A DATA AWARE APPROACH TO SALVAGE THE ENDURANCE OF PHASE-CHANGE MEMORY

by

Rakan Maddah

B.Sc. in Computer Science, Lebanese American University, 2007

M.Sc. in Computer Science, Lebanese American University, 2009

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Rakan Maddah

It was defended on

May 4^{th}, 2015

and approved by

Dr. Rami Melhem, Department of Computer Science

Dr. Sangyeun Cho, Samsung Electronics Co.

Dr. Yiran Chen, Department of Electrical and Computer Engineering

Dr. Youtao Zhang, Department of Computer Science

Dr. Taieb Znati, Department of Computer Science

Dissertation Director: Dr. Rami Melhem, Department of Computer Science

A DATA AWARE APPROACH TO SALVAGE THE ENDURANCE OF PHASE-CHANGE MEMORY

Rakan Maddah, PhD

University of Pittsburgh, 2015

Phase Change Memory (PCM) is an emerging non-volatile memory technology that could either replace or augment DRAM and NAND flash that are hindered by scalability challenges. PCM suffers from a limited endurance problem that needs to be alleviated before it can be endorsed into the memory stack. This thesis is based on the observation that the endurance problem and its ramification depend on the write data. Accordingly, a data-aware approach is applied to salvage the endurance of PCM at three different stages: pre-write fault avoidance, post-write fault tolerance and post-failure recovery.

The pre-write fault avoidance stage aims at reducing the endurance cost of servicing write requests. To this end, Cost Aware Flip Optimization (CAFO) is presented as an efficient technique to lessen the endurance degradation. Essentially, CAFO relies on a cost model that captures the endurance cost of programming memory cells based on their already stored values. Subsequently, the write data is encoded into a form that incurs a lower endurance cost than the original write data. Overall, CAFO is capable of reducing the endurance cost by up to 65% more than the existing schemes.

Worn out PCM cells exhibit a stuck-at fault model which makes the manifestation of errors dependent on the values that cells are stuck at. When a write fails, the data is rewritten inverted. This dissertation shows that applying data inversion at the post-write fault tolerance stage exploits the data dependent nature of errors which enables ECCs to tolerate faults up to double their nominal capability. Furthermore, extensions to RDIS which is an ECC designed specifically for the stuck-at fault model are presented.

At the post-failure recovery stage, Data Dependent Sparing is presented to manage bad blocks in PCM. Departing from the observation that defective blocks in the context of the stuck-at fault model still exhibit a low write failure probability due to the data dependent nature of errors, this thesis takes the approach of reusing blocks that are defective yet better-than-bad through a dynamic management of the reserve spare space. Data Dependent Sparing is capable of increasing the lifetime of PCM by up to 18%.

Keywords PCM, Endurance, Bit Flip Reduction, ECC, Bad Block Management.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF EQUATIONS

PREFACE

I am thankful to my adviser professor Rami Melhem for all the support and encouragement he provided during my Ph.D. journey. During my 5 years in the Ph.D. program, I do not recall a single instance where Rami shut his door against a discussion. Rami was always available to listen and give advice. I owe Rami for most of what I have learned during my Ph.D. years and he is and will always be a role model to me.

I would like to thank my thesis committee members professors Taieb Znati, Sangyeun Cho, Youtao Zhang and Yiran Chen for their support and constructive feedback during the materialization of my thesis. I am particularly grateful to professor Cho who co-advised me during the first 3 years of my Ph.D. and helped me set the path for my dissertation work.

I am grateful to all my friends that were always supportive throughout my graduate years. Particularly, I thank Deema Abdallah, Wissam Beaino, John Feghali, Iyad Batal, Jamal Alrifai, Shadi Halabi and Ohan Oumoudian.

I cannot but to express my gratitude to my wife's family for their support and their warm hospitality during my regular trips to the lovely city of Toronto.

I am eternally thankful to my brother Bacel and my sisters Wijdan and Dina for their unconditional love and support. You guys mean the world to me.

I am short of words that can truly describe my gratitude to my wife Zeina. This thesis would not have been achieved without your love, care, encouragement and patience. Thank you for being there for me in the good and bad times and for believing in us.

At last, I dedicate this thesis to my father Samih and my mother Souad. I would never have been the person I am without your guidance and support. Thank you for all what you have done to me. All I have and will accomplish are possible because of your sacrifices and love.

1.0 INTRODUCTION

Computer systems have integrated all aspects of human endeavors and activities. The enormous computational power gave researchers and scientists the ability to tackle and harness complex problems and mine enormous amount of data. This ability has led to unprecedented improvements in various fields, whether biological, economic or social. At the same time, these improvements shed light on new challenges and dimensions that require even more computational power than what is available. It is projected that every human being will produce more than 5000 gigabytes of data yearly which amounts to over 40 trillion gigabytes of memory [26, 2]. Clearly, the universe is turning digital. Therefore, computer scientists and engineers are faced with the challenge of offering exascale computing paradigms while sustaining the dependability and reliability of computer systems.

Among many elements that have contributed to the evolution of computer systems, the memory system is a defining component. DRAM as a main memory technology has scaled from memory chips that can hold few megabytes of data to memory chips that can hold gigabytes of data with improved performance. On the other hand, NAND flash solid state drives (SSD) replaced traditional hard disk drives while providing significantly faster access times. In the middle of this era of data explosion [26], it is of great importance to continue improving the performance and the density of main and secondary memories to meet the requirements of exascale computing power. Unfortunately, both DRAM and NAND flash are facing physical limitations putting their further scalability into jeopardy [30, 10, 12, 38]. Further scaling of DRAM's feature size makes the manufacturing process complex and costly with a decreased cell reliability and increased power consumption– 30% of total system power is consumed by DRAM [42, 14, 1]. Similarly, scaling NAND flash to smaller technology nodes remarkably decreases the endurance of the storage cells and increases the number of transient and hard errors.

1.1 ALTERNATIVE MEMORY TECHNOLOGY

To circumvent the challenges faced by DRAM and NAND flash and sustain the evolution of the memory system, researchers and engineers are turning their attention to alternative memory technologies [44, 42, 37] that can either replace or augment existing technologies.

Amongst several memory candidates, phase-change memory (PCM) is emerging as one of the most promising technologies due to its desirable characteristics in terms of superior scalability, low access latency and negligible standby power [44, 43]. Initial Assessments and measurements show that PCM can compete with DRAM and NAND flash in terms of performance while providing improved scalability and density [44, 88]. PCM encodes bits in different physical states of chalcogenide alloy that consists of Ge, Sb and Te. Data is stored in PCM devices in the form of either a low resistance crystalline state (SET) or a high resistance amorphous state (RESET). Switching between the states happens through the application of different programming currents that cause the chalcogenide material to melt and then re-solidify the material into one of the SET/RESET states.

1.2 CHALLENGES

Figure 1 compares PCM against both DRAM and NAND flash [69]. While PCM could be perceived as a substantial improvement over NAND flash, it falls short to DRAM. Specifically, PCM suffers from slow writes and limited endurance. To make PCM a viable memory technology that can be endorsed within the memory stack, tackling the aforementioned shortcomings is key and essential. While addressing the slow write problem [21, 63, 86, 33] is important, salvaging the limited endurance is a must to prevent system failures due to worn out memory devices. In fact, each PCM cell can endure a limited number of SET/RESET cycles. The heating and cooling process to program a cell leads to frequent expansion and contraction of the chalcogenide material. Consequently, the heating element detaches from the chalcogenide material after sustaining 10^6 to 10^8 writes on average [22, 23]. Such a phenomenon results in a stuck-at hard fault that can be subsequently read but not reprogrammed [46, 74]. Accordingly, this thesis investigates techniques

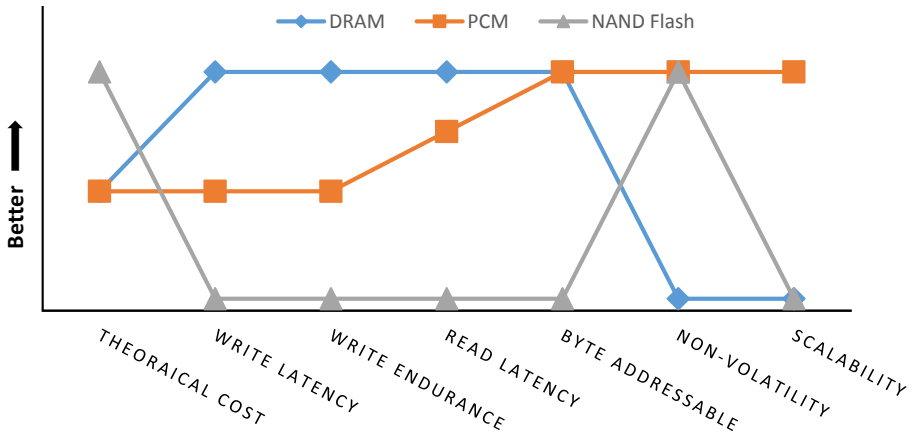

Figure 1: A comparison among DRAM, PCM and NAND flash [69]

to preserve the endurance and mitigate the wear related failures in PCM.

1.3 DATA AWARE APPROACH

A careful analysis of the characteristics of PCM would reveal that the data of write requests have a major impact on the endurance and its related problems. For instance, a submitted write request to a block with a data that has low resemblance to the already stored data would aggravate the endurance degradation rate. In fact, such a request results in a large number of cells that require programming as their intended values are different than the already stored values. Also, worn out cells are permanently stuck at a specific bit value. Thus, if a worn out cell gets written with a bit value identical to the bit value it is stuck at then no error would be manifested. Consequently, a submitted write request to a block classifies its worn out cells into two categories: (1) Stuck-at Wrong (SA-W) cells which are cells stuck at a bit value different than the intended value to be written and (2) Stuck-at Right (SA-R) which are cells stuck at a bit value identical the intended value to be written. Accordingly, a write request that would induce a large number of SA-W cells can lead to an unrecoverable write failure. Clearly, PCM's endurance problem and its ramifications

depend on the data pattern to be written. Therefore, this thesis approaches the endurance problem by exploiting this dependency through proposing data-aware solutions.

1.4 ELEMENTS OF A GOOD SOLUTION

This thesis envisions that tackling the limited endurance problem of PCM can be accomplished at three different stages:(1) **the pre-write fault avoidance stage**, (2) **the post-write fault tolerance stage** and (3) **the post-failure recovery stage**. At the pre-write fault avoidance stage, a good solution would schedule the write on a block that has not been written extensively and try to service the write with the minimal possible degradation in endurance. While such a solution lessens the endurance degradation rate of PCM, cell wear-outs are unavoidable. A worn-out cell cannot be programmed reliably to the intended bit value. Accordingly, such a cell will often manifest errors in the post-write stage. Therefore, combating cell failures through proactive error correcting schemes is essential for the successful deployment of PCM within the memory system. Undoubtedly, error correcting schemes are efficient and reliable to recover from errors. On the flip side, the capability of error correcting schemes has to be limited for complexity concerns. Consequently, a memory block turns defective when the number of accumulated faults gets above the capability of the deployed error correcting scheme. Hence, managing defective block at the post-failure recovery stage is a necessity.

1.5 THESIS CONTRIBUTION AND PREMISE

This thesis presents techniques to salvage the endurance of PCM at each of the aforementioned stages as depicted in Figure 2. All the proposed techniques exploit the dependency of the endurance on the write data. Overall, the key contributions of this thesis are the following:

1. *CAFO*, a pre-write fault avoidance technique that services write requests while minimizing the wear out rate through taking the endurance cost of bit flips into consideration.

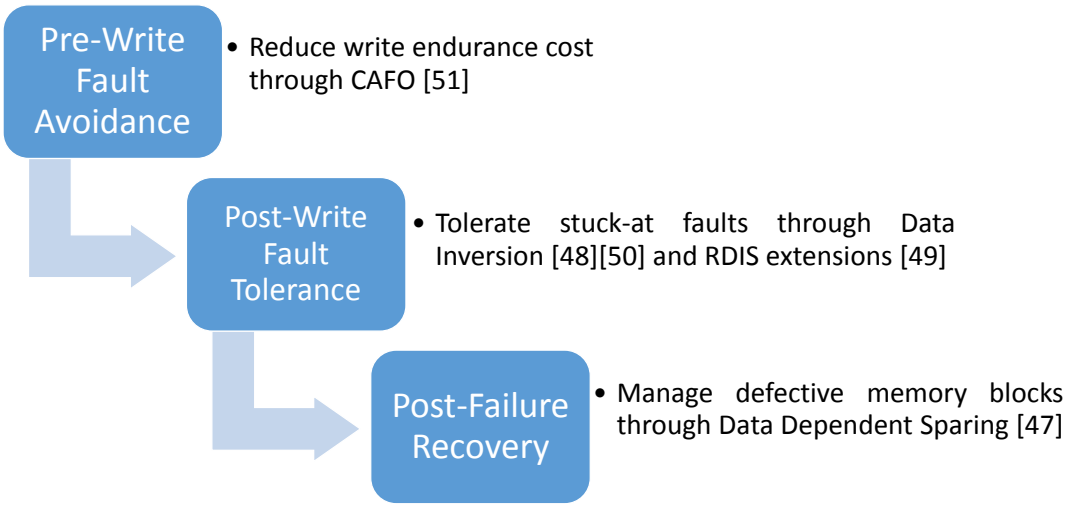

Figure 2: Thesis Contributions

2. *Data Inversion*, a post-write fault tolerance approach that increases the number of faults that error correcting codes can cover through exploiting the data dependent nature of errors. In addition, extensions to the PCM specific error correcting scheme *RDIS* are proposed to improve its error correcting capability.
3. *Data Dependent Sparing*, a post-failure recovery technique that delays the retirement of defective blocks by being aware of the data dependent nature of write failures.

The aforementioned techniques are effective in salvaging the endurance of PCM. *CAFO* is capable of reducing the endurance write cost by up to 65%. *Data Inversion* enables error correcting codes to tolerate a number of faults that is up to double their nominal capability which leads to up to 34.5% increase in lifetime. Similarly, the *extensions to RDIS* allow the correction scheme to effectively recover from detrimental fault patterns and reduce its space overhead. At a small management cost and with less than 1% spare over-provisioning, *Data Dependent Sparing* achieves the same lifetime as the conventional approach of managing defective blocks that assumes 20% spare over-provisioning.

Salvaging the endurance problem of PCM improves its reliability which paves the way for its integration within the memory stack. Accordingly, computer systems can benefit from a memory

technology that is dense and scalable.

1.6 ORGANIZATION

The remainder of this thesis is organized as follows. Chapter 2 presents a background on PCM and lists the related work. Chapter 3 presents and evaluates CAFO. Chapter 4 presents and evaluates Data Inversion and extensions to RDIS. Chapter 5 presents and evaluates Data Dependent Sparing. Chapter 6 presents the application of the proposed schemes in the context of multilevel cells. Finally, Chapter 7 summarizes the thesis and proposes future directions.

2.0 BACKGROUND AND RELATED WORK

2.1 PCM TECHNOLOGY

The concept of Phase Change Memory was envisioned by Dr. Stanford Ovshinsky back in 1960s [58]. PCM is a non-volatile memory that is amenable to process scaling [44]. The storage element of PCM, depicted in Figure 3(a), is composed of chalcogenide material sandwiched in between two electrodes and a heater (resistance). The chalcogenide material is characterized by its ability to switch into stable phases through the application of a heating current.

2.1.1 Write Operation

A PCM cell can be in two states as illustrated in Figure 3(b). To put a cell into the RESET state, the heating element injects a high short current pulse. A short current increases resistivity by abruptly disconnecting the current quenching the heat generation and freezing the chalcogenide into the amorphous state [44, 43]. On the other hand, the SET state is reached by applying a long moderate current pulse. A long current ramps down gradually inducing crystal growth [44, 43].

2.1.2 Read Operation

To read a PCM cell, a moderate read voltage is applied to sense the resistance of the cell. The sensed resistance is compared against a reference threshold value to differentiate between the SET and RESET states.

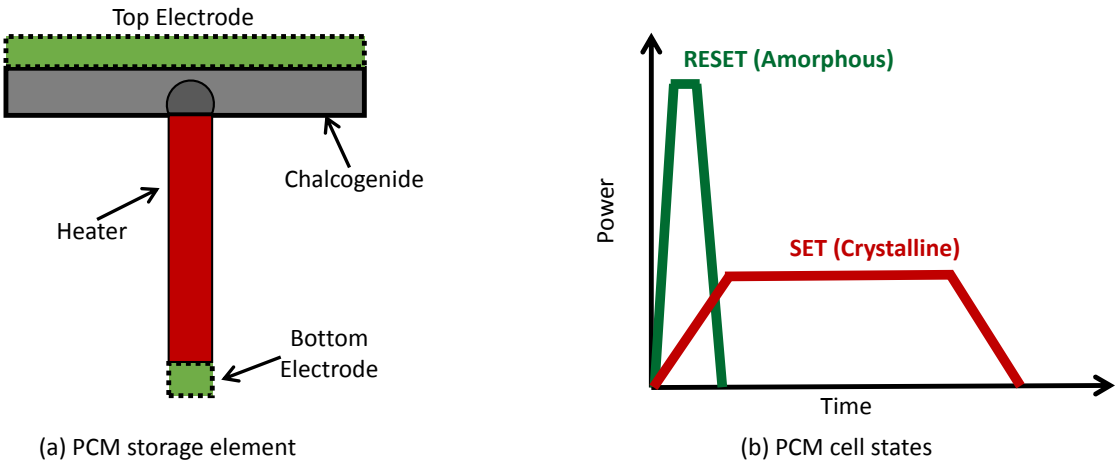

Figure 3: An example of PCM storage elements (a) and the states it can take (b).

2.1.3 Endurance

The wear mechanism in PCM is associated with the write operation. The heating and cooling process to program a cell leads to frequent expansions and contractions of the material. Consequently, programming current cannot be injected reliably anymore to switch the state of a cell [9].

The endurance of a cell is determined by the number of writes that can be performed before that cell wears out. On average, PCM cells can endure 10^6 to 10^8 writes [22, 23]. When a cell reaches its endurance limit, the heating element detaches from the chalcogenide material resulting in a cell that is stuck at a specific state. While stuck-at cells cannot be reliably programmed, their stuck-at value can still be read [46, 74]. Generally, PCM cells can be classified intro three categories as illustrated in Figure 4: (a) good cell, (b) stuck-at SET cell, and (c) stuck-at RESET cell. Furthermore, stuck-at cells can be perceived as either stuck-at wrong (SA-W) or stuck-at right (SA-R) after a write operation. A SA-W is a stuck-at cell that exhibits an error as it is expected to store a value different than its stuck-at value. Conversely, a SA-R is an error free cell as it is stuck at a value that matches the expected value to be stored.

Both the SET and the RESET operations deteriorate the endurance of PCM cells. Since the programming current of the RESET state is significantly more intense than the SET state, studies

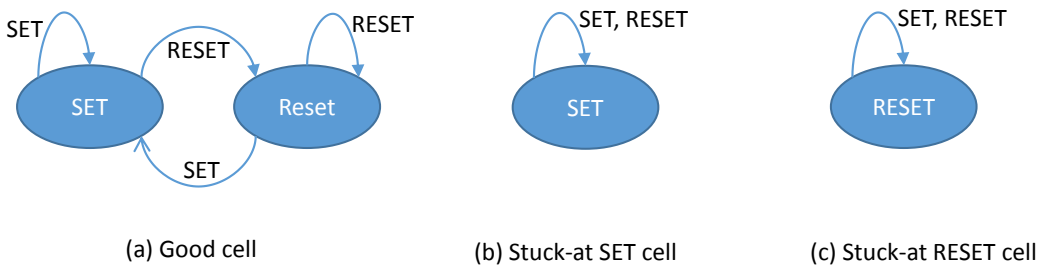

Figure 4: Difference between a good and stuck cell.

have shown that the endurance is mainly dependent on the RESET state [87, 39]. In fact, the RESET state could degrade the endurance by up to four times more than the SET state. This phenomenon creates a write endurance asymmetry in between the SET and the RESET states.

