions. We used a metric function that incorporated BFR, SCI, and WL to re-write pages. For f = 1, we found that 44% of cells were not written to when a coset representative to re-write the page could not be found. Improved metric functions may allow for more re-writes of the page. We can also design metric functions to optimize lifetime in other ways such as using the lifetime of each individual cell and adaptively writing with the goal that all cells in the page fail simulateously.

System integration. There is also work to do on the system issues of coset coding integration such as analyzing write-retry rates, reducing metadata storage overheads, and ensuring compatibility with FTL-less drives that use a Flash filesystem. SSD FTLs are used to allow SSDs to be used as a drop-in replacement for hard drives. Future SSDs and other Flash systems will use industry standard protocols for Flash access such as NVMe [48] or specialized protocols such as DC Express [58]. These protocols do not abstract away the underlying Flash memory as does an FTL. Exposing Flash to the operating systems may allow more efficient Flash management than what is used currently by FTLs. Exposing Flash to the operating system also creates new issues for integrating coset coding such as increased write-retry latency. Future SSDs that do not have an FTL may require a different implementation of coset coding than the one we proposed to mitigate the effects of these issues.

New applications. Coset coding can be used to enhance non-volatile memory in other ways than just improving endurance. For example, coset coding could be used as a replacement for a scrambler. One such adaptation has been proposed [61] but has yet to be refined and evaluated. Another possible use for coset coding is to improve security. We designed metric functions to select repersentatives to write that maximize lifetime; instead, we could design a metric function to select coset representatives that prevent write attacks designed to wear-out a subset of cells.

Layering codes. In this dissertation, we proposed using coset coding with either Waterfall coding or logical cells to increase the number of cell levels. However, other codes could be used to increase the number of times a bit can be flipped. For example, the floating code technique discussed in Section 4.2.1 could be combined with coset coding to create bit with high *f* values.

Future Flash technology nodes will require advanced coding techniques such as coset coding to deal with endurance and other physical issues to allow for the endurance required for use in computing systems. As Flash memory shrinks, its endurance also decreases. While our version of coset coding may be improved and optimized, we have shown it to be effective method to increase the lifetime of non-volatile memories compared to prior work and writing uncoded data.

References

[1] J. S. D. F. 20 and 2014 20 Comments, “SSD vs. HDD: What’s the Difference?,” *PCMAG*. [Online]. Available: http://www.pcmag.com/article2/0,2817,2404258,00.asp. [Accessed: 23-Jun-2014].

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, “Design Tradeoffs for SSD Performance,” in *USENIX 2008 Annual Technical Conference*, Berkeley, CA, USA, 2008, pp. 57–70.

[3] R. Ahlswede and Z. Zhang, “Coding for Write-Efficient Memory,” *Information and Computation*, vol. 83, no. 1, pp. 80–97, Oct. 1989.

[4] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, “Onyx: a protoype phase change memory storage array,” in *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, 2011, pp. 2–2.

[5] D. G. Andersen and S. Swanson, “Rethinking Flash in the Data Center,” *IEEE Micro*, vol. 30, no. 4, pp. 52–54, Jul. 2010.

[6] M. Bagatin, S. Gerardin, and A. Paccagnella, “Alpha-Induced Soft Errors in Floating Gate Flash Memories,” *IEEE International Reliability Physics Symposium (IRPS)*, 2012.

[7] T. Berger, *Rate Distortion Theory: Mathematical Basis for Data Compression*. Englewood Cliffs, N.J: Prentice Hall, 1971.

[8] N. Binkert et al., “The Gem5 Simulator,” *International Symposium on Computer Architecture (ISCA)*, Aug. 2011.

[9] G. W. Burr et al., “Phase change memory technology,” *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures*, vol. 28, no. 2, p. 223, 2010.

[10] A. R. Calderbank, P. C. Fishburn, and A. Rabinovich, “Covering properties of convolutional codes and associated lattices,” *IEEE Transactions on Information Theory*, vol. 41, no. 3, pp. 732–746, May 1995.

[11] A. R. Calderbank and N. J. A. Sloane, “New Trellis Codes Based on Lattices and Cosets,” *IEEE Trans. Inf. Theor.*, vol. 33, pp. 177–195, Mar. 1987.

[12] C. Chen and C. Kuan, “Encoding flash memory data with a randomizer using different seeds for different sectors,” United States Patent 8719491, 06-May-2014.

[13] F. Chierichetti, H. Finucane, Z. Liu, and M. Mitzenmacher, “Designing Floating Codes for Expected Performance,” *IEEE Transactions on Information Theory*, vol. 56, no. 3, pp. 968–978, Mar. 2010.

[14] S. Cho and H. Lee, “Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2009, pp. 347–357.