2.1.4 Stored Levels

PCM cells can be used to store two or more states. A two states memory chip, also known as single level cell (SLC), stores binary data. Conventionally, a cell in the crystalline state is interpreted as a bit value of "1" while a cell in the amorphous state is interpreted as a bit value of "0". Nevertheless, the resistances in the fully crystalline and amorphous states differ by two to three orders of magnitude. This difference in resistance can be exploited to store multi-bits per cells [60, 4]. Instead of representing the data in a cell using two states, the resistance range can be broken down into multiple states where each state represents a particular data pattern i.e. multi-level cell (MLC). Figure 5 shows a PCM cell arranged in two states (a) and 4 states (b) capable of storing 1 bit and 2 bits per cell respectively.

Obviously, the major advantage of MLC over SLC is density. However, a denser MLC PCM chip suffers from drift errors. As a matter of fact, PCM cells are susceptible to resistance drift over time. This drift in resistance is not large enough to cause SLC cells to switch levels, but it affects MLC cells leading to a high drift error rate that cannot be practically tolerated by error correcting codes [84]. Also, programming of MLC cells could require several iterations to make

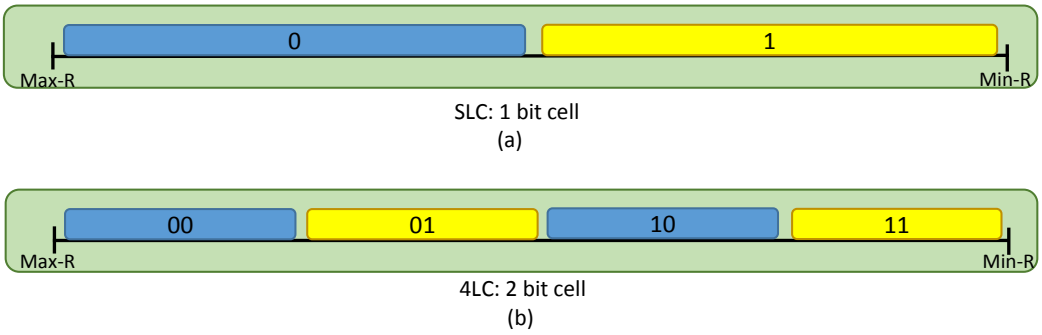

Figure 5: The concept of multilevel PCM cell.

the cells attain the desired resistance level which could exacerbates the endurance degradation rate compared to SLC cells [4, 57]. Yet, the drift errors are the major challenge for MLC cells. One way of mitigating the drift errors is to store fewer states per cell. In [84, 34, 77], it was shown that storing 3 levels per cell instead of 4 would eliminate most of the drift errors.

2.2 RELATED WORK

2.2.1 Bit Flips Reduction Techniques

To tackle the endurance problem of PCM at the pre-write fault avoidance stage, several techniques have been proposed. The key concept of those techniques is to service write requests while reducing the number of cells that requires flipping as such a practice would decelerate the wear out rate. In this realm, Differential Writes [43] is a standard technique deployed in PCM. When a write to a block is submitted, differential write compares the data vector to be written to the already stored data vector within the block. Subsequently, only those cells with non-matching values are to be programmed.

Moreover, several techniques have been proposed to complement Differential Writes. The target of those technique is to service write requests while programming as few cells within a

block as possible. For instance, Flip-N-Write [15] encodes the data to be written into two different vectors, one in the regular form and the other in the inverted form. The vector that would yield in the least number of bit flips when paired with Differential Writes is chosen to be written. Flip-N-Write requires the introduction of one auxiliary bit to flag the inversion step, which makes the space overhead of Flip-N-Write depends on the granularity of the block it is applied to e.g. $8:1$, $16:1$, $32:1$ etc.

In addition to Flip-N-Write, Flip-Min [31] has been proposed. Flip-Min is based on the concept of coset coding. An inverted (72, 64) Reed Muller code is used to encode each possible input data vector into 256 unique different vectors. Among those vectors, the vector that would yield in the least number of bit flips when paired with differential write is selected to be written. Flip-Min has been showed to achieve fewer bit flips than Flip-N-Write, while incurring a space overhead of 12.5%.

2.2.2 Error Correction Schemes

For post-write fault tolerance, coding theory is an old and well established discipline in which several codes to tolerate errors have been proposed. For instance, SEC-DED [28] is a well known error correcting code deployed in DRAM modules to recover from bit errors. SEC-DED can cover one error and detect a second one. Given its low capability, SEC-DED does not provide enough coverage for memories expected to exhibit multiple simultaneous errors such as PCM. Accordingly, multi-bit error correcting codes such as *BCH* [7] are considered. Interestingly, the error correcting capability of multi-bit error correcting codes is tunable. Actually, setting the capability of the code depends on two factors. The first is the space overhead that can be incurred in terms of the number of added auxiliary bits as increasing the error correction capability requires increasing the number of auxiliary bits. The second is the complexity of the code as increasing its capability increases its complexity [82, 79]. Hence, there exists a trade-off between the capability of error correcting codes and their space and complexity overhead.

In addition to coding theory, several error correction schemes that were designed specifically for PCM's fault model have been proposed. Among those, *RDIS* [53] was proven to have a capability of tolerating a significantly large number of faults. RDIS' encoding process consists of

identifying a set containing all the SA-W cells. This set is dubbed the *invertible set* as it is enough to read its elements inverted to retrieve the intended data to be written. To identify this set, RDIS logically represents a memory block as a two-dimensional array and introduces a counter for each row and column. The value of each counter is incremented if the row or column it represents exhibits a SA-W cell. Subsequently, all the cells whose corresponding row and column counters share the same value are extracted as a sub-array. The latter is a candidate invertible set. Nevertheless, the sub-array could contain SA-R cells as they happen to be on the intersection of a row and a column both containing at least one SA-W cell. To take out the SA-R cells, the initial sub-array cells are inverted and the corresponding counters are incremented based on the newly revealed SA-W cells. Hence, a new sub-array that is smaller than the initial one is formed. This process is repeated recursively until the size of the initial sub-array is reduced to zero. Figure. 6.

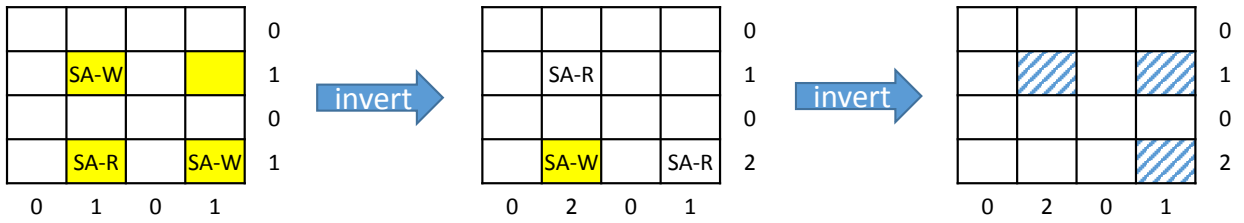

Figure 6: RDIS encoding process. Highlighted cell(s) represent(s) the extracted sub-array at each step. Patterned cells represent the identified invertible set.

shows an example of RDIS's encoding process on a block that suffer from three stuck-at cells. After the completion of the first write operation, two of the stuck-at faults end up being SA-W and one cell ends up being SA-R leading to an initial sub-array of 4 cells (highlighted in yellow). To recover from the two SA-W cells, RDIS inverts the data to written within the initial sub-array. Consequently, one cell ends up being SA-W as the inversion step has the effect of exchanging the role of SA-W and SA-R cells. Subsequently, the corresponding flags and of the new SA-W cell are incremented and a second sub-array is formed. At last, RDIS inverts the data within the second sub-array rendering the remaining SA-W cell to be become SA-R. Thus, reducing the size of the initial sub-array to zero and forming an invertible set of 3 cells (blue pattern).

When it comes to the correction capability of RDIS, the recovery from three stuck-at faults is

guaranteed. Nevertheless, RDIS can recover from a higher number of faults with a significantly high probability. In fact, a specific pattern of faults has to form for RDIS to halt. For instance, if the rows and columns of an extracted sub-array each contains at least two stuck-at cells such that their location is adjacent to the location of the faults in another row or column then such a pattern causes RDIS to halt and is known as a loop of faults. In fact, such a pattern prevents an extracted sub-array from getting reduced to 0 when the faults happen to be alternatively stuck. Figure 7 shows an example of a loop of four faults. The size of the initial sub-array cannot be reduced as the example shows. Each inversion step will simply exchange the role of the faulty cells between SA-R and SA-W without being able to eliminate any cell.

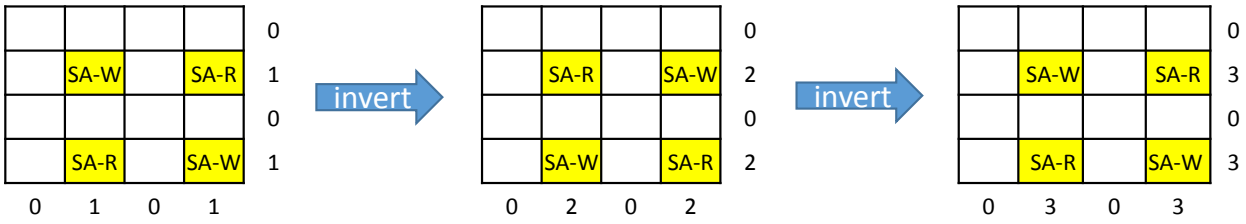

Figure 7: A loop of faults that causes RDIS to halt. Highlighted cells represent the extracted sub-array at each inversion step.

In addition to RDIS, several other error correcting schemes designed specifically for PCM have been proposed. First, Error Correcting Pointer (ECP) [73] provides a preassigned number of programmable "correction entries". A correction entry holds a pointer (address) to a faulty cell within the protected block and a "patch" cell that replaces the faulty one. When a faulty cell is detected, a correction entry is allocated to cover the cell. A memory block is de-commissioned when the number of faulty cells exceeds that of the correction entries.

SAFER (Stuck-at-Fault Error Recovery) [76] When the value of the faulty cell is different from the intended value to be written, all cells in the group are written inverted. If the data block is to be partitioned into n groups, then SAFER allows $\log_2 n$ "repartitions". Repartitioning is done whenever a new fault is detected. Therefore, SAFER guarantees the recovery from $\log_2 n + 1$ faults. Any additional fault is tolerated only if it occurs in a fault-free group. Otherwise, the block has to be retired.

PAYG (Pay-As-You-Go) [66] is a resilient architecture proposed to decrease the storage overhead of auxiliary bits information required by ECP and SAFER. Essentially, PAYG moves from a uniform allocation of auxiliary bits across the protected memory blocks to a dynamic on demand allocation. PAYG exploits the variability in lifetime that the memory blocks exhibit and assigns more auxiliary bits to memory blocks that suffer from a large number of faulty cells.

2.2.3 Bad Block Management

When the number of hard faults within a protected block gets above the capability of the deployed error correction scheme, the block turns defective and write failures can start to occur. While this scenario is rare in DRAM where transient errors are dominant, it is a common practice in NAND flash SSDs [55] as several blocks turn defective due to limited endurance. After the first write failure is detected on a block, it is labelled bad and it is subsequently replaced by a healthy spare block; a practice commonly known as *bad block management* [56, 85, 32]. Bad block management is expected to remain integral for PCM due to its limited endurance.

2.3 DIFFERENCES FROM PREVIOUS WORK

The work in this thesis has substantial differences to the previously proposed work. All the previous work at the pre-write fault avoidance stage focused on reducing the number of bit flips without exploiting the endurance degradation asymmetry in writing different bit values. CAFO introduces a model that assigns different endurance costs to each cell that requires programming; thus capturing the asymmetry in the write cost. In addition, CAFO introduces a new a 2D encoding scheme while previous work considered a 1D model. A 2D encoding allows for several modifications of the write data, which lead to a further reduction to the endurance cost of servicing write requests.

To increase the capability of an error correcting code, a number of auxiliary cells has to be added. In general, a block of size n requires $t \times lgn$ auxiliary cells to recover from t errors [7]. Increasing the capability of an error correcting code does not only increase its space overhead, but also increases its encoding/decoding complexity. Data Inversion is capable of allowing an error

correcting code of capability t to tolerate additional faults without the need of introducing a large number of auxiliary cells. In fact, Data Inversion introduces only one additional auxiliary cell. Tolerating the additional number of faults is accomplished through exploiting the data dependent nature of errors. Thus, Data Inversion enables tolerating an additional faults with a significantly decreased complexity and space overhead.

When it comes to bad block management, this thesis takes a completely different approach in managing defective blocks. Data Dependent Sparing builds on the observation that write failures on defective blocks are rare due to the data dependent nature of errors to delay the retirement of defective blocks and better utilizes the available spare blocks.

3.0 PRE-WRITE FAULT AVOIDANCE

For endurance constrained memories such as PCM, it is of great importance to decelerate the wear-out rate of storage cells. One way of achieving this goal is to service write requests while inducing as few bit flips as possible. In its simplest form, bit flip reduction can be achieved through a bit by bit comparison between the new data vector to be written and currently stored data vector. Subsequently, only those bit positions with non-matching bit values would be toggled. In this sequel, the similar the new data is to the old stored data, the fewer the number of bit flips. Accordingly, manipulating the new data vector to make it more similar to the old vector would yield in bit flips reduction. This manipulation consists of a well defined encoding of the new data vector to an intermediate vector that is similar to the old stored vector. It is to be noted that a well defined encoding is a requirement so that the data is properly decoded at retrieval time. While a number of bit flip reduction techniques have been proposed [15, 31], none of those techniques are aware of the write endurance asymmetry (refer to Section 2.1.3) manifested by the RESET state (binary "0") being more detrimental to endurance than the SET state (binary "1"). This chapter presents *CAFO:Cost-Aware Flip Optimization* [51]. CAFO is aware of the write endurance asymmetry. It introduces a cost model that captures the deterioration of endurance caused by each bit flip. Subsequently, CAFO encodes the data vector into a form that would require a lower overall endurance cost than the originally intended data vector to be written. Conceptually, CAFO makes the key contribution of moving from bit flip reduction to cost reduction.

3.1 CAFO DETAILS

Minimizing the number of bit flips seems to be a good solution to tackle the endurance problem of PCM. Nonetheless, minimizing the number of bit flips in a manner that is oblivious to the asymmetry of writing different bit values could backfire. For example, let us assume that writing a bit value of ”0” on a PCM cell has an endurance cost that is 4 times more than the endurance cost of writing a bit value of ”1” [87, 39]. Furthermore, let us assume that the data vector of a submitted write request on a PCM block differs than the previously programmed vector on the same block by 3 cells that would need to be flipped from ”0” to ”1”. If a bit flip minimization scheme encodes the data in such a way that brings the number of bit flips from 3 to 1, then the actual gain that is achieved depends on the bit value to be written by the single flip. While a single bit flip of ”0→1” would be a significant improvement, a single bit flip of ”1→0” would be harmful as writing a ”0” is 4 times more detrimental to endurance than writing a ”1”.

Clearly, focusing solely on reducing the number of bit flips of a write operation does not capture all the dimensions affecting the endurance. This said, there is a need to associate a cost to write operations and to present schemes that aim at reducing the cost of servicing write requests.

3.1.1 Cost Model

When a write request to a block is submitted, its data vector is compared to the previously written data vector at the same block. Subsequently, the cells in the block can be classified into two categories: matching cells and non-matching cells. Accordingly, the proposed model labels the cost of matching cells with ”c” and ”d” for matching bit values of ”0” and ”1” respectively. Similarly, the cost of non-matching cells are labeled with ”a” and ”b” for bit flips of ”0→1” and ”1→0” respectively. Figure 8 shows an example for an 8 bit block where the stored data is compared to the new data to determine the endurance cost of programming each cell within the block.

Setting the cost for ”a”, ”b”, ”c” and ”d” depends on the memory technology being modeled and the objective function being optimized. For example, only non-matching cells would require programming in PCM. Hence, both ”c” and ”d” would be assigned a cost value of ”0” as cells with such labels do not cause any endurance degradation. Moreover, the asymmetry of flipping

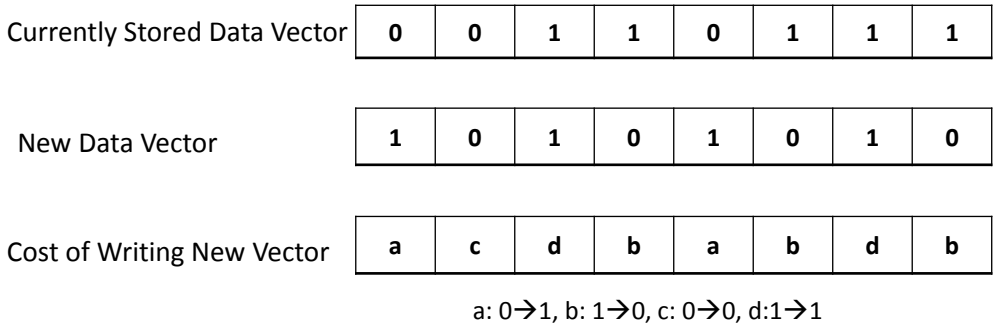

Figure 8: Cost model.

”0→1”and ”1→0” would be captured through assigning different cost values to ”a” and ”b” e.g. ”a =1 ”, ”b = 3” if the endurance cost of writing a ”0” is 3 times the endurance cost of writing a ”1”.

Adopting this cost model would give the ability to associate a cost to every write operation. As a matter of fact, the cost of a write operation, C, would be computed by the following formula:

$$C = n_{0\to1}a + n_{1\to0}b + n_{0\to0}c + n_{1\to1}d \tag{3.1}$$

where $n_{0\to1}$, $n_{1\to0}$, $n_{0\to0}$ and $n_{1\to1}$ are the total numbers of a, b, c and d flips respectively. Once the cost of a write operation has been computed, it is of great benefit to service write requests while minimizing the write cost, C. It is worth noting that the proposed cost model is general and can be applied to other objectives than calculating the endurance cost. For example, the cost model could be applied to capture the write energy cost. In addition, the model is not specific to PCM and can be applied to any memory technology that exhibits asymmetries such as STT-RAM. Nevertheless, this thesis focuses on salvaging the endurance of PCM. Hence, the next section presents CAFO's new encoding scheme that aims at servicing write requests while minimizing the overall endurance cost.

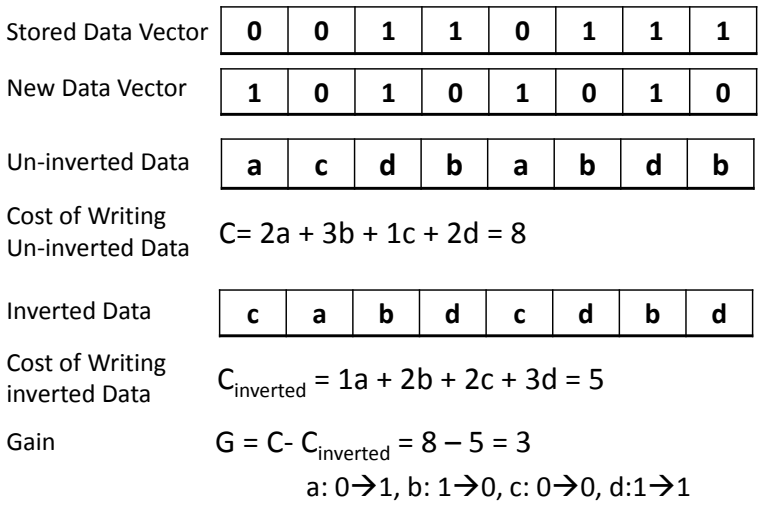

Figure 9: Gain calculation where the costs of a, b, c and d are 1, 2, 0 and 0 respectively.

3.1.2 Cost Minimization Encoding Scheme

CAFO models an N bits memory block as a logical arrangement of an $n \times m$ array. For each row and column, an additional auxiliary bit is added. When a write request is submitted, the new data is compared to the old written data in order to determine the cells that need to be updated as their bit values differ from previously programmed bit values. Subsequently, the cost of each bit flip is assigned in accordance with the cost model presented in the previous section.

Next, the gain of each row is determined as the difference between the costs of writing the data vector in two different forms. The first is the un-inverted form while the second is the inverted form. Specifically, the gain can be determined according to the following formula:

$$G = C - C_{inverted} \tag{3.2}$$

where G is the gain, C is the cost of writing the data in the un-inverted form and $C_{inverted}$ is the cost of writing the data in the inverted form. Figure 9 shows an example of calculating the gain where the cost of writing the data is calculated in accordance to Formula (3.1). At first, the cost of writing the data un-inverted is determined. Then, the data is logically inverted and its writing cost

is determined. Subsequently, the gain (3) is calculated through subtracting the cost of writing the data un-inverted (8) from the cost of writing the data inverted (5).

A positive gain implies that inverting the data for a row has a lower overall cost than the un-inverted data. Hence, every row with a positive gain is inverted and its corresponding auxiliary bit is toggled. After flipping the rows with positive gains, the same process is applied to the columns. Next, the gain of the rows is recalculated as flipping the columns may have turned the gain of certain rows to become positive. In practice, the process of flipping the rows and columns is repeated until the gain of every row and column is either zero or negative. Algorithm 3.1 shows the steps taken by CAFO's encoding scheme to reduce the cost of write operations.

Algorithm 3.1 CAFO Encoding

```
1:  Invert = True
2:  ColumnCheck = False
3:  while Invert do
4:      if PositiveGainRows() then
5:          Flip Rows with Positive Gain
6:      else
7:          if ColumnCheck then
8:              Invert = False
9:              break
10:         end if
11:     end if
12:     if PositiveGainColumns() then
13:         Flip Columns with Positive Gain
14:         ColumnCheck = True
15:     else
16:         Invert = False
17:     end if
18: end while
```

The *Invert* variable at line 1 serves as the loop invariant at line 3. In case the rows shows a negative gain, the *ColumnCheck* variable at line 2 serves as a flag to avoid calculating the gain of

the columns again if they have already been calculated. In fact, if the columns have already been flipped and the gain of the rows does not yield in a positive gain for any of the rows then the gain of the columns does not change and remains negative. The procedure at line 4, *PositiveGainRows()*, computes the gain of every row and return the list of rows that yields a positive gain in order to be flipped. If the list is empty and the gain of the columns have already been calculated, i.e. this is not the first iteration of the loop, then the algorithm terminates through setting the loop invariant to false at line 8 and discontinues the execution of the remaining of the statements within the body of the loop at line 9. In case any row got flipped or it is the first iteration of the loop, the *PositiveGainColumns()* procedure at line 12 returns a list of columns that have a positive gain. If the list in not empty, then the returned columns are flipped and the *ColumnCheck* flag is set at lines 13 and 14 respectively. If the list is empty, then the algorithm terminates through setting the loop invariant at line 16 to false. An empty list indicates no positive gain for any of the columns which is the same for the row as their gain has already been calculated and the rows with positive gain have already been flipped.

Applying a data flip to reduce the cost of write operations have been considered before e.g. Flip-N-Write. However, the flips are only considered at either the level of rows or columns without interleaving. The rationale behind the proposed encoding scheme is that allowing the interleaving of rows and columns data flips could allow more overall cost reduction as compared with inverting the rows or columns only. As a matter of fact, it can be shown that the interleaving could turn a row (column) that has had initially a negative gain into a positive gain. Hence, such a row (column) should be flipped resulting in an overall reduction in the write operation cost. It is to be noted that a flip is always guaranteed to reduce the overall cost as the decision to flip is based on a positive gain derived through Formula (3.2).

It is important to mention that CAFO's encoding algorithm is guaranteed to terminate. Stated differently, it is not possible for the alternation of the flips between the rows and the columns to continue infinitely. Consider any combination of one row and one column within the 2D block. They both intersect at one cell. If the row gets inverted then the intersecting cell could end up in the state of either a matching cell or a non-matching cell. Next, if the column gets flipped due to positive gain and the intersecting cell before the flip was in the non-matching state then the non-positive gain of the row increases (the row got flipped before so its gain is either negative or zero)

as the non-matching state of the intersecting cell turned matching i.e. the cell does not require programming which decreases the cost of writing the data un-inverted. Hence, the flip of the column will not lead to a second flip for the row. In the other case where the state of the intersecting cell before flipping the column was matching, the gain of the row could become positive as the state of the intersecting cell turns into non-matching which increases the cost of writing the data un-inverted. Now, if the gain of row becomes positive then it will be flipped again which will increase the non-positive gain of the already flipped column as the state of the intersecting cell becomes non-matching after the second flip of the row. Accordingly, given any combination of a row and a column after two flips for the row (column) and one flip of the column (row) it is guaranteed that one of them will incur a negative gain which prevents an infinite alternation between the flips of the rows and the columns and guarantees termination.

Another point to note is concerning the optimality of CAFO's encoding algorithm. In theory, CAFO's encoding could be interpreted as the Gale-Berlekamp Switching Game[13]. This game considers an m by n grid of light bulbs and assigns and light switch for every row and for every column. Two players, A and B, alternate in turning the switches in order to achieve two different objectives. Player A tries to maximize the number of light bulbs that are "on" while player B tries to maximize the number of light bulbs that are "off". The Gale-Berlekamp game was proven to be NP-hard [72]. Without loss of generality, CAFO's encoding setting can be reduced to the Gale-Berlekamp game which makes an optimal encoding for CAFO NP-hard. Hence, Algorithm 3.1 is not optimal and uses the heuristic of alternating the flips between the rows and the columns until a non-positive gain is reached for every row and column. In fact, Section 3.1.4 presents an optimization to further improve the cost reduction capability of the encoding algorithm.

3.1.3 CAFO Encoding Example

To make the encoding process clear, Figure 10 shows an example where a 64 bits block is represented as an 8×8 array. For ease of illustration, let us assume that the costs of the flips "0→1"and "1→0", i.e. "a" and "b", are equal while the costs of the flips "0→0" and "1→1", i.e. "c" and "d", are both zero. Accordingly, "1" represents a cell that needs to flipped while "0" otherwise. Since rows 5 and 8 both exhibit a positive gain as shown in Figure 10(a), those rows are flipped.

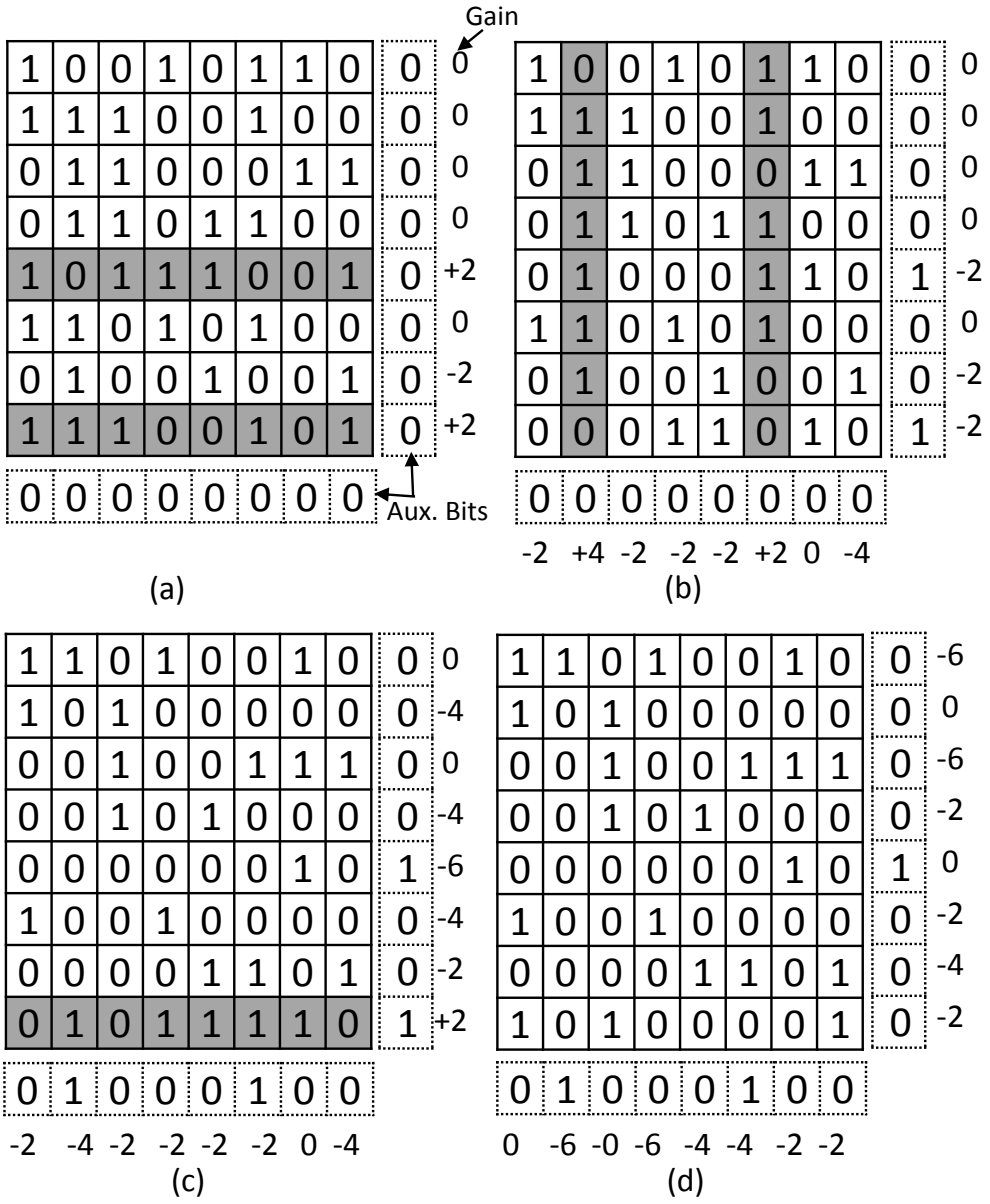

Figure 10: CAFO encoding example where the number of bit flips is reduced from 33 to 21. A ”0” represents a matching cell (c = d = 0), and a ”1” represents a non-matching cell (a = b = 1). ”Gain” is calculated as per Formula (3.2). Shaded cells represent rows and columns that are to be flipped.

Subsequently, the gain of columns is computed as shown in Figure 10(b). Since columns 4 and 6 both show a positive gain, they are both flipped and the gain of the rows is recalculated as illustrated in Figure 10(c). Next, row 8, incurring a positive gain once again, gets flipped as shown in Figure 10(d). Now that all rows and columns incur a non-positive gain, the encoding process terminates. Overall, Figure 10 shows that the proposed encoding scheme reduces the cost (number of bit flips) from 33 to only 21. It is to be noted that the example does not take into consideration the cost of toggling the auxiliary bits. Nevertheless, our scheme can easily account for the cost of the auxiliary bits as shown in Section 3.1.5.

To illustrate the difference in the encoding process between CAFO and a row only flip scheme, Figure 11 shows that applying Flip-N-Write on the same data as in Figure 10(a) would reduce the number of bit flips from 33 to 25. Hence, CAFO reduces the cost of the same write operation by almost 20% more than Flip-N-Write. It is to be noted that a row is formed of 4 bits so that both schemes have the same level of space overhead (16 auxiliary bits).

In general, flipping a row (column) could increase the gain of the columns (rows). In turn, when those columns (rows) are flipped the overall cost would be reduced. For example, row 5 in Fig. 10(b) got flipped which increased the gain of column 2 as the intersection bit in between the row and the column flipped from bit value ”0” to ”1”. When column 2 got flipped, the intersection bit with row 5 got reverted back to bit value ”0”. Thus, interleaving the flips of rows and columns allowed for the reduction of one additional bit for both the row and the column as opposed to flipping only the row or the column.

3.1.4 Encoding Scheme Optimization

Algorithm 3.1 assumes that the overall cost of a write operation cannot be further reduced once the gain of every row and column is non-positive. This section investigates the possibility of further reducing the cost of a write operation even when no row or column shows a positive gain. Consider the example in Figure 12 where a data cell with “1” represents a cell whose bit value differs from the current written value and “0” otherwise i.e. the costs of a, b, c and d flips are 1, 1, 0 and 0 respectively. Although all the rows and columns incur a non-positive gain, the example portrays that flipping row 2 and column 1 would reduce the cost of the write operation from 5 bit flips to 3.

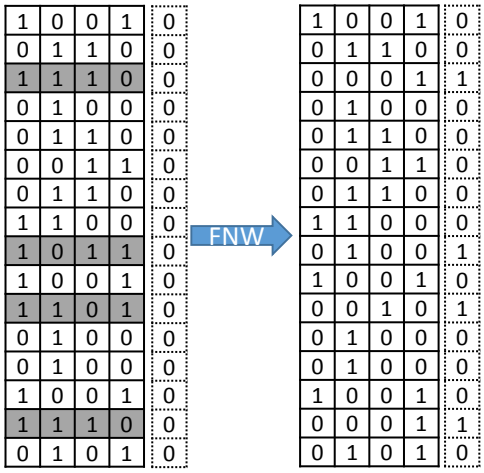

Figure 11: An example of Flip-N-Write (FNW) encoding where the number of bit flips is reduced from 33 to 25. A data cell with "0" represents a match with the already written bit, "1" otherwise. Shaded cells represent a row that is to be flipped.

Motivated by the example in Figure 12, the conditions that have to be met in order for the cost to be reduced when flipping a row and a column with non-positive gains are next formulated.

Let us start by defining $g_{r,c}$ as the gain of flipping the cell intersecting at row r and column c, which is determined by subtracting the cost of writing the intersecting cell un-inverted from the cost of writing the intersecting cell inverted. Furthermore, let G_r and G_c be the gains of row r and column c such that $G_r \leq 0$ and $G_c \leq 0$. When row r and column c are both flipped together, the cell at their intersection does not get flipped. Accordingly, the gain of flipping the intersecting cell, $g_{r,c}$, has to be deducted from the overall gain of flipping both the row and the column. Hence, the total gain, G_{r+c}, of flipping row r and column c would be:

$$G_{r+c} = G_r + G_c - 2g_{r,c} \tag{3.3}$$

where $g_{r,c}$ is multiplied by 2 as the local gain of the intersecting cell has to be deducted from the gains of both row r and column c.

Based on Formula (3.3), it is possible to incur gain for $G_r \leq 0$ and $G_c \leq 0$ if $g_{r,c} < 0$. In fact, $g_{r,c}$ is less than zero when its bit value to be programmed matches the previously written bit value, i.e. a matching cell, as only such a cell would incur a negative gain in a flip operation. Accordingly,

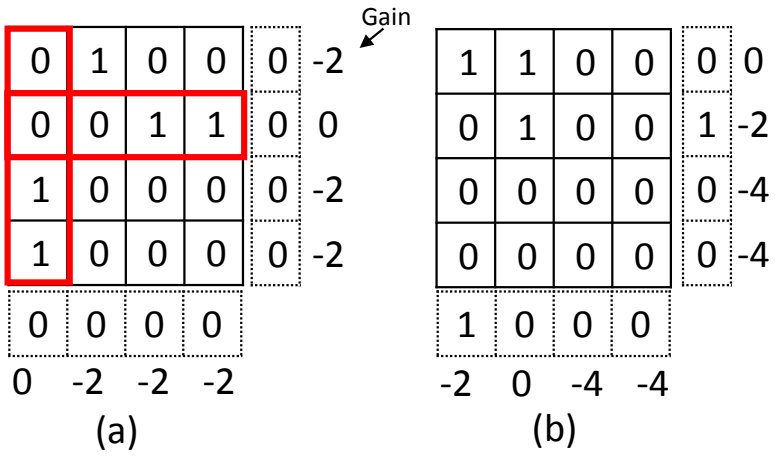

Figure 12: An example that shows that the cost of a write can still be reduced even when no row or column shows a positive gain. A data cell with "0" represents a match with the already written bit, "1" otherwise. Flipped rows and column are highlighted. The number of bit flips is reduced from 5 to 3.

flipping a row and a column incurring a non-positive gain would yield in cost reduction if the following condition is met:

$$G_{r+c} = G_r + G_c - 2g_{r,c} > 0. \tag{3.4}$$

Going back to the example in Figure 12, $g_{2,1}$ is -1 as the cost of "a" and "b" flips is one. Thus, Formula (3.4) is satisfied as $0 + 0 - 2(-1) > 0$.

The suggested optimization is not limited to one column and one row. As a matter of fact, the optimization works for one column with multiple rows as long as the overall cost is reduced. Generally, flipping one column, c, intersecting with a subset of rows, S, yields in overall cost reduction if the following formula is satisfied:

$$\sum_{r \in S} G_r + G_c - 2 \sum_{r \in S} g_{r,c} > 0. \tag{3.5}$$

Similarly, one row could be selected with multiple columns and Formula 3.5 can be equally applied.

Figure 13 illustrates an optimization example where one column is flipped with two rows. Initially, all columns and rows show a non-positive gain (Figure 13(a)). Nevertheless, column 2 intersects with rows 2 and 3 at two matching cells. Accordingly, the selected column and rows

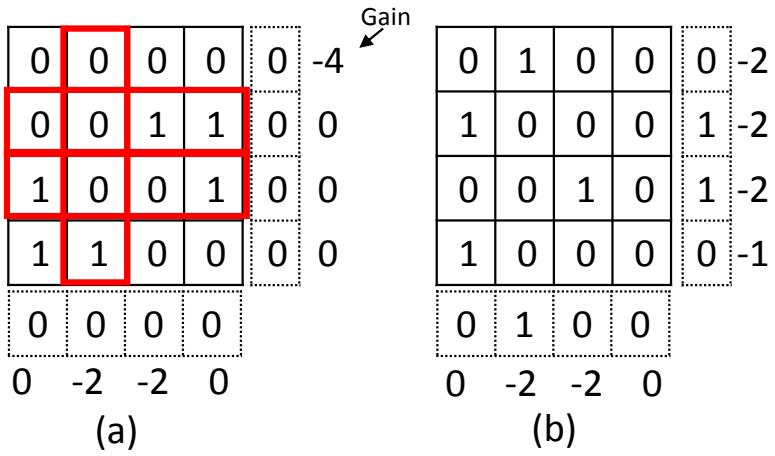

Figure 13: An example of CAFO's optimization. A data cell with "0" represents a match with the already written bit, "1" otherwise. The cost of flipping an intersecting matching cell $c_{r,c}$ is -1. Flipped rows and column are highlighted in red. The number of bit flips is reduced from 6 to 4.

would satisfy Formula (3.5), i.e. -2 + 0 + 4 >0. Consequently, the overall cost of the write operation could be reduced from 6 bit flips to 4 after flipping the select column and rows as shown in Figure 13(b). This example does not consider the cost of auxiliary bits for simplicity. Taking the cost of writing the auxiliary bits is tackled in the next section.

At last, the proposed optimization can be theoretically generalized so that a subset of rows and a subset of columns are flipped simultaneously. Nevertheless, selecting multiple rows and columns is not feasible from an implementation point of view because of the combinatorial number of row/column combinations. Accordingly, this thesis only considers the case of selecting one row with multiple columns and one column with multiple rows. Moreover, applying the optimizations is a purely design choice. As a matter of fact, the evaluation section shows that CAFO outperforms the existing schemes even without applying any optimization.