[15] G. Cohen, P. Godlewski, and F. Merkx, “Linear binary code for write-once memories (Corresp.),” *IEEE Transactions on Information Theory*, vol. 32, no. 5, pp. 697 – 700, Sep. 1986.

[16] T. M. Cover, “Enumerative source encoding,” *IEEE Transactions on Information Theory*, vol. 19, no. 1, pp. 73–77, Jan. 1973.

[17] D. S. Dummit and R. M. Foote, *Abstract Algebra, 3rd Edition*, 3rd ed. Wiley, 2003.

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

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

[20] J. Forney, G.D., “The viterbi algorithm,” *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, Mar. 1973.

[21] S. Gerardin et al., “Scaling trends of neutron effects in MLC NAND Flash memories,” in *2010 IEEE International Reliability Physics Symposium (IRPS)*, 2010, pp. 400–406.

[22] L. M. Grupp, J. D. Davis, and S. Swanson, “The bleak future of NAND flash memory,” in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012, pp. 2–2.

[23] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, “Leveraging Value Locality in Optimizing NAND Flash-Based SSDs,” in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2011, pp. 7–7.

[24] A. Hadoop, *Apache Hadoop*. 2011.

[25] R. W. Hamming, “Error detecting and error correcting codes,” *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[26] J. Hu, H. Jiang, L. Tian, and L. Xu, “PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD,” in *2010 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010, pp. 69 –78.

[27] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, “Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2010, pp. 3–14.

[28] A. N. Jacobvitz, A. R. Calderbank, and D. J. Sorin, “Writing Cosets of a Convolutional Code to Increase the Lifetime of Flash Memory,” in *Proceedings of the 50th Annual Allerton Conference on Communication, Control, and Computing*, 2012.

[29] A. Jagmohan, M. Franceschini, and L. Lastras, “Write Amplification Reduction in NAND Flash Through Multi-Write Coding,” in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–6.

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

[31] A. Jiang and J. Bruck, “Joint coding for flash memory storage,” in *IEEE International Symposium on Information Theory, 2008. ISIT 2008*, 2008, pp. 1741–1745.

[32] J. Jin and C.-Y. Tsui, “A Low Power Viterbi Decoder Implementation using Scarce State Transition and Path Pruning Scheme for High Throughput Wireless Applications,” in *2006 International Symposium on Low Power Electronics and Design (ISLPED)*, 2006, pp. 406 – 411.

[33] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, “Energy- and Endurance-Aware Design of Phase Change Memory Caches,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, 3001 Leuven, Belgium, Belgium, 2010, pp. 136–141.

[34] H. Kamabe, “Floating codes with good average performance,” in *2010 International Symposium on Information Theory and its Applications (ISITA)*, 2010, pp. 867–872.

[35] S. Kayser, E. Yaakobi, P. H. Siegel, A. Vardy, and J. K. Wolf, “Multiple-write WOM-codes,” in *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*, 2010, pp. 1062 –1068.

[36] L. A. Lastras-Montaño, M. Franceschini, T. Mittelholzer, J. Karidis, and M. Wegman, “On the Lifetime of Multilevel Memories,” in *2009 IEEE International Symposium on Information Theory (ISIT)*, Piscataway, NJ, USA, 2009, pp. 1224–1228.

[37] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting Phase Change Memory as a Scalable DRAM Alternative,” in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009, pp. 2–13.

[38] S. Lin and D. J. Costello, Jr, *Error Control Coding*, 2nd ed. Pearson Prentice Hall, 2004.

[39] Marina Mariano, “ECC Options for Improving NAND Flash Memory Reliability.” 2012.

[40] R. Melhem, R. Maddah, and S. Cho, “RDIS: A Recursively Defined Invertible Set Scheme to Tolerate Multiple Stuck-At Faults in Resistive Memory,” in *2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1 –12.

[41] Micron Technology, Inc., "P8P Parallel Phase Change Memory (PCM) NP8P128A13B1760E." Micron, 2005.

[42] Micron Technology, Inc., "NAND Flash Memory MT29F32G08CBACA, MT29F64G08CEACA, MT29F64G08CFACA, MT29F128G08CXACA, MT29F64G08CECCB, MT29F64G08CFACB MT29F128G08CKCCB, MT29F256G08CUCCB." .

[43] N. Mielke et al., "Flash EEPROM threshold instabilities due to charge trapping during program/erase cycling," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 3, pp. 335–344, Sep. 2004.

[44] E. H. Moore, "On the reciprocal of the general algebraic matrix," *Bulletin of the American Mathematical Society*, vol. 26, pp. 394–395, 1920.

[45] M. Moshayedi and P. Wilkison, "Enterprise SSDs," *ACM Queue*, vol. 6, no. 4, pp. 32–39, Jul. 2008.