3.1.5 Cost of Auxiliary Bits

So far, CAFO's encoding process did not consider the cost of writing the auxiliary bits. Taking the cost of the auxiliary bits into consideration is easy and simple. As stated before, the cost model of CAFO would compare the new data to be written on a block to the previously written data in order

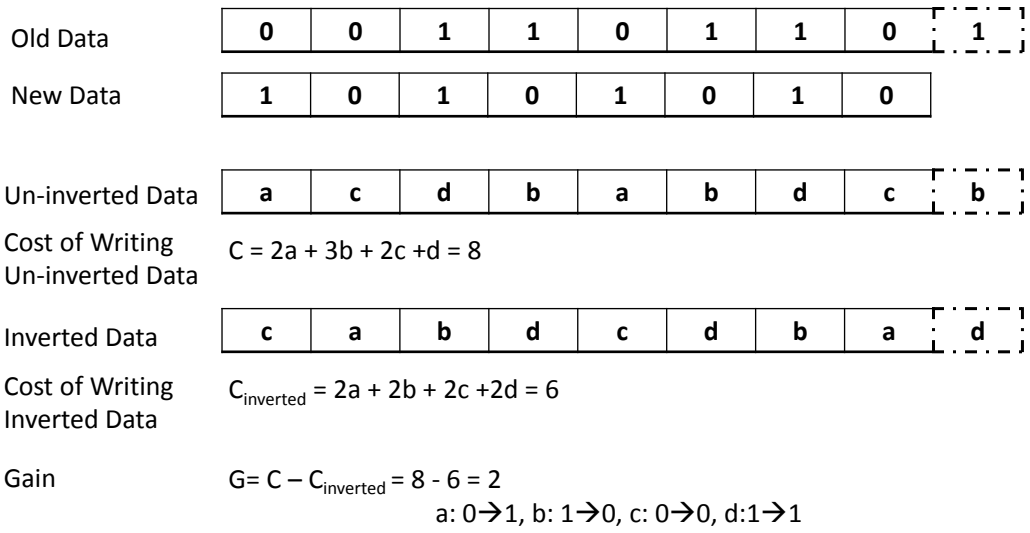

Figure 14: An example of taking the cost of the auxiliary bit into consideration when computing the gain. The costs of ”a”, ”b”, ”c” and ”d” are 1, 2, 0 and 0 respectively. Dotted cells represent auxiliary bits.

to form the differential vector. Accordingly, the previously written auxiliary bits are read within the step of forming the differential vector. Subsequently, the cost of the new auxiliary bits to be written can be taken into consideration during the encoding step of CAFO.

When computing the overall cost of writing the data in the un-inverted form, the overall cost has to be augmented by the cost of flipping ”1→0”, i.e. ”b”, when the bit value of the previously written auxiliary bit is ”1” and by the cost of flipping ”0→0”, i.e. ”c”, when the bit value of the previously written auxiliary bit is ”0”. Similarly, the overall cost of writing the data vector in the inverted form has to be augmented by the cost of flipping ”0→1”, i.e. ”a”, when the bit value of the previously written auxiliary bit is ”0” and by the cost of flipping ”1→1”, i.e. ”d”, when the bit value of the previously written auxiliary bit is ”1”.

Figure 14 shows an illustrative example. Since the old value of the auxiliary bit is “1”, it has to be flipped to “0” if the data is to be written in the regular form. Thus, the cost of the regular form increases by “b” as shown in the figure to capture the auxiliary bit flip cost. On the other hand, the data in the inverted form requires an auxiliary bit of value “1” to flag the inversion step. As the old value of the auxiliary bit is “1”, the inverted form of the data does not toggle the auxiliary bit and incurs no additional cost as a “d” flip has a zero cost.

3.1.6 Decoding Process

The decoding process of CAFO is very simple. For each cell, it is enough to compute the exclusive OR (XOR) of its corresponding row and column auxiliary bits. If the outcome of the XOR operation is 0, then the bit value of the cell is to be retrieved without inversion. Otherwise, the bit value of the cell has to be inverted after retrieval. As a matter of fact, an outcome of "0" implies that both auxiliary bits have either a bit value of "1" or "0". A bit value of "1" for both auxiliary bits implies that the corresponding row and column have been both flipped which leaves their intersecting cell un-inverted. Thus, the value of that cell must be retrieved un-inverted. Similarly, a bit value of "0" implies that the cell must be retrieved without inversion as none of the corresponding row and column has been flipped. When the outcomes of the XOR operation is 1, then either the row or the column has been flipped which leaves their intersecting cell inverted. Thus, the value of the intersecting cell has to be retrieved inverted. Algorithm 3.2 captures the decoding process where *VX* and *VY* represent vertical and horizontal auxiliary bits of a 2D data block b with dimensions n and m. The decoding algorithm loops through data cells at lines 3 and 4. At line 5, the exclusive OR of the corresponding vertical and horizontal auxiliary bits of each cell is computed. If the outcomes of the exclusive OR is "1" then the cell is read inverted at line 6. Otherwise, the cell is read un-inverted at line 8.

Algorithm 3.2 CAFO DECODING

1: $//VX[], VY[] : Aux.bitsarrays$
2: $//b[n][m] : DataBlock$
3: **for** $i = 0..n$ **do**
4: **for** $j = 0..m$ **do**
5: **if** $(VX(i) \oplus VY(j))$ **then**
6: *Read* $\overline{b[j][j]}$
7: **else**
8: *Read b[i][j]*
9: **end if**
10: **end for**
11: **end for**

3.1.7 Design Considerations

The implementation of CAFO necessitates a space overhead and could incur a marginal performance overhead. As far as space overhead is concerned, the major overhead comes from the assigned auxiliary bits. Applying CAFO on a block of size $n \times m$ requires $n + m$ auxiliary bits. Hence, the space overhead can be computed as $\frac{n+m}{n \times m}$.

As for the performance overhead, it is of great importance not to incur latency on the read path which CAFO achieves successfully. Retrieving a data vector requires a simple XOR operation to determine whether a cell value is to be retrieved in the regular or inverted form as indicated by Algorithm 3.2. Hence, CAFO does not excessively penalize the read operation and maintains its performance.

On the other hand, CAFO's encoding process could incur a computational overhead on the write path. Nevertheless, write data are typically buffered before being written to memory in an iterative manner in PCM [18]. Hence, the computational overhead is incurred while the data is buffered and is off the critical write path. As a matter of fact, the write path depicted in Figure 15 includes a (1) differential XOR logic in order to determine the difference between the old and new data, (2) a cost model logic to set the cost of write operations in terms of "a", "b", "c" and "d" flips and (3) an encoding logic that manipulates the data into a form that cuts down the write cost. While the differential XOR logic and cost model logic are simple, the computational overhead comes from the encoding logic that consists of two steps: the first is the flip operation and the second is the switching of the flips in between the rows and columns as captured in Algorithm 3.1. The overhead of the aforementioned steps is discusses next.

When it comes to the flip operation, the gain of every row or column is computed in accordance to Formula (3.2). If the gain is positive, then the corresponding row or column has to be flipped and the corresponding auxiliary bit has to be set. CAFO's flip operation is identical to Flip-N-Write [15], which is characterized by being simple and fast. Hence, the flip operation incurs an affordable overhead.

While the flips at either the level of the rows or the columns can be executed in parallel, the flip switching between rows and columns has to be serialized which could incur some delay. Nevertheless, our assessments shows that CAFO requires about 4 alternations on average for a

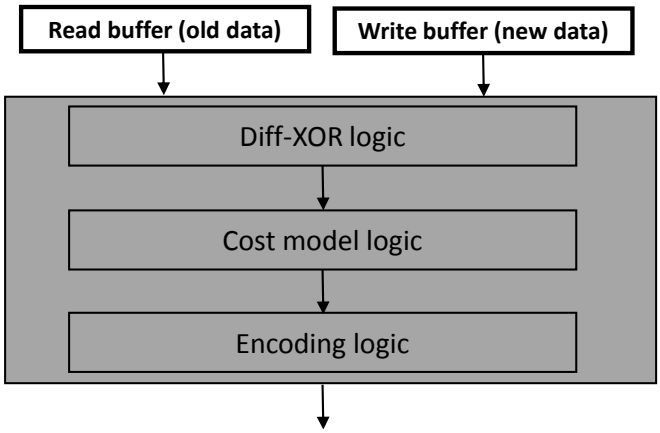

Figure 15: Write datapath logic for CAFO.

32B block, e.g. rows flip→columns flip→rows flip→columns flip→rows flip, before terminating the encoding process. Thus, CAFO's encoding process does not incur a substantial computational overhead overall.

As indicated in Section 3.1.4, applying the encoding optimization is a design choice. Next, the potential overhead of the optimization step, if it is to be implemented, is discussed. Figure 16 depicts the logic required for the optimization where a single column and multiple rows could be selected for a 4×4 block. For each row ri intersecting with a selected column c, the expected gain (EG) is computed as $EG_i = G_{ri} - 2g_{ri,c}$ where G_{ri} is the gain of the row ri computed in accordance with Formula (3.2) and $g_{ri,c}$ is the gain of flipping the intersecting cell between row ri and column c as defined in Section 3.1.4. The calculation of EG is derived from Formula (3.5) that implies that the expected gain of each row is its initial gain subtracted from the gain of flipping the intersecting cell multiplied by two.

A positive, EG, implies that flipping a row could lead to an overall cost reduction of the write operation. Accordingly, a control flag signal, F, has to be set to mark rows with positive EG as depicted in Figure 16. At last, the total expected gain for the rows with positive EG is computed and added to the gain of column c, G_c, to form the overall gain, $G_{overall}$. If $G_{overall}$ is positive, then column c and the rows with their control signal F set are to be flipped.

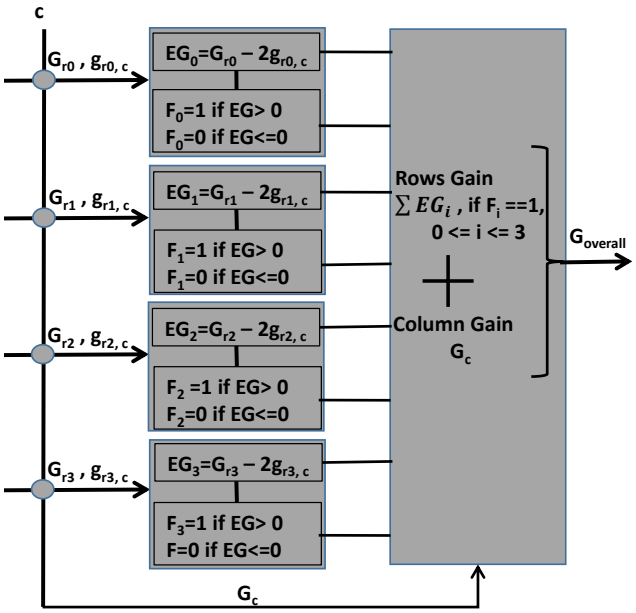

Figure 16: Optimization logic. EG is the expected gain of flipping a row. G is the initial gain of rows or columns. $g_{r,c}$ is the gain of flipping the cell intersecting at row r and column c. F is a control signal to flag positive EG. $G_{overall}$ is the overall gain of flipping the selected column with the rows with $EG > 0$.

As a matter of fact, Figure 16 should be perceived as a potential implementation of Formula (3.2). In practice, every column has to be tested with multiple rows to check if further cost reduction can be achieved. However, those checks can be executed in parallel. Hence, the optimization process is expected to be fast without incurring a notable delay. At last, it is worth mentioning that the above implementation holds equally for a row selected with multiple columns.

3.2 EVALUATION

To evaluate CAFO, several experiments are conducted. The results are compared against the state of the art Flip-Min and Flip-N-Write (refer to Section 2.2.1). As the various schemes need to be assessed at the same level of space overhead, the block sizes for both Flip-Min and Flip-N-Write are picked such that their corresponding space overhead matches the overhead of the block size

chosen for CAFO. The goal of the experimental work is to compute the average cost reduction per write operation achieved by the various schemes. Moreover, Flip-Min and Flip-N-Write are made cost aware through integrating the proposed cost model with them. Subsequently, the write cost against Flip-Min and Flip-N-Write is compared while being cost aware and cost oblivious. Furthermore, the effect of the suggested encoding optimization on the write cost is studied through reporting the cost reduction achieved with and without applying the optimization. Experiments are conducted with data from random input streams and from SPEC benchmarks [29] collected traces. In addition, PCM is modeled through setting the costs of bit flips that would match the characteristics of the underlying technology. Specifically, the cost of bit flips "c" (0→0) and "d" (1→1) are set to 0 and the cost of bit flips "a" (0→1) and "b" (1→0) are varied. Differential Writes (refer to Section 2.2.1) is chosen as the baseline scheme against which all other schemes are compared. Differential writes flips half of the data bits on average for random input streams.

3.2.1 Random Input Streams

To evaluate CAFO with random input streams, Monte Carlo simulations are conducted. For each iteration, two random data vectors are generated. The first represents the old data vector and the second represents the new data vector to be written. Furthermore, the old values of the auxiliary bits are randomly generated and the cost of setting their new values is took into consideration. CAFO is evaluated with block sizes of 32B, 128B and 512B that incur a space overhead of 12.5%, 6.25% and 3.125% respectively. For Flip-N-Write, the block size granularity is varied so that an equal space overhead to CAFO is reached. As for Flip-Min, the block size is kept at 8B as proposed by the authors. However, the number of assigned auxiliary bits is changed to match the overhead of CAFO. Reducing the number of auxiliary bits for Flip-Min lowers its corresponding space overhead but decreases the number of elements per coset.

3.2.1.1 Cost Reduction

In this section, the performance of the various schemes is assessed in terms of the average cost per write. "Cost" is used as the evaluation metric which is computed as the average cost incurred by each scheme relative to Differential Write normalized against CAFO. Moreover, the cost of bit flip

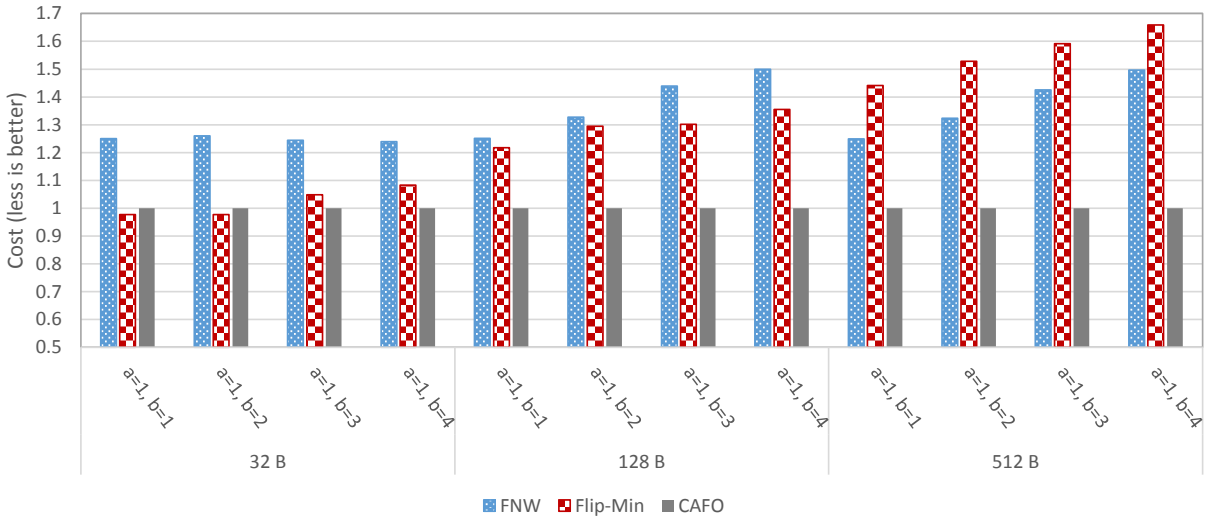

Figure 17: Cost reduction achieved by CAFO over Flip-N-Write (FNW) and Flip-Min.

“a” is fixed to 1 and the cost of bit flip ”b” is varied in order the capture the write asymmetry in PCM with ”a = 1” and ”b = 1” representing the cost reduction in terms of minimizing the number of bit flips. Figure 17 shows the results across different block sizes.

Compared to Flip-N-Write, CAFO is capable of reducing the write cost across all block sizes and for different costs of ”a” and ”b” flips. Actually, the average cost reduction per write operation is at least 24% for a block size of 32B and up to 49% for a block size of 512B. As a matter of fact, the results indicate that Flip-N-Write is more affected by the reduction in space overhead than CAFO. Another observation to note is that an increase in the cost of bit flip ”b” would yield in more cost reduction achieved by CAFO across all block sizes.

As for Flip-Min, the results reveal that CAFO is capable of reducing the write cost across all block sizes except for a block of size 32B with a cost of ”b” flip of either 1 or 2. Nevertheless, a block of 32B incurs a space overhead of 12.5% making it an unlikely design choice. An interesting observation to note is that when the space overhead of Flip-Min is reduced (bigger block size), its cost reduction capabilities reduce significantly. As a matter of fact, Flip-Min falls short to CAFO by incurring up to 65% more cost with block sizes of 128B and 512B.

At last, it is worth mentioning that the reported results have to be interpreted in the context of

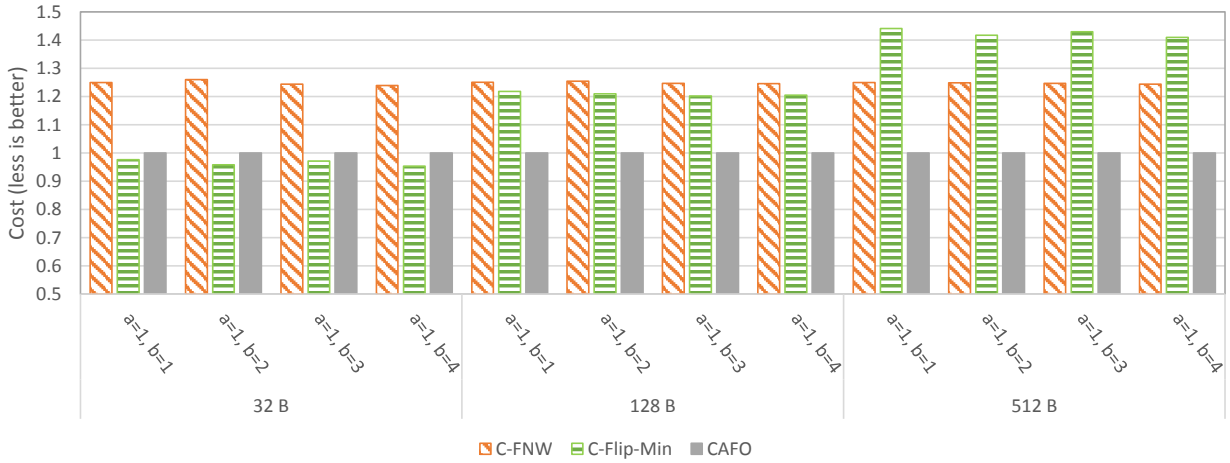

Figure 18: Cost reduction achieved by CAFO over cost aware Flip-N-Write (C-FNW) and cost aware Flip-Min (C-Flip-Min).

the memory being modeled. For instance, a reduction in PCM write cost leads to a deceleration in the wear out rate of memory cells. Accordingly, CAFO helps PCM to preserve its limited endurance through reducing the write cost. Consequently, the reduction in write cost would lead to an extended lifetime of PCM memory chips. Hence, it can be said that CAFO allow PCM chips to undergo up to 65% more writes compared to both Flip-Min and Flip-N-Write.

3.2.1.2 Cost Model Integration

In the previous section, Flip-N-Write and Flip-Min are simulated as presented by their authors i.e. oblivious to the cost asymmetry of bit flips. Nevertheless, both schemes can be made cost aware through integrating our cost model with them. Accordingly, C-FNW and C-Flip-Min are presented as the cost aware versions of both Flip-N-Write and Flip-Min respectively. Subsequently, the results in Figure 17 are reproduced with all schemes being cost aware as shown in Figure 18.

The collected results indicate that CAFO is still capable of reducing the write cost over C-FNW across all block sizes and for different costs of ”a” and ”b” flips. Nevertheless, the gap in between the two schemes has been narrowed to up to 25% cost reduction. Such a result is a direct consequence to the integration of the cost model with Flip-N-Write.

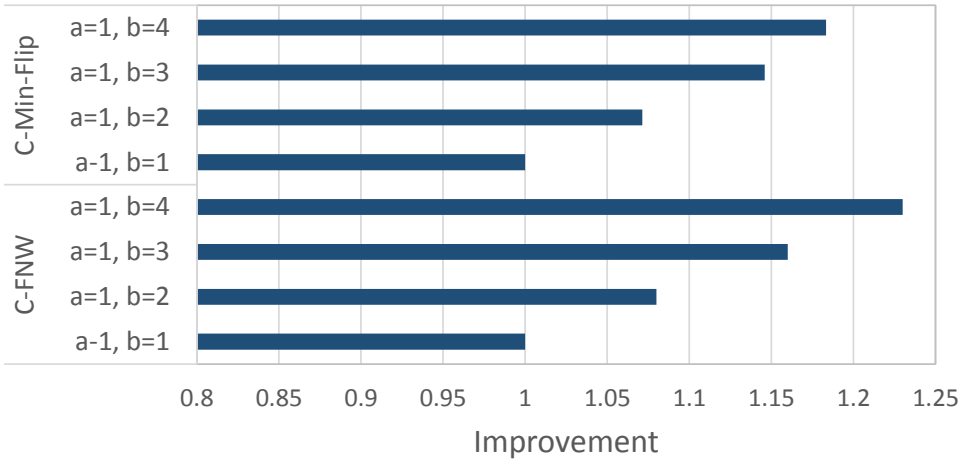

Figure 19: Cost reduction improvement of cost aware Flip-N-Write (C-FNW) and Flip-Min (C-Flip-Min) over cost oblivious Flip-N-Write and Flip-Min through the integration of CAFO's cost model.

When it comes to C-Flip-Min, CAFO incurs by up to 2% more cost for a block size of 32B as a result of integrating our cost model. Nevertheless, C-Flip-Min incurs by up to 44% more cost than CAFO for blocks of sizes 128B and 512B. Similarly to Figure 17, C-Flip-Min keeps the same trend of a reduced cost reduction capability with space overhead of 6.25% and 3.125% (block sizes of 128B and 512B) respectively and falls short to both C-FNW and CAFO.

The results in Figure 18 are indicative of the importance of taking the actual cost of bit flips into consideration for write cost minimization. As a matter of fact, Figure 19 shows that integrating the cost model with Flip-N-Write and Flip-Min has improved the cost reduction capability of both schemes by up to 23% and 18% relative the cost oblivious versions of Flip-N-Write and Flip-Min respectively for a block size of 32B. Actually, a data flip operation in cost oblivious Flip-N-Write can lead to an increased write cost as the flip decision is solely based on the number of bit flips. In contrary, a data flip operation in cost aware Flip-N-Write always leads to a reduction in the write cost as the flip decision is based on the actual cost of individual bit flips. Similarly, a cost oblivious Flip-Min would pick the coset element that yields in the least number of bit flips while a cost aware Flip-Min would pick the coset element that yields in the least cost. Hence, the cost model brings a substantial improvement to both Flip-Min and Flip-N-Write. Accordingly, a simple integration

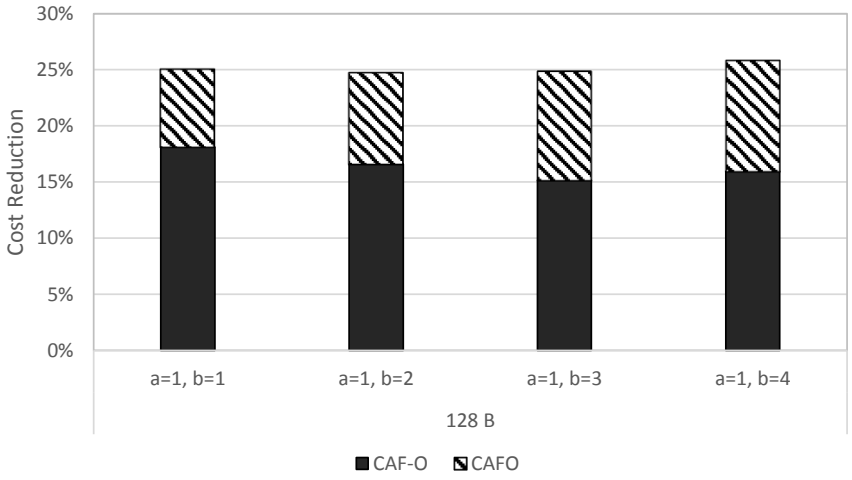

Figure 20: Contribution of CAFO minus optimization (CAF-O) to the cost reduction over C-FNW for a 128B block.

of the proposed cost model to memory system already deploying Flip-N-Write or Flip-Min could extend the lifetime of the memory system by up to 23%.

3.2.1.3 Optimization Isolation

The results that are presented thus far assume that CAFO applies the encoding optimization presented in Section 3.1.4. Nevertheless, applying the optimization is a design choice and could be dropped from CAFO's implementation. Accordingly, the effect of the optimization on the cost reduction achieved by CAFO can be isolated. To this end, CAFO minus Optimization (CAF-O) is presented and its cost reduction capability is compared against regular CAFO. Figure 20 shows the contribution of CAF-O to the cost reduction over cost aware Flip-N-Write (C-FNW) for a 128B block with space overhead of 6.25%.

The results in Figure 20 indicate that the encoding scheme of CAFO still manages to achieve a cost reduction above 15% over C-FNW across different costs of "a" and "b" flips even without applying any optimization. This achievement conforms to the assumption that the encoding optimization is optional, which renders the application of the optimization a trade-off between the cost reduction capability and the desired encoding complexity.

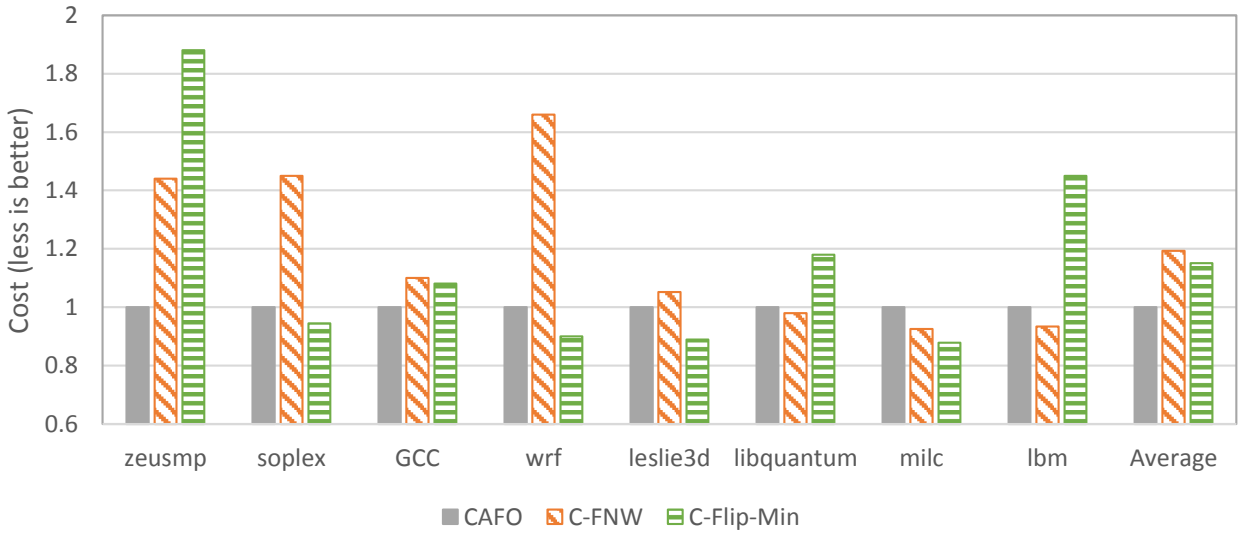

Figure 21: Cost reduction achieved by CAFO over cost aware Flip-N-Write (C-FNW) and Flip-Min (C-Flip-Min).

3.2.2 Benchmark Data

In this section, CAFO is evaluated against other schemes using real data from SPEC 2006 benchmark programs. To this end, Virtutech Simics [52] is used to collect PCM main memory traces that contain command identifier and address of each memory request. The trace files include the previous and new data vectors for each write request. Subsequently, the trace files serve as an input to an in-house trace-driven simulator. The in-house simulator maintains a list of auxiliary bits for each write address within the trace files. Consequently, this list is updated after every write to retain the new values of the auxiliary bits. Furthermore, the in-house simulator processes each record within the trace files and reports the average cost per write request for CAFO, C-FNW and C-Flip-Min in accordance to a preset cost of bit flips for ”a”, ”b”, ”c” and ”d”.

Figure 21 shows the cost reduction achieved by CAFO over C-FNW and C-Flip-Min for a 128B. The plot reveals that CAFO is capable of reducing the cost on average by up to 19% and 15% more than C-FNW and C-Flip-Min respectively. The reported results conform with those of random data presented in Figure 18 for a 128B block, where CAFO reduces the cost on average by

up to 25% and 20% more than C-FNW and C-Flip-Min respectively. Our analysis reveals that the 5% loss with real data is attributed to programs that exhibit a low frequency of bit flips between the old data and new data (around 10% to 12%) i.e. a high similarity between the old and new data of write requests. Nevertheless, CAFO is capable of performing well when the frequency of bit flips is relatively high.

3.3 CONCLUSION

Reducing the cost of servicing write requests at the pre-write fault avoidance stage is essential to streamline the endorsement of PCM into the memory system. Accordingly, this chapter presented CAFO as a new scheme that targets minimizing the endurance cost incurred by the write operation. CAFO encompasses a cost model that captures the asymmetry in flipping bit values of memory cells. In addition, CAFO applies a new encoding technique that is integrated with the cost model to minimize the cost of write operations.

In summary, this thesis makes the following key contributions at the pre-write fault avoidance stage:

1. It highlights the endurance cost asymmetry for bit flips of 0→1 and 1→0 manifested by PCM writes. Consequently, it presents a cost model that can capture the aforementioned asymmetry. The evaluation reveals that the proposed cost model can reduce the cost of servicing write request for existing schemes by up to 23% .
2. It presents a new encoding scheme that logically maps a block of memory into the 2D space. The encoding methodology is characterized by a tunable space overhead and can reduce the write cost over existing schemes by up to 65% .

Al last it is worth noting that although CAFO is applied to reduce the endurance degradation in this thesis, it can be used for other applications. The flexibility of the cost model makes CAFO capable of being applied to other objectives such as write energy reduction and write latency improvement. As a matter of fact, the encoding engine of CAFO is not tied to a specific application. It encodes the data to reduce any cost that it is fed to it through the cost model.

4.0 POST-WRITE FAULT TOLERANCE

While pre-write fault avoidance techniques are effective in preserving the endurance of PCM cells and decelerating the endurance degradation rate, wear outs are unavoidable. Also, parametric and random variations in the manufacturing process induce a non uniform distribution of cells lifetime. Consequently, early failure of cells are common. To this end, multi-bit error correction schemes to combat cell wear-outs error are essential at the post-write fault tolerance stage.

PCM worn out cells exhibit a stuck-at fault model. That is, a cell gets stuck permanently at a specific bit value with the ability of being subsequently read but not reprogrammed [46]. The fact that a failed stuck-at cell is still readable makes the nature of errors to be data dependent. In a sense, a faulty cell is erroneous only when the bit value read is different than the intended bit value to be written. For example, if a cell is stuck-at-1 then an error occurs only when the cell needs to be written with 0. Consequently, a stuck-at cell can be classified into either "stuck-at-wrong" (SA-W) or "stuck-at-right" (SA-R) depending on whether the cell gets written with a value different or identical to the value it is stuck at. Henceforth, the errors that an error correction scheme needs to mask in the context of PCM correspond to those cells that happen to be SA-W after a given write operation.

This chapter presents *Data inversion* [48] as a technique to increase the number of faults that a conventional error correction code such as BCH can tolerate through exploiting the data-dependent nature of errors. In addition, this chapter introduces extensions to *RDIS* [50, 53] which is an error correction scheme designed specifically for PCM's fault model. The proposed extensions are capable of increasing the number of faults that RDIS can tolerate as well as decrease its incurred space overhead.

4.1 POWER OF ONE BIT: INCREASING ERROR CORRECTION CAPABILITY WITH DATA INVERSION

Data Inversion is a simple technique which can increase the number of faulty cells that an error correcting scheme, such as BCH [7], can recover from at a very small cost. Given an error correcting scheme with the capability of masking t errors, a write operation, in the context, of PCM fails only when $t + 1$ faults are SA-W. Following a block write failure, *data inversion* attempts an additional write operation while applying a transformation on the data to be written. This transformation consists of inverting the data within each cell of the protected block. Flipping the data changes the classification of the faulty cells from SA-W to SA-R and has the effect of increasing the number of faults that can be accrued within a block before $t + 1$ errors can be manifested. In this realm, *data inversion* can be looked at as a data mapping technique offering the choice between two encodings, one un-inverted and one inverted. When the un-inverted encoding results in an unrecoverable error pattern, an inverted encoding is attempted and is highly likely to induce a recoverable error pattern. Hence, *data inversion* can complete an otherwise failed write operation successfully. It is worth mentioning that *data inversion* may attempt an additional write operation with transformed data only after the number of stuck-at faults within a protected block exceeds the nominal capability of the error correcting code. Nevertheless, the need for an additional write is rare due to the data dependent nature of errors as will be shown in the evaluation section.

Clearly, *data inversion* needs to introduce an additional polarity bit to record the inversion step. Accordingly, two variations of *data inversion* are presented. In the first, the polarity bit is integrated as part of the codeword. In the second, the polarity bit is taken out of the codeword. This thesis shows that the additional number of stuck-at faults that can be tolerated depends on the distribution of the faults within the protected block in the first variation. As for the second variation, this thesis shows that the additional number of stuck-at faults that can be tolerated is constant.

While the concept of applying an inversion has been used before, i.e. reducing the write endurance cost as in CAFO and FNW (refer to Chapter 3), this thesis shows and proves the effect of applying an inversion on the data post to a write failure. Hence, the inversion concept is used for a different application and to achieve a different goal.

4.1.1 Theoretical Foundation

Before delving into the details of the data inversion scheme, let us start with some preliminary definitions that will set the grounds for consequent discussions.

Definition 1. *A memory/storage block is non-defective if any data pattern can be written successfully on the block.*

Definition 2. *A memory/storage block is defective if some data patterns cannot be written successfully on the block.*

Definitions 1 and 2 establish the distinction between a defective and a non-defective block. A defective block is not free of stuck-at faults, but these faults can never lead to a pattern of errors uncorrectable by the error correcting code. Conversely, the stuck-at faults in a defective block can lead to write failures. However, not every write request necessarily fails on a defective block since some of the faulty cells may be stuck at the value that is to be written i.e. SA-R. In fact, failures are related to specific data patterns that end up in a number of SA-W cells beyond the capability of a deployed error correcting code.

4.1.2 Integrated Protection

When a write request is submitted, the data cells are augmented with a polarity bit that is set to zero. Subsequently, the error correcting code computes the auxiliary information to protect against errors in the original data cells and the polarity bit. Next, the codeword (data cells + polarity bit + auxiliary cells) is physically written. If the codeword manifests a number of SA-W cells greater than the error correcting capability of the code, then the codeword is recomputed with inverted data and the polarity bit set to one to flag the inversion step.

Applying an inversion on the data has the potential effect of increasing the number of stuck-at faults that can be tolerated within the data cells, while preserving the non-defectiveness of the memory block as described in the following theorem.

Theorem 1. *Given a memory/storage block protected by an error correcting code that can correct up to t-bit errors, applying data inversion with integrated protection extends the correction capability of the code to $Q + R$ faults such that $Q/2 + R = t$, where Q is the number stuck-at faults in*

the data bits and R the number of stuck-at faults in the auxiliary bits. That is, the block is defective only if $(Q/2 + R) > t$.

Proof: By construction, an error correcting code of capability t fails only when at least $t+1$ errors are manifested i.e. $t+1$ SA-W faults in the context of the stuck-at fault model. In the worst case, the number of SA-W cells is equal to the number of SA-R cells within the data part after a write failure. Therefore, $Q/2$ stuck-at faults can happen to be SA-W in the worst case after inverting the data cells. In addition, at most R errors can be manifested within the auxiliary bits in the worst case as recomputing the auxiliary information does not necessarily change the value of every auxiliary bit. Hence, a memory/storage block becomes defective only if $(Q/2 + R) > t$.

It follows from Theorem 1 that data inversion makes the defectiveness of memory blocks correlated with the distribution of the stuck-at faults between the data and the auxiliary bits within the blocks. It is highly likely that the distribution of faults allows for increasing the number of faults within a block before it becomes defective. For example, let $Q = 6$, $R = 3$ and $t = 6$. Although the number of stuck-at fault ($Q + R = 9$) is greater than the correction capability of the error correcting code ($t = 6$), the protected block is not defective. In fact, the maximum number of errors that can occur when data inversion is integrated is $Q/2 + R = 6$ which is within the nominal capability of the error correcting code. Hence, data inversion can be looked at as increasing the capability of the error correcting code.

Encoding/Decoding Example

When a write request is issued, data inversion with integrated protection augments the data cells with the polarity bit set to 0. Subsequently, the error correcting code computes the auxiliary information with a correction capability that covers the polarity bit. Next, the codeword (data bits + polarity + auxiliary bits) is physically written on the PCM medium. In the event that the codeword exhibits a number of SA-W cells above the capability of the error correcting code, an extra write operation is needed. Before submitting the second write attempt, the data is inverted and the polarity bit is set to one. If the second write attempt exhibits a number of SA-W cells within the reach of the error correcting code, then the write request has completed successfully. At read time, the data cells and the polarity bit are retrieved after the error correcting code decodes the codeword and corrects errors. When the value of the polarity bit is different than 0, the data cells are inverted to obtain the intended data that had to be written initially.

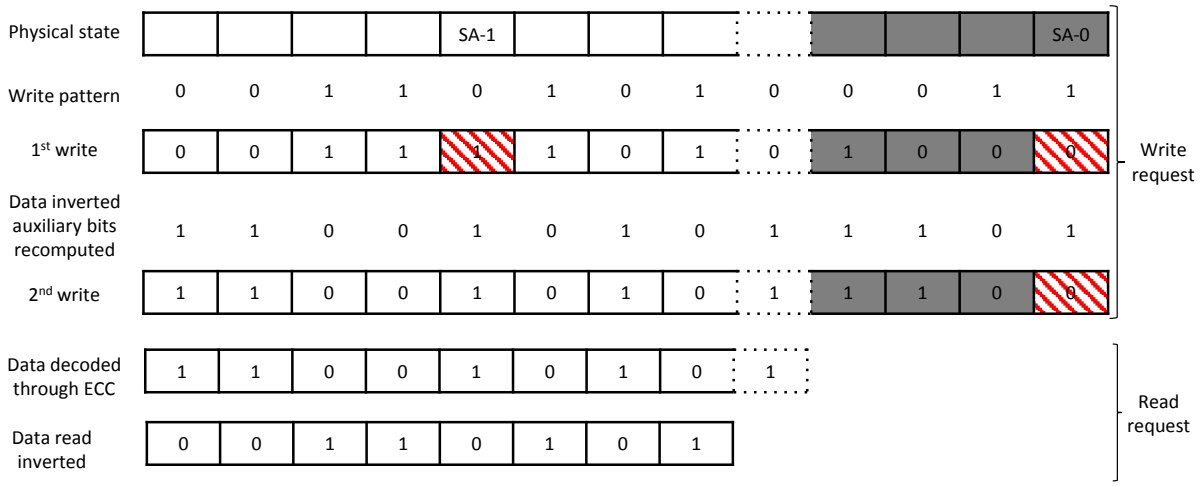

Figure 22: An example of executing write and read requests with integrated protection complementing an error correcting code of capability 1. Dotted cells represent the polarity bit, grey cells represent the auxiliary bits and red patterned cells represent errors.

Fig. 22 gives an example where a byte is protected by an error correcting code of capability 1 complemented with data inversion with integrated protection. Protecting a byte against one error requires a Galois field of size 4, which is large enough to accommodate the need to augment the data bits with the polarity bits. Fig 22 shows that the first write attempt fails as 2 SA-W cells are manifested. Consequently, a second write is to be attempted with inverted data bits. Accordingly, the data bits and the polarity are inverted, the auxiliary bits are recomputed and a second write is submitted. The second write completes successfully as only one SA-W cell is manifested, which is within the nominal capability of the ECC code. At read time, the error correcting code is capable of recovering from the single error and correctly decodes the data bits and the polarity bit. As a last step, the data bits are read inverted since the value of the polarity bit is 1. This example shows that data inversion is capable of completing an otherwise failing write request successfully and extends the number of faults before a block turns defective.

4.1.3 Un-integrated Protection

Data inversion is a post failure technique. Actually, data inversion interferes with the write operation only after a failure occurs. As long as the number of stuck-at faults accrued in a block is still within the capability of the deployed error correcting code, no write failure can occur. Accordingly, the polarity bit does not start to be toggled until that point is reached. Even after the number of stuck-at faults gets above the capability of the error correcting code, the polarity is not always toggled due to the data dependent nature of errors.

Given its infrequent toggling, separating the polarity out of the protection scope of the error correcting scheme is an option to be considered. Such an approach could make data inversion vulnerable to a single point of failure. Nevertheless, the raw endurance of the polarity bit is expected to be resilient enough to sustain its infrequent toggling. Furthermore, the vulnerability to polarity bit failures can be mitigated through redundancy techniques such as triple modular redundancy (TMR). Taking the polarity bit out of the codeword brings two enhancements to data inversion. The first enhancement is to abolish the need to recompute the auxiliary information prior to the second write attempt. By separating the polarity bit, both the data and auxiliary cells are inverted without recomputing the auxiliary information. Accordingly, with un-integrated protection the additional write is a simple inversion of the entire codeword. The second and most important enhancement is that separating the polarity bit out of the codeword guarantees a constant additional number of faults that can be tolerated. Theorem 2 proves the conjuncture.

Theorem 2. *Given a memory/storage block protected by an error correction code that can cover up to t-bit errors, applying data inversion with un-integrated protection can correct up to $2t+1$ stuck-at faults. That is, the block is defective only after $2t+2$ stuck-at faults are accrued.*

Proof: By construction, an error correcting code of capability t fails only when at least $t+1$ errors are manifested. Since applying data inversion exchanges the role of SA-W and SA-R, at least $t+1$ SA-R faults in addition to $t+1$ SA-W faults have to exist in the block for the error correction code to fail after inverting the codeword to be written. Thus, a memory/storage block becomes defective only after $2t+2$ stuck-at faults are accrued.

It follows from Theorem 2 that un-integrating the polarity bit out of the codeword allows an error correction code to be capable of extending the non-defectiveness of a block until the number

of faults within the block becomes double the capability of the error correcting code irrespective of the distribution of the faults. Before that point is reached, the probability of defectiveness is 0. For example, consider a memory block containing 12 stuck-at faults and protected with an error correcting code of capability 6 ($t = 6$). Augmenting the error correcting code with data inversion renders the block non-defective as the maximum number of SA-W that can be manifested is $12/2 = 6$ which is within the capability of the error correcting code.

Encoding/Decoding Example

Data inversion with un-integrated protection separates the polarity bit from the codeword, which alters the execution flow of write and read requests. When a write request is issued, the error correcting code computes the auxiliary information with correcting capability that does not cover the polarity bit. Subsequently, the polarity bit is set to 0 and it is physically written along with the codeword (data cells + auxiliary cells) into the PCM medium. In the event of a write failure, an additional write operation is attempted with the codeword inverted and the polarity set to one. At read time, the codeword is read inverted if the value of the polarity bit is set to 1. Finally, the data cell are retrieved through decoding the codeword by the error correcting code.

Figure 23 shows an example of a byte of memory protected by an error correcting code of capability 1 complemented with data inversion with un-integrated protection. The example shows that the first write attempt fails as a consequence of 2 SA-W cells. Subsequently, a second write is attempted with inverted codeword and the polarity bit set to 1. The second write is successful as it ends up with one SA-W which is within the reach of the deployed error correcting code. At read time, firstly the codeword is retrieved inverted as the value of the polarity bit is 1. Secondly, the codeword is decoded to retrieve the data cells as where intended to be read initially. Thus, data inversion with un-integrated protection completes an otherwise failing write request successfully.

4.1.4 Defectiveness Probability

So far, two approaches that can be followed to couple data inversion with error correcting codes to increase the number of faults that can be tolerated within a protected block have been presented. It was shown that with integrated protection, the additional number of faults that we can protect depends on the distribution of the accrued faults within a block between the data and auxiliary cells.

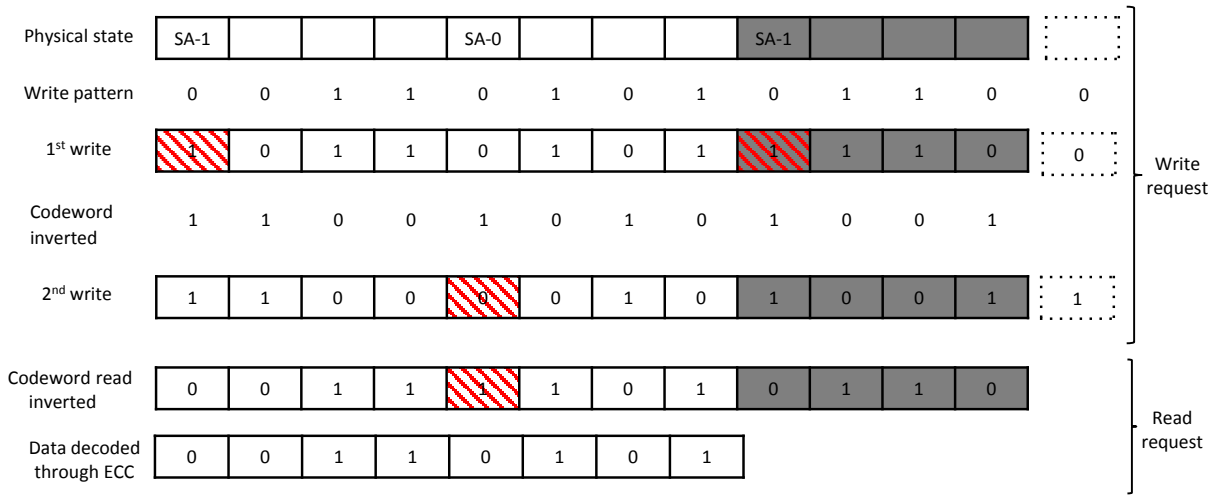

Figure 23: An example of executing write and read requests with un-integrated protection complementing an error correcting code of capability 1 protecting one byte of memory. Dotted cells represent the polarity bit, grey cells represent the auxiliary bits and red patterned cells represent errors.

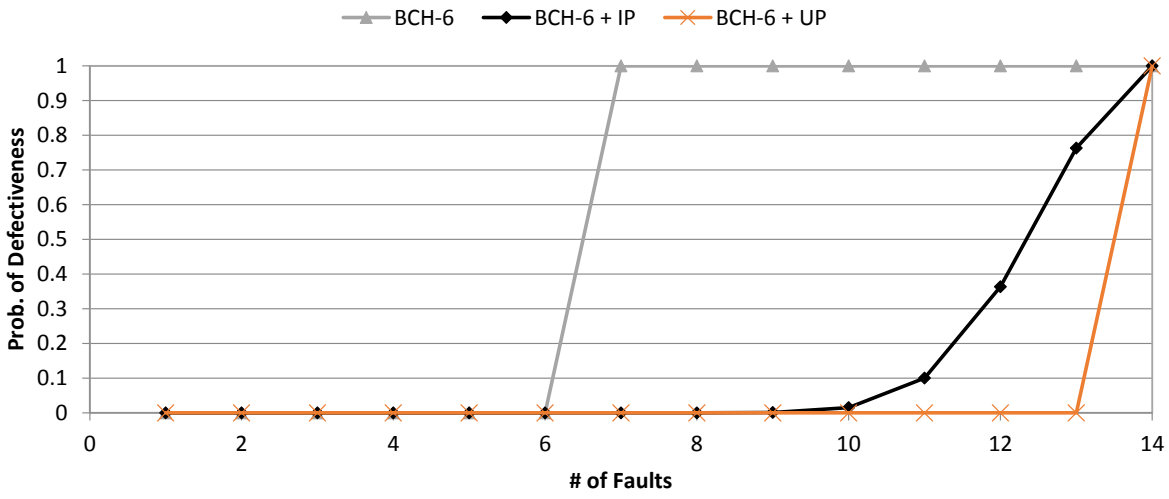

Figure 24: Probability of defectiveness as a function of the number of stuck-at faults, where a BCH-6 protects a block of size 512 bits. "IP" denotes integrated protection and "UP" denotes un-integrated protection.

On the other hand, un-integrated protection guaranties an increase of a constant number of faults irrespective of any distribution. Fig. 24 captures the difference between the integrated and un-integrated protection approaches when it comes to the probability of defectiveness as a function of the number of faulty cells within a block. For all approaches, a block cannot turn defective before the number of faults within the block is beyond the nominal capability of the deployed error correcting code i.e. 6. Once the number of faults increases beyond 6, a probability of defectiveness forms for the integrated protection approach. Nevertheless, this probability increases slowly with the relative increase in the number of faults within the block. When it comes to the un-integrated protection scheme, the probability of defectiveness remains 0 until the number of faults attains 13.

4.1.5 Surviving Polarity Bit Defectiveness with Un-integrated Protection

Since data inversion with un-integrated protection does not protect the polarity bit, the defectiveness of that bit may result in making data inversion failing to meet the premise of increasing the number of faults that can be tolerated within a block. However, the number of faults that can be tolerated is never below the nominal capability of the error correcting code even if the polarity bit is defective. Imagine that a polarity bit is defective by manufacturing and stuck at a value different than 0. As the polarity bit is not protected, covering the faulty cell is not possible. However, the codeword data can always be inverted to match the status of the polarity bit, in which case the error correcting code can cover errors within the inverted codeword up to the nominal capability.

To gain insight about the effect of a faulty polarity bit on blocks' defectiveness with un-integrated protection, Table 1 reports the probability of defectiveness of the un-integrated protection approach where the polarity bit could turn defective for a 512-bit block protected by a BCH-6 code. Similarly to integrated protection, defectiveness with un-integrated protection can now occur when the number of faults exceeds the nominal capability of the error correcting code. Yet, it is notable that the probability of defectiveness for the un-integrated approach remains low until the number of faults within the block exceeds the threshold that the un-integrated approach guarantees to tolerate. Those results are attributed to the fact that the probability of the polarity bit turning defective is low.

Allowing operation with a defective polarity bit is possible, but it changes an important aspect

# of faults	7	8	9	10	11	12	13
BCH-6 + UP with PC defectiveness	0.012	0.013	0.015	0.017	0.019	0.020	0.022

Table 1: Probability of defectiveness for un-integrated protection because of a defective polarity bit as a function of the number of faults within a 512 bit block protected by a BCH-6 code.

of data inversion. That is, an additional write attempt may be required only after the number of faults exceeds the nominal capability of the error correcting code. Nevertheless, the target of this thesis is to present a technique that exploits the major fault model of PCM, which is worn-out cells. In general, memory manufacturers follow a rigorous testing phase to eliminate blocks where certain cells are defective by manufacturing. Even though manufacturers allow variations in cells lifetime, a minimal level of endurance has to be achieved. Accordingly, the case of continuing operations with a defective polarity bit is not considered. In the event of a defective polarity bit by manufacturing or because of low endurance, the block associated with the polarity bit can be declared defective and mapped out of the address space. Yet, redundancy techniques could be applied to mitigate the unlikely event of polarity bit failures as indicated in the previous section.

4.1.6 Error Detection and Block Retirement

So far, our discussion of data inversion did not address two important points. The first is how to determine the successfulness of a write operation while the second is how to handle the failure of the second write operation. Since PCM's major fault model is stuck-at faults resulting in hard errors, a standard practice is to apply a read-after-write (RAW) operation to verify that the data was written successfully [53, 73, 76]. Given the data dependent nature of errors, RAW discovers which cells are SA-W along with their stuck-at bit value. Thus, the successfulness of the write operation is determined based on the number of SA-W cells discovered by the RAW operation. Accordingly, data inversion requires one RAW operation after each write attempted to determine

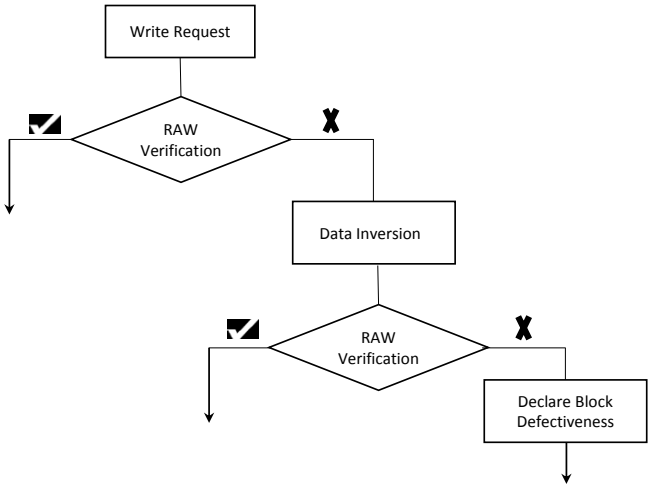

Figure 25: Flow of executing a write request. "RAW" denotes read after write.

the successfulness of the write operation.

After the failure of the first write, the data in the protected block are inverted and a second write operation is submitted. The failure of the second write attempt implies that the protected block turned defective and that data inversion cannot tolerate the faults in that block. Hence, such a block has to be retired and mapped-out of the address space. The execution flow of data inversion for a write request is captured in Fig. 25.

4.1.7 Evaluation

To assess the performance of data inversion, the lifetime that can be achieved when an error correcting code complemented with data inversion in comparison to the lifetime that can be achieved with the same error correcting code without data inversion is noted. In addition, the overhead of the additional write operation that data inversion may require is quantified. At last, the effect of the parametric process variation on the lifetime is studied. It is worth noting that a number of error correcting schemes to mask stuck-at faults have been proposed [53, 76, 73]. Nevertheless, the road map to endorse those schemes in real system implementation is still unclear. Error correcting codes, such as BCH, are an industry standard that have been thoroughly implemented and opti-

mized in functional systems. Unlike data inversion that is a post-failure technique, other proposed schemes are proactive techniques that try to delay the occurrence of write failures. To the best of our knowledge, data inversion is the first technique that targets extending the defectiveness criteria of memory blocks beyond the nominal capability of the deployed error correcting code. Hence, the goal of this chapter is not to compare with other existing schemes but to present a simple architectural technique capable of increasing the number of faults that error correcting codes can cover.

4.1.7.1 Experimental Setup

To evaluate data inversion, Monte Carlo simulations were conducted. Since a detailed simulation of a large memory capacity is impractical within a reasonable time span, 2,000 pages of main memory and secondary storage each of size 4KB were considered. For main memory, it is assumed that each page is an aggregation of 512-bit cache line size blocks. To protect a cache line, a BCH code of capability 6 (BCH-6) is deployed. BCH-6 incurs a physical overhead amounting up to 12%, which is within the generally accepted overhead assigned to error correcting at the level of main memory. As for secondary storage system, it is assumed that each page is an aggregation of 512-byte sector size blocks protected by BCH-20. The choice of the error correcting code capability follows the need to recover from more than 20 errors in a sector size block in NAND flash [82]. To generate write traffic, data from various real file types ranging from MPEG videos and JPEG images to PDF documents were collected.

To assign lifetime to the memory cells of simulated blocks, a Gaussian distribution with a mean of 10^8 and a standard deviation of 25×10^6 [73, 4] was used. It is assumed that an efficient wear levelling scheme [67, 75, 64] distributes writes evenly across the available blocks so that we achieve comparable wear-out rate. A memory block is retired after a write request fails to be written successfully as a result of a number of errors that could not be masked. In addition, the possibility for the polarity bit to wear out when data inversion with un-integrated protection is in use is taken into consideration . That is, in the rare event of a polarity bit wearing out before its associated memory block becomes defective, the memory block is retired. At last, the simulation methodology takes into consideration the possible need for extra write operations and simulates

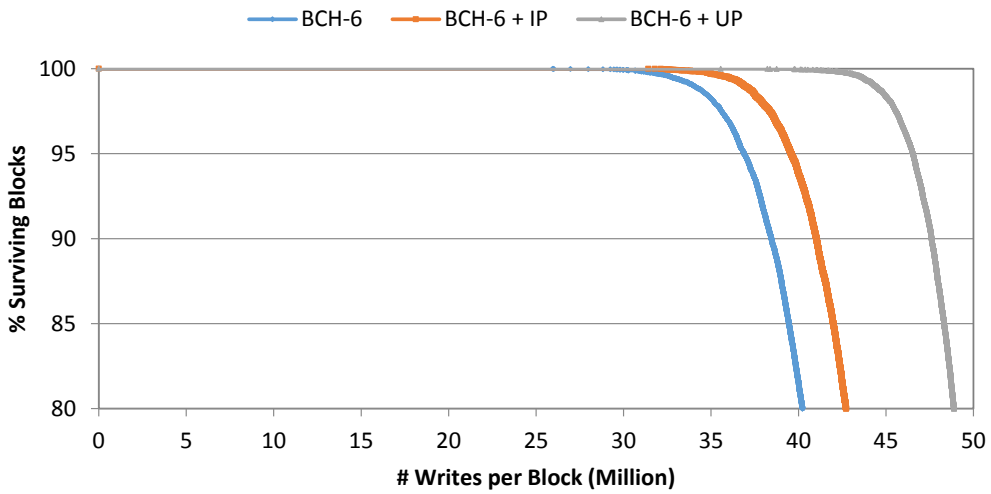

Figure 26: Lifetime of PCM main memory blocks achieved with BCH-6 and BCH-6 complemented by data inversion with integrated protection (IP) and un-integrated protection (UP).

them accurately. Overall, the methodology is similar to the one in [73].

Because the main failure mode of PCM is hard faults, it is safe to assume that the BCH code capability is dedicated to masking hard faults. In case other failure modes, such as transient errors, become an issue in the future, a separate BCH capability must be provisioned to recover from them, which is orthogonal to this work.

4.1.7.2 Main Memory Lifetime Improvement

To study the effect of data inversion on the lifetime of main memory, the lifetime achieved with data inversion complementing a BCH-6 code is compared to that of the lifetime achieved by the BCH-6 code itself. The evaluation metric is the total number of writes executed on memory blocks before retirement. The lifetime until 20% of the memory blocks are retired is studied. Fig. 26 shows the collected results where the Y axis represents the percentage of surviving blocks as a function of the number of writes per block represented by the X axis.

It is clear from Fig. 26 that data inversion is capable of substantially improving the lifetime

of memory chips. After retiring the first memory block, data inversion extends the lifetime over BCH-6 by 21.1% and 37% with integrated and un-integrated protection respectively. It is worth noting that data inversion with un-integrated protection significantly surpasses data inversion with integrated protection in terms of achievable lifetime. This result is attributed to the fact that un-integrated protection guarantees an additional number of faults to be tolerated while the number of additional faults that can be tolerated with integrated protection depends on the distribution of faulty cells within the memory block between the data cells and the auxiliary cells.

Overall, protecting memory chips solely with a BCH code leads to an earlier retirement of the blocks. Adding data inversion delays the defectiveness of the memory blocks leading to a significant lifetime improvement.

4.1.7.3 Secondary Storage Lifetime Improvement

To evaluate the impact of data inversion on storage devices, the lifetime in terms of the number of writes per block achieved with a BCH-20 code complemented with data inversion versus a regular BCH-20 code is noted. Conversely to main memory chips, block retirements that cause the degradation of the actual storage capacity are not allowed in storage devices. As a matter of fact, block defectiveness is combated through over-provisioned spare blocks. Consequently, an over-provisioning of an additional 20% of the total storage capacity is assigned as spares which is a typical practice in server products. In this sequel, a storage device remains in operation until the defectiveness of the first sector that cannot be replaced with a spare. Fig. 27 plots the lifetime of storage blocks while providing 20% worth of spares.

Similarly to main memory, the results show that when the first storage block is retired, data inversion extends the lifetime over BCH-20 by 19.1% and 24.9% with integrated and un-integrated protection respectively. It is notable that the difference in lifetime between integrated and un-integrated protection is smaller than that of main memory. As a matter of fact, having a BCH code of high capability and a bigger block size increases the likelihood of having a distribution of faults in between the data bits and the auxiliary bits that preserves the non-defectiveness of a storage block. In fact, the larger the number of faults that can be tolerated the more likely for the faults to be distributed in between the data cells and the auxiliary cells.

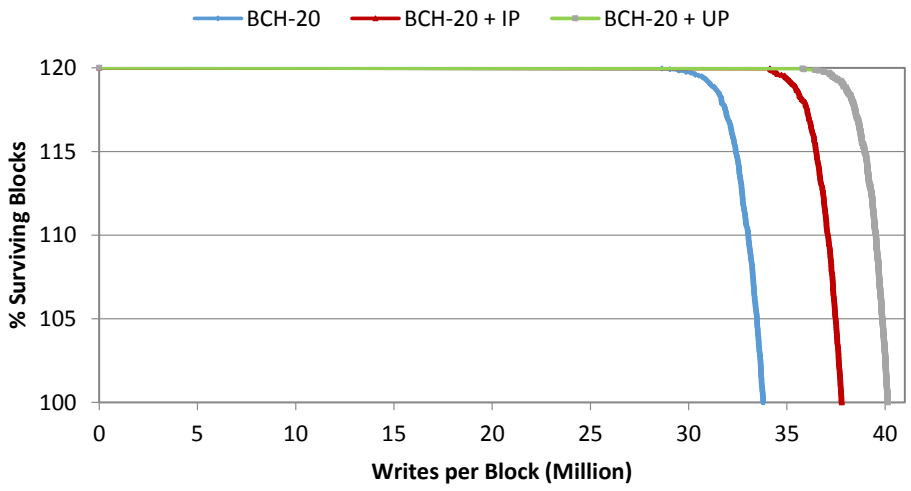

Figure 27: Lifetime of PCM storage blocks achieved with BCH-20 and BCH-20 with integrated protection (IP) and un-integrated protection (UP). This experiment assumed that 20% of spare storage capacity was provided.

Lastly, it is noticeable that the overall lifetime improvement for a storage device is less than that for a main memory chip. In fact, spares have smaller impact on the lifetime when data inversion complements a BCH code than with the regular code itself. Since data inversion increases the number of stuck-at faults that can be tolerated within storage blocks, it delays the retirement of the blocks. Nevertheless, the number of blocks nearing their lifetime limit increases. Consequently, spares are allocated at a higher rate once defective blocks start to occur. Our findings indicate that when the BCH code complemented with data inversion uses the first spare block, 5% of the spares have already been allocated by the BCH code solely. Yet, the allocation rate of spares with data inversion increases notably from that point on.

4.1.7.4 Lifetime Variability

As PCM scales to small feature sizes, the manufacturing process is expected to yield in high variability when it comes to the lifetime of memory cells. Accordingly, the impact of data inversion

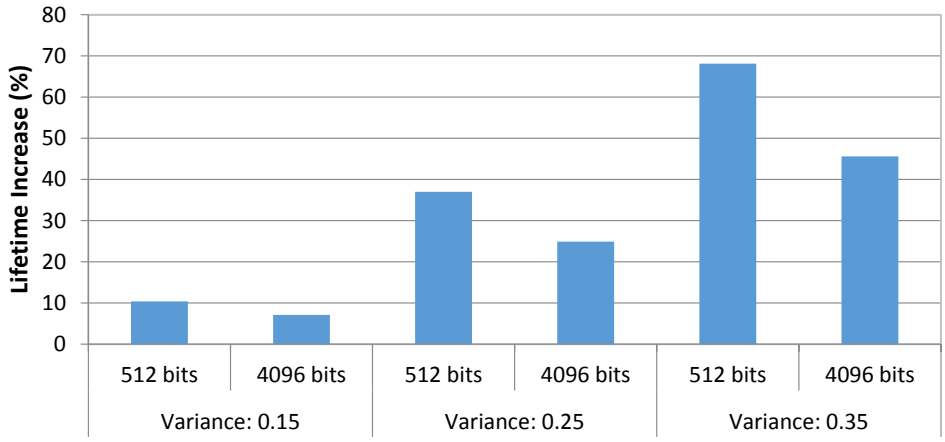

Figure 28: Lifetime increase with BCH complemented with data inversion relative to regular BCH with various cell lifetime variance.

on lifetime in light of the imperfect process control is assessed. To this end, the lifetime of memory blocks while varying the cell lifetime variance is evaluated. That is, the lifetime mean is fixed to 10^8 while varying the lifetime variance to 0.15, 0.25 and 0.35 respectively. Our evaluation metric is the lifetime increase achieved through complementing a BCH code with data inversion relative to the regular BCH code after the retirement of the first memory block. To calculate this metric, the total number of writes performed with data inversion is subtracted from the total number of writes performed by the regular code and the difference is divided by the latter. Fig. 28 shows the increase in lifetime achieved by data inversion with the un-integrated protection scheme.

Fig. 28 reveals that data inversion can increase the lifetime across all variances. Yet, smaller lifetime increase is noted with smaller variability in cells' lifetime. In fact, if the cells within a memory block exhibit low lifetime variability, then a large number of cells are expected to wear out within close proximity. Accordingly, this domino effect makes data inversion less effective in extending the lifetime. Nevertheless, data inversion still manages to increase the lifetime of memory blocks by up to 10.4% with a variance of 0.15. On the other hand, increasing the number of faults that an error correcting code can cover has a significant effect on the lifetime when the variability in cell lifetime is high. For example, a striking lifetime increase is marked with a cell

lifetime variance of 0.35. As a matter of fact, a high variability in lifetime implies that cells within a memory block vary between notably weak and strong cells. Consequently, increasing the number of faults that can be covered has a significant effect on lifetime as it allows to cover weak cells that fail early in the lifetime of memory blocks and preserves enough capability to cover cells that fail later. Hence, coupling an error correcting with data inversion achieves a significant lifetime improvement with high variability in cell lifetime; a case which is common due to the imperfect process control with a very deep sub-micron technology.

4.1.7.5 Performance Overhead

When a write request fails to complete successfully, data inversion submits another write request after inverting the data. Nevertheless, data inversion does not interfere with the write process except when failures actually occur. That is, after the number of faults within a protected block exceeds the nominal capability of the error correcting code. Hence, the potential overhead incurred through the second write operation is not a constant overhead that every write request pays. Even after the number of faults exceeds the capability of the error correcting code, an extra write operation is not always required.

Fig. 29 reveals that the probability of the first write to fail for a 512-byte block is low even after the number of faults within the block significantly exceeds the capability of the BCH code due to the data dependent nature of errors. For example, with 34 faults the probability of the first write failure is only 10%. Overall, writing on a block having a number of faults above the capability of the error correcting code would still succeed with a high probability.

To quantify the overhead of additional write operations that could occur, their performance overhead should be isolated. To this end, the number of additional write operations that a block undergoes before it is retired is counted. In Table 2, the results for a cache line size block and a storage sector size block protected by a BCH of capabilities 6 and 20 are showed.

The collected results show that no performance overhead in terms of additional writes is incurred before the number of faults within a block exceeds the nominal capability of the error correcting code. Thereafter, only 4.9% to 13.1% of the additional write operations that data inversion allows require a second write operation. Hence, data inversion increases the number of faults that

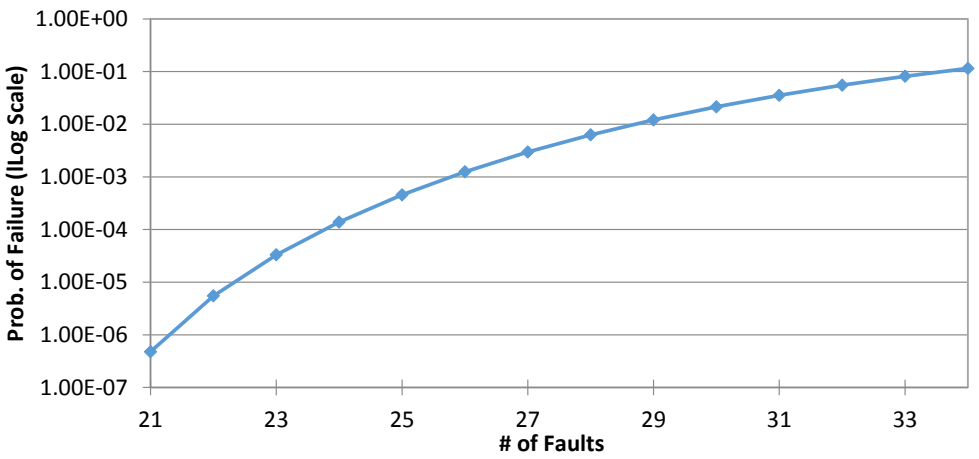

Figure 29: Probability of the first write failure on a 512-bytes storage block protected by a BCH-20 code.

can be tolerated within blocks, salvages otherwise decommissioned blocks and notably augments the write volume of memory blocks while incurring an affordable overhead.

Lastly, it is worth noting that the overhead of the second write with the integrated protection scheme is less than that of the un-integrated scheme. As a matter of fact, our evaluation shows that the integrated protection scheme extends the lifetime of memory blocks less than the un-integrated protection scheme. Consequently, the number of additional successful writes achieved with the integrated protection scheme is less than that of the un-integrated scheme when both schemes are compared against a regular code. Hence, the un-integrated protection scheme incurs more extra writes than the integrated protection scheme.

4.1.8 Summary

Data inversion is an architectural technique capable of allowing error correcting codes to tolerate a number of faults greater than their nominal capability. After a write request fails to complete successfully, an inversion on the data pattern to be written is applied. Consequently, this inversion step is likely to bring the number of erroneous cells within the capability of the error correcting code. Thus, data inversion completes successfully a write request that would otherwise have failed

	Integrated Protection		Un-Integrated Protection	
	Before nominal capability is exceeded	After nominal capability is exceeded	Before nominal capability is exceeded	After nominal capability is exceeded
512 bits	0%	4.9%	0%	13.1%
4096 bits	0%	4.1%	0%	8.9%

Table 2: Performance evaluation in terms of extra write operations required by data inversion to complete write requests successfully after the number of faults exceeds the nominal capability of the error correction code.

which extends the usability of memory blocks.

This thesis brings the following key contributions:

1. Proves that applying an inversion on the write data after a write failure is capable of increasing the number of faults that an error correcting code can tolerate within a memory block while requiring a single additional polarity bit. In addition, it presents two settings in which data inversion can be applied. The first protects the polarity bit and increases the number of faults that can be tolerated by the error correcting code based on the distribution of faults within the protected memory block. The second separates the polarity bit from the codedword and guarantees a constant increase in the number of faults that can be tolerated by the error correcting code.
2. Shows that data inversion can significantly increase the lifetime of PCM by up to 37% while incurring an affordable performance overhead. The evaluation shows that the overhead in terms of a second write operation is incurred only after the number of faults within a block exceeds the nominal capability of the deployed error correcting code.
3. Studies the effect of data inversion on the lifetime in light of imperfect manufacturing process

control. The conducted study shows that data inversion copes with the variability in cells' endurance and is especially effective when the variability is high, which is expected to be a common manufacturing phenomenon.

In conclusion, data inversion is an architectural technique that effectively addresses the critical endurance reliability issue that PCM suffers from. Data inversion is powerful, yet simple enough to be readily incorporated into computer systems under realistic usage scenarios.

4.2 RDIS EXTENSION

Conventional error correcting schemes such as BCH and SEC-DED are designed for a general fault model. In fact, those schemes can equally recover from transient and hard errors. On the other hand, some other error correcting schemes are designed for a specific fault model. For instance, RDIS (refer to Section 2.2.2) is an error correcting scheme designed specifically for the stuck-at fault model prominent in PCM. RDIS' encoding identifies a set containing all the SA-W cell through a 2D grouping of the protected memory block. At decoding time, the identified set is retrieved inverted which guarantees reading the correct bit values that were intended to be written within the SA-W cells. As any other correcting scheme, RDIS has a limited capability when it comes to the number of faults that it can cover. Specifically, RDIS capability is constrained by the formation of a specific pattern of faults within a protected memory block (details are in Section 2.2.2). This thesis makes the contribution of presenting two techniques capable of recovering from RDIS' deadlock pattern; thus allowing the scheme to recover from additional faults. Also, a major overhead of RDIS is the number of auxiliary bits it introduces to retain the encoding/decoding information. This thesis makes as well the contribution of presenting a technique to reduce the space overhead required by RDIS to encode and decode data.

4.2.1 Block Repair

For RDIS to halt, the accumulated faults within a protected block have to form a specific pattern i.e. a loop of faults (refer to Section 2.2.2). Put simply, such a pattern makes it impossible to identify an invertible set as the size of the initial sub-array in the encoding process can never be reduced to zero. Recovering from the halting patterns can be achieved through the following two techniques.

The first technique, *Pointer Break*, consists of allocating an error correcting pointer [73](ECP) that consists of a pointer entry in addition to a patch cell that are to be used to break the loop of faults. The pointer entry specifies the location of a stuck-at cell that is substituted by the patch cell. Choosing any faulty cell in the loop of faults is enough to break the defective pattern.

The second technique, *Shift Break*, consists of changing the mapping of the cells into the

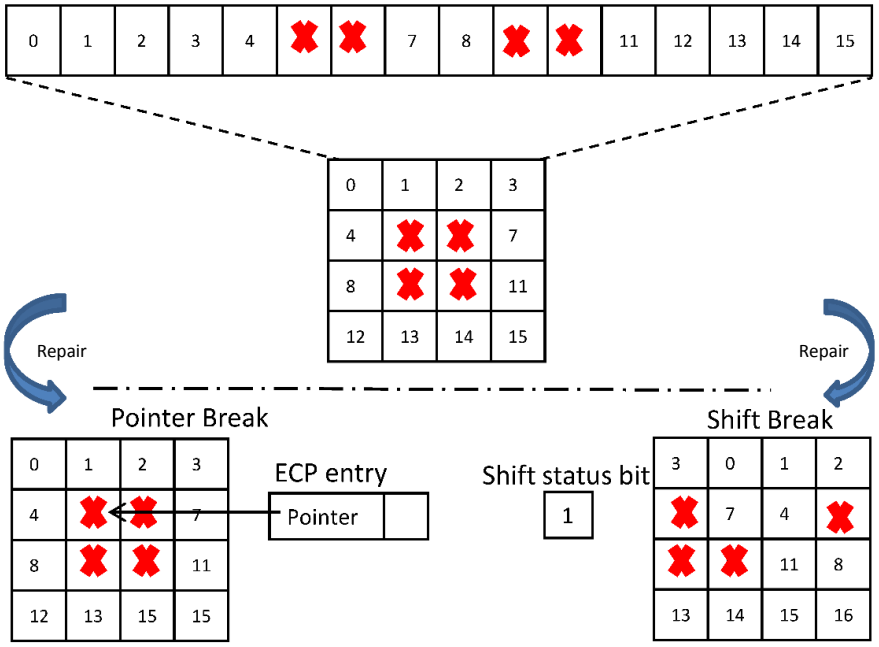

Figure 30: Defective block fixing techniques.

logical 2D $n \times m$ structure after a defective pattern is formed. One possible implementation would be to shift the position of a cell by its row number modulo m. Though shifting the cells in the block is not masking any of the stuck-at faults, it is likely to cause the faulty cells to form a pattern that is not anomalous to RDIS. Both techniques are depicted in Fig. 30.

It is worth mentioning that both techniques are complementary to RDIS. Accordingly, it is a design choice to adopt either of them. *Pointer Break* guarantees the recovery from defective patterns. On the other hand, *Shift Break* cannot guarantee the recovery from defective patterns. While shifting the cells in the protected block breaks the existing defective pattern, other stuck-at faults within the block could form a new defective pattern. Nevertheless, the likelihood of this event is low.

To implement both techniques, *Shift Break* requires a simple remapping function and one additional auxiliary bit that serves as a flag to be set when a shift is applied. On the other hand, *Pointer Break* necessitates $lg(n \times m) + 1$ auxiliary bits to determine the location of the stuck at cell that

breaks the pattern. It also requires the determination of the cells that contribute a defective pattern in order to pick one of them to break the pattern. The evaluation section reports the additional number of faults that can be tolerated with both techniques

4.2.2 Multidimensional RDIS

In order to enable the determination and retrieval of the invertible set, RDIS adds a number of auxiliary counters as described in Section 2.2.2. These counters form the major space overhead of RDIS. As a matter of fact, RDIS represents an N bits memory block as a logical two-dimensional array of n rows and m columns. In addition, RDIS introduces $n + m$ counters that hold the auxiliary information. Accordingly, the physical overhead of RDIS is equal to $\frac{(n+m)\times k}{N}$ where k is the number of bits used for to each auxiliary counter. Interestingly, the concepts of RDIS are not limited to a logical two-dimensional grouping of memory cells. In fact, the grouping of cells can be extended up to d-dimensional, where d is $log_2(N)$. For example, a 512-bit memory block can be represented as a logical 9-dimensional array i.e. 2^9.

Extending the array dimensions of the logical representation of a memory block reduces the space overhead incurred by RDIS. For example, consider a 512-bit memory block. Assuming each auxiliary counter is 2 bits, the space overhead incurred by RDIS for a logical 2-D arrangement of 16×32 is 18.75%. If the number of dimension is extended to 3 i.e. $8 \times 8 \times 8$ logical block, the space overhead of RDIS is reduced to 9.37%. In this sequel, an additional increase in the number of dimensions further reduces the overhead of RDIS. Nevertheless, increasing the number of dimensions can have a diminishing return. In fact, the overhead cannot be further reduced when the sum of the dimensions of the d^{th}-D arrangement is equal to that of the d^{th+1}-D arrangement. Going back to our 512-bit memory block example, the space overhead of a 5-D logical arrangement, i.e. $4 \times 4 \times 4 \times 4 \times 2$, is 7.03%. Moving to a 6-D arrangement, i.e. $4 \times 4 \times 4 \times 2 \times 2 \times 2$, keeps the space overhead of RDIS at 7.03%. Thus, increasing the number of dimensions past 5 for a 512-bit block does not bring any merit in terms of space overhead reduction. Fig. 31 shows the decrease in overhead for RDIS as the number of dimension for a logical array arrangement of a 512-bit block increases from 2D to 5D.

On the flip side, RDIS loses some of its correction capability with each move to a higher

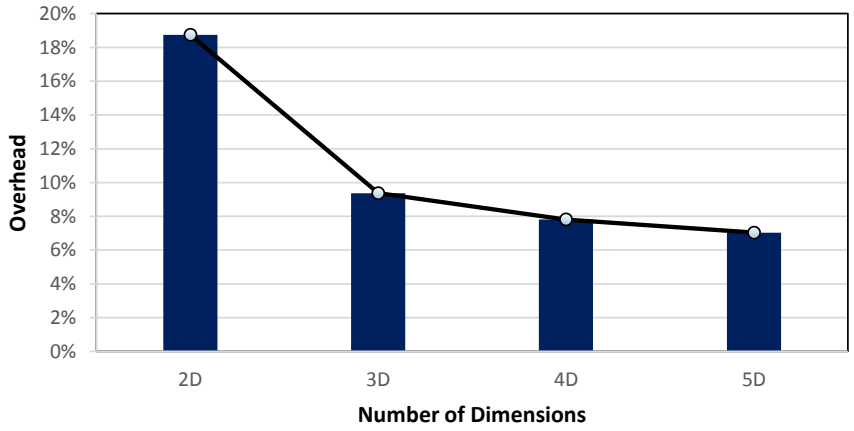

Figure 31: Space overhead of RDIS for a 512-bit block with various dimensional arrangements.

dimension. Nevertheless, multidimensional RDIS still guarantees the recovery of at least 3 faulty cells as in the 2D arrangement.

4.2.3 Block Repair Evaluation

In this section, Pointer Break and Shift Break are evaluated in terms of the average number of additional faults that can be tolerated after breaking a defective pattern in a memory block that suffers from k stuck-at faults.To this end, Monte-Carlo simulations are conducted. The simulations starts with a block that already has k faults and is defective with a loop of faults that is generated randomly. Subsequently, Pointer Break and Shift Break are applied to break the defective pattern and the additional faults that could be tolerated in the block until a new defective pattern is formed are counted. The simulation results are depicted in Fig. 32.

It is notable that both techniques are capable of significantly tolerating a large number of faults after fixing a block in which a defective pattern occurred with a relatively small number of faults. This finding is a direct consequence of the low probability of defectiveness that RDIS exhibits with a small number of faults in the block. Hence, fixing a block that got defective with a small number of faults yields into a greater number of faults that can be tolerated after the fix.

In addition, it is notable that fixing a defective block with a pointer performs better when the

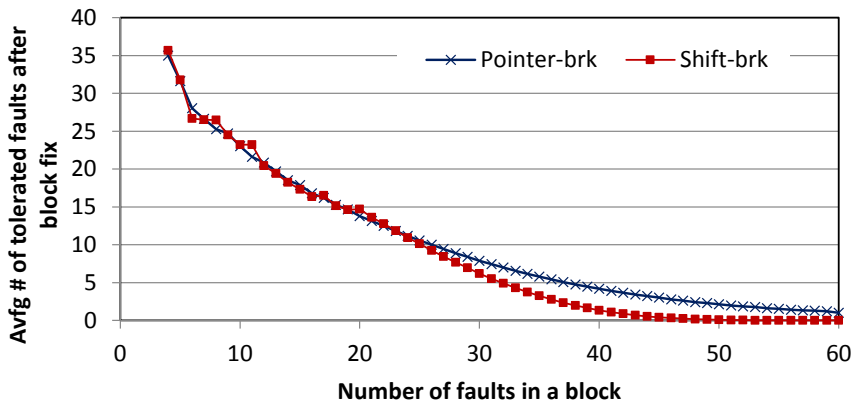

Figure 32: Avg. number of additional tolerated faults after breaking a defective pattern in a 2048-bits block size.

number of faults in the protected block is high. By shifting the cells in a block, the defective pattern is broken. However, a new defective pattern could form due to the large number of faults already existing in the block. On the other hand, fixing a block with a pointer is guaranteed to break the defective pattern. Nevertheless, implementing the shifting technique is simple and easy. It only requires one additional bit of overhead to indicate whether the data was written shifted and tolerates a significant number of additional faults.

4.2.4 Multidimensional RDIS Evaluation

Multidimensional RDIS consists of extending the number of dimensions of the logical array used to envision a protected memory block. We have shown that extending the number of dimensions reduces the space overhead incurred by RDIS. In this section, the impact of extending the number of dimension on the error correction capability of RDIS is studied. To this end, the average number of faults that can be tolerated by RDIS with a 2D, 4D and 6D arrangement of memory cells for 4096-bit block is reported where the number of bits per auxiliary counter is limited two. The maximum number of dimensions is limited to 6 as no further reduction in space overhead can be achieved past a 6D arrangement. In addition, multidimensional RDIS is compared to SAFER (refer to Section 2.2.1) where each configuration of RDIS is compared to that of SAFER k that

is either at the same level of space overhead or slightly larger where k is the number of groups a block is arranged into and $log_2(k)$ is the number of faults that can be tolerated. Moreover, Multidimensional RDIS is compared to BCH. A major difference between RDIS and BCH is the probabilistic nature of the error correction capability. While RDIS can tolerate a large number of faults beyond the threshold it guarantees with high probability, the capability of a deployed BCH code predetermines the maximum number of faults that can be tolerated. Similarly to SAFER, RDIS is compared to a BCH code of capability t that is either at the same level of space overhead or slightly larger where t indicates the maximum number of faults that BCH can tolerate. The comparison is held through conducting Monte Carlo Simulation in which faults are generated at random. Subsequently, the pattern of faults are analyzed to determine whether the corresponding error correcting scheme can cover them.

Fig. 33 shows the average number of faults that can be tolerated by each of the three schemes. The results reveal that RDIS maintains its superiority to other schemes even with higher dimensions. Nevertheless, RDIS loses some of its correction capability with the increase in the number of dimensions used to logically arrange a protected block. Our findings indicate that the probability of block defectiveness for RDIS, i.e. the probability of forming a loop of faults, increases with the increase in the number of dimensions. In fact, a new plane is added with each additional dimension. Each plane can be envisioned as a sub-block. With the increase in the number of dimensions, the size of each plane decreases. Hence, a smaller plane size implies a higher probability of defectiveness. This phenomenon is captured in Fig. 34 where the probability of defectiveness of RDIS for 4096-bit block with 2D, 4D and 6D arrangement is plotted.

4.2.5 Summary

RDIS is an error correcting scheme designed for the stuck-at fault model prominent in PCM. This thesis makes the following contribution to RDIS.

1. Presents two techniques, Pointer Break and Shift Break, capable of recovering from the detrimental fault patterns that cause RDIS to halt.
2. Presents Multidimensional RDIS as a variation of RDIS that decreases the incurred space overhead.

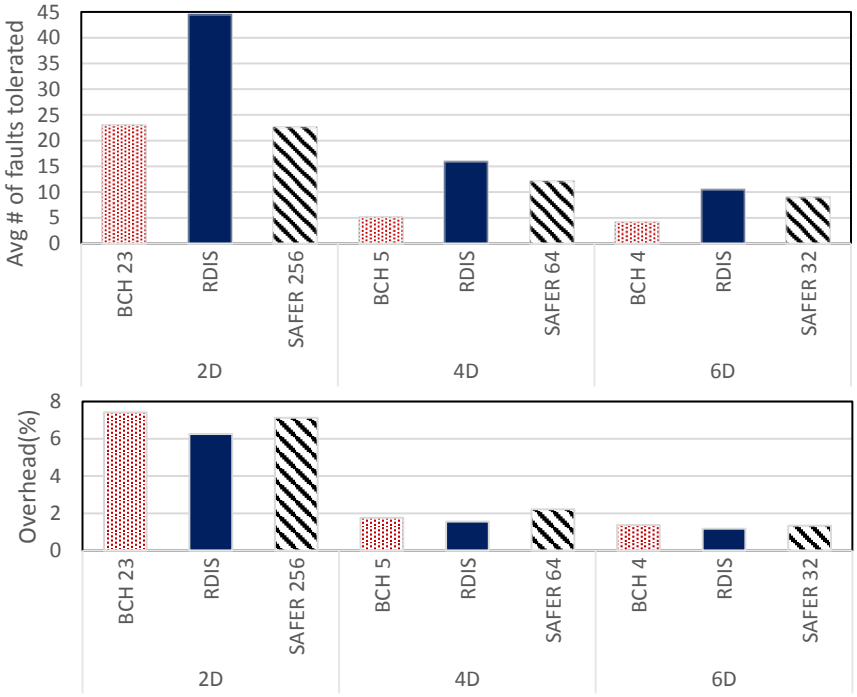

Figure 33: Average number of faults that can be tolerated by RDIS with various dimensional arrangement compare to SAFER and BCH for 4096-bit block.

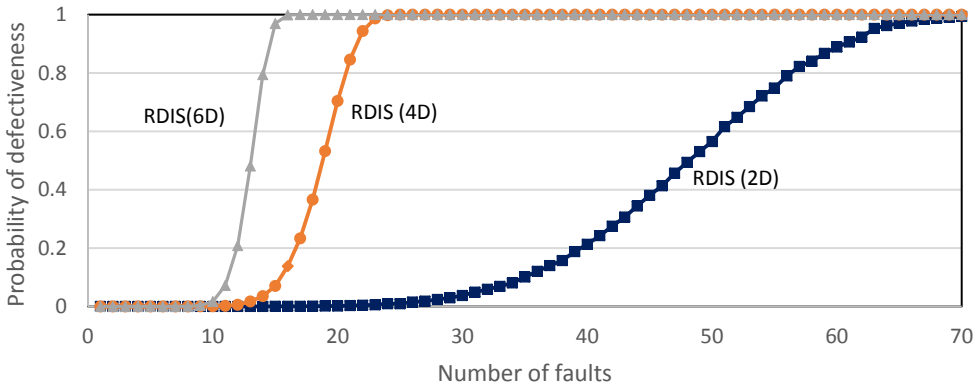

Figure 34: Probability of defectiveness with different dimensional arrangements for a 4096-bit block.

5.0 POST-FAILURE RECOVERY

Error correcting schemes to recover from cell failures are robust and reliable. Nevertheless, those schemes have a limited capability when it comes to the number of faults that can be tolerated. Thus, write failures are inevitable in memories subject to relatively low write endurance when the number of worn out cells within a protected block exceeds the capability of the deployed error correcting scheme. Such a block is labelled as "bad" as it cannot be written reliably any further. Accordingly, *bad block management* is a vital component to endurance limited memories such as PCM to preserve the dependability and performance of the underlying memory devices.

Bad block management reserves a number of memory block as spare blocks that are used to replace blocks that cannot be written reliably. When a write failure is detected for the first time on a block, the common practice is to label such a block as bad. Subsequently, bad block management replaces bad blocks with good spare blocks after the first write failure [55, 56]. This thesis shows that such a policy is inefficient in the context of PCM and that a better policy would be to delay the retirement of bad blocks through exploiting the data dependent nature of errors.

5.1 MOTIVATION

Given an error correction scheme capable of masking t errors, a write operation fails if $t+1$ or more errors are manifested. Accordingly, the probability of failing to write on a block with F faults of size N is $P(F,N) = \sum_{f=t+1}^{F} \binom{F}{f} (\frac{1}{2})^f \times (\frac{1}{2})^{(F-f)}$ for $F > t$. Fig. 35 plots the probability of failing to write on a block as a function of the number of stuck-at faults in the block. The plot assumes that the block is protected by an error correction scheme capable of fixing up to 20 errors.

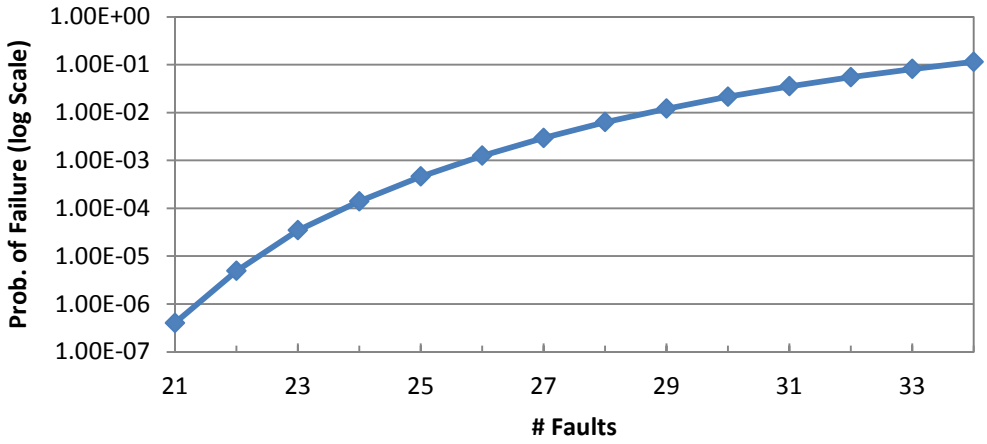

Figure 35: Block write failure probability vs. # of faults within a 4KB storage block, when an error correction mechanism covers up to 20 errors.

It shows that a block write fails rarely even if there are more than 20 faults. For example, when a block has 29 faults, the probability of block write failure, i.e., at least 21 errors are manifested, is only $\sim$1%. Clearly, retiring a block immediately when it becomes faulty (having more than 20 faults given the capability of the error correction mechanism) is overly conservative and fails to squeeze more lifetime from faulty, yet "better-than-bad" blocks. Accordingly, this thesis proposes *Data Dependent Sparing* [47], a new physical block sparing scheme that delays the retirement of a faulty block when the memory exhibits a stuck-at fault model. Instead of retiring a faulty block after the first write failure, data dependent sparing assigns a temporal spare to complete the current write operation and retains the faulty blocks. Since faulty blocks have a low probability of write failure, Data Dependent Sparing attempts a later write on the faulty block as it is likely to complete successfully.

Before delving into the implementation details, it is worth noting that Data Dependent Sparing is not specific to BCH-20. It can work and is equally effective with other error correcting capabilities of the BCH code as well as other error correcting codes. In fact, the evaluation section considers the application of Data Dependent Sparing with error correcting codes of with various capabilities.

5.2 DATA DEPENDENT SPARING

Data dependent sparing builds on the observation that faulty blocks have a low probability of actual write failure. This section describes in detail the idea of data dependent sparing, design trade-offs and its overheads.

5.2.1 Mechanism of Block Reuse

When an attempt to write to a block fails, i.e., there are more errors than can be corrected by the error correcting mechanism, a spare block replaces the original block. In conventional bad block management, referred to as "static sparing" in this thesis, this block is labeled "bad" and is discarded at once [62, 55, 56]. In data dependent sparing, the original block is not immediately retired because a later write to the same block with different data will likely succeed. When a later write to the block succeeds, the initially assigned spare is reclaimed to the spare pool, essentially increasing the write volume of the device without help from a spare. The concept of data dependent-sparing is illustrated in Fig. 36, where blocks 1 and 3 are defective. Fig. 36 (left) shows that writing to block 1 has failed while writing to block 3 has not. This has caused block 1 to be mapped to the spare block. However, a later write to block 1 was successful. Accordingly, the spare block was reclaimed and assigned to block 3 after a write request failed on it as depicted by Fig. 36 (right).

Essentially, *data dependent sparing* classifies storage blocks into "good", "better-than-bad" and "bad" (retired) and utilize both good and better-than-bad blocks on writes. After more write cycles are applied to a better-than-bad block, however, the block will wear out further and become notably unreliable. Fig. 35 shows that the probability of write failure increases with the number of faults in the block. Hence, once a block is deemed to have a high probability of producing write failure, it is retired to prevent repeated writes and associated overheads. Data dependent sparing introduces a *failure frequency threshold* to determine when a block is retired. Overall, Data Dependent Sparing's flow of execution is depicted in Fig. 37.

Data dependent sparing is expected to significantly prolong the lifetime of a memory devices. By continuously involving better-than-bad blocks in data writes and dynamically allocating spares

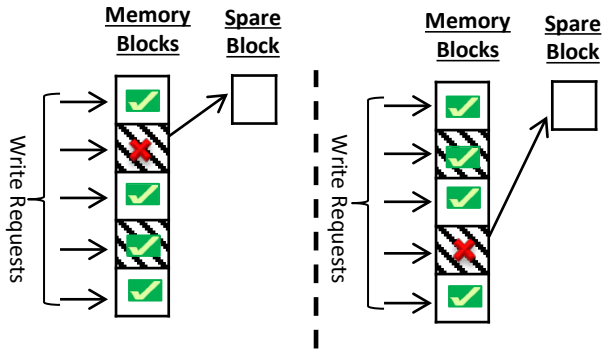

Figure 36: Data-dependent Sparing. Shaded cells represent defective blocks.

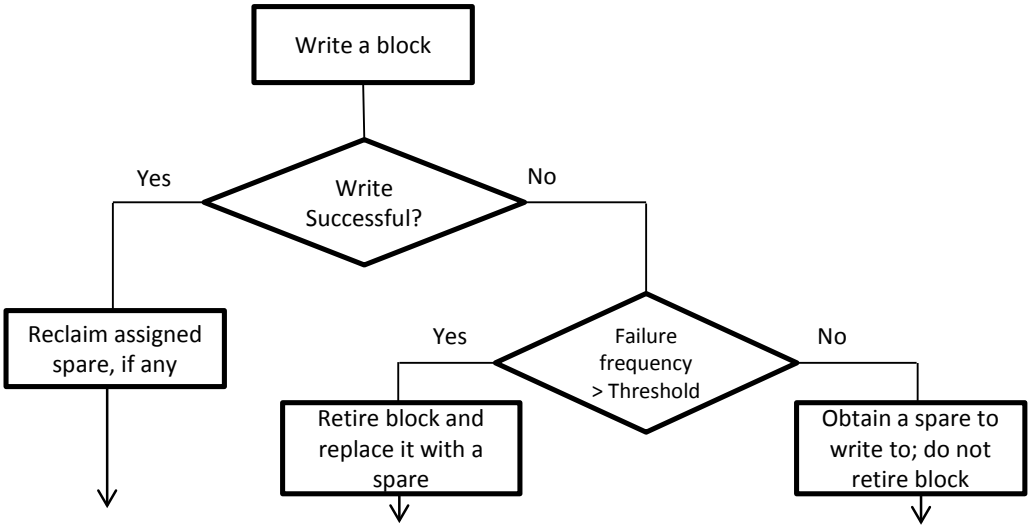

Figure 37: Flow of execution in data dependent sparing.

to temporarily accommodate writes that failed, data dependent sparing utilizes and manages spares more efficiently than conventional bad block management.

5.2.2 Design Trade-Offs

There are several interesting design decisions to be made when realizing data dependent sparing. The first has to do with *how spares are allocated upon a write failure*. Two strategies are feasible: (1) **Temp-sparing**: a healthy spare block temporarily substitutes the failing better-than-bad block and completes the write request; and (2) **Role-exchange**: a spare block permanently replaces the failing block and the failing block is added to the pool of spares. Temp-sparing has the advantage of reducing the wear of spares because the spares will be written infrequently compared with regular blocks. This strategy maximizes the chances of finding a healthy spare quickly when needed. Role-exchange has the advantage of spreading writes across the entire capacity of the device including spares. Hence, regular and spare blocks will wear at the same rate. It may incur multiple writes on spares more frequently than temp-sparing.

The second design choice is about how to map a logical block (that failed) to a new physical block. If the temp-sparing strategy is applied, a small table could be implemented to store pointers to spare blocks. The number of entries in the table is equal to the number of provided spares. If the role-exchange strategy is applied, then this requires address remapping so that read and write accesses are mapped to the new block. These two design choices are similar to Micron's *Skip Block Method* and *Reserve Block Method* [56].

Finally, we need a mechanism to determine when a better-than-bad block retires. This mechanism kicks in when a block write fails and is queried about the block's history of failures. If the block is failing more frequently than a *failure frequency threshold*, it is retired. A straightforward strategy to track past failure history is to associate a counter with each (better-than-bad) block that records the number of failures. In another strategy, one could employ a global data structure that approximates individual counters, like a counting Bloom filter. If the number of better-than-bad blocks is expected to be small in a device during its life (e.g., the device steers writes to known better-than-bad blocks to exhaust them first), this strategy could result in smaller bookkeeping space overheads than the first strategy.

5.2.3 Overheads

In data dependent sparing, a single block write operation may incur a series of writes before completion due to error occurrences, adding to the performance cost of a write. However, this cost is expected to be very small because erroneous writes are rare. If the temp-sparing strategy is applied, only one extra write is expected as this strategy preserves the health of spares. If the role-exchange strategy is applied, multiple writes could occur as the spare wears at the same level of other blocks. However, multiple writes will occur with a low probability. For example, consider a failure frequency threshold of 1/100. The probability of a write operation failing twice in a row for a given data pattern is at most 1/10,000.

The second source of overheads in data dependent sparing is the data structure to bookkeep history of errors. If data dependent sparing introduces a one-byte counter per 4KB block, this overhead corresponds to 1/4,096 = 0.24%. The overhead can be made smaller if we allocate counters on demand (i.e., no counter for a healthy block) or use an approximation data structure like Bloom filter.

Lastly, a major storage capacity overhead—of up to 20%—comes from over-provisioned capacity (the amount of spares). Spares are consumed as bad blocks occur, and hence, a storage device must provision sufficient spares to guarantee a target lifetime. Because data dependent sparing increases the write volume each storage block successfully absorbs, it effectively delays the wearing of available blocks and increases a storage device's lifetime. In turn, compared with static sparing, data dependent sparing reduces the overhead due to over-provisioning given a target lifetime.

5.3 EVALUATION

This section evaluates the performance of data dependent sparing, focusing on the relative advantage of data dependent sparing compared with static sparing in terms of lifetime improvement and reduction in required spare over-provisioning given a lifetime target.

Monte Carlo simulations were resorted to in the evaluation. Since detailed simulation of a

large storage capacity is impractical, 2,000 storage blocks of 4KB were simulated and results were driven statistically. Cells in the storage blocks have a write endurance following a Gaussian distribution with a mean of 10^8 and a standard deviation of 25×10^6. Overall, our methodology is similar to the one in [73].

It is assumed that an efficient wear levelling scheme distributes writes evenly across the available storage blocks. Each storage block is protected by a BCH code built capable of correcting t errors (BCH-t), where t is a configurable parameter is chosen. The