[46] Nangate Development Team, "Nangate 45nm Open Cell Library." 2012.

[47] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-loading: Practical Power Management for Enterprise Storage," *Trans. Storage*, vol. 4, no. 3, pp. 10:1–10:23, Nov. 2008.

[48] NVM Express Workgroup, "NVM Express Specification 1.1b." .

[49] M. K. Qureshi, "Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2011, pp. 318–328.

[50] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, New York, NY, USA, 2009, pp. 14–23.

[51] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2009, pp. 24–33.

[52] I. Reed, "A Class of Multiple-Error-Correcting Codes and the Decoding Scheme," *IRE Professional Group on Information Theory*, vol. 4, no. 4, pp. 38–49, Sep. 1954.

[53] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *ACM SIGARCH Computer Architecture News*, New York, NY, USA, 2010, pp. 141–152.

[54] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, “Use ECP, Not ECC, for Hard Failures in Resistive Memories,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

[55] N. H. Seong, D. H. Woo, and H.-H. S. Lee, “Security Refresh: Prevent Malicious Wear-Out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping,” in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 383–394.

[56] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, “SAFER: Stuck-At-Fault Error Recovery for Memories,” in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA, 2010, pp. 115–124.

[57] S. M. Strande et al., “Gordon: Design, Performance, and Experiences Deploying and Supporting a Data Intensive Supercomputer,” in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*, New York, NY, USA, 2012, pp. 3:1–3:8.

[58] D. Vučinić et al., “DC express: shortest latency protocol for reading phase change memory over PCI express,” in *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014, pp. 309–315.

[59] D. T. Wang, “Modern Dram Memory Systems: Performance Analysis and Scheduling Algorithm,” University of Maryland at College Park, College Park, MD, USA, 2005.

[60] H.-S. P. Wong et al., “Phase Change Memory,” *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.

[61] T.-C. Yang, “Method for performing data shaping, and associated memory device and controller thereof,” US20140189222 A1, 03-Jul-2014.

[62] M. Yokotsuka, “Memory motivates cell-phone growth,” *Wireless Systems Design*, vol. 9, no. 3, pp. 27–30, 2004.

[63] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, “FREE-p: Protecting Non-Volatile Memory Against Both Hard and Soft Errors,” in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2011.

[64] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2009, pp. 14–23.

[65] “International Technology Roadmap for Semiconductors, 2011 Edition, Process Integration, Devices, and Structures,” 2011.

[66] “HowStuffWorks ‘How Flash Memory Works,’” *HowStuffWorks*. [Online]. Available: http://computer.howstuffworks.com/flash-memory.htm. [Accessed: 23-Jun-2014].

[67] "HowStuffWorks 'Inside a Digital Cell Phone,'" *HowStuffWorks*. [Online]. Available: http://electronics.howstuffworks.com/cell-phone.htm. [Accessed: 23-Jun-2014].

[68] "SSD | Understanding over-provisioning." [Online]. Available: http://www.kingston.com/us/ssd/overprovisioning. [Accessed: 03-Jul-2014].

Biography

Adam Jacobvitz was born on May 20, 1987 in Petaluma, CA, and lived in Sonoma County until graduating high school. In 2009, he received a B.S. in Electrical Engineering from California Polytechnic University, San Luis Obispo. He started working with Professor Daniel J. Sorin at Duke University the same year. During his graduate studies, Adam was primary author on three publications, a co-author on one publication, and a co-author on a patent related to work presented in this dissertation. He was also the receipient of an IBM Ph.D Fellowship in 2012. During his graduate studies, his research primarily focused on improving the endurance of non-volatile memories.

Publications

[1] John Ingalls, Adam Jacobvitz, Patrick Eibl, Michael Ansel and Daniel Sorin. "Experiences in Developing and Evaluating a Low-Cost Soft-Error-Tolerant Multicore Processor." *10th Workshop on Silicon Errors in Logic - System Effects (SELSE)*, April 2014.

[2] Adam N. Jacobvitz, Robert Calderbank, and Daniel J. Sorin. "Coset Coding to Extend the Lifetime of Non-Volatile Memory." *Non-Volatile Memories Workshop (NVMW)*, March 2014

[3] Adam N. Jacobvitz, A. Robert Calderbank, and Daniel J. Sorin. "Coset Coding to Improve the Lifetime of Memory." *19th International Symposium on High Performance Computer Architecture (HPCA)*, February, 2013.

[4] Adam N. Jacobvitz, A. Robert Calderbank, and Daniel J. Sorin. "Writing Cosets of a Convolutional Code to Increase the Lifetime of Flash Memory." *50th Annual Allerton Conference on Communication, Control, and Computing*, October, 2012. (Invited Paper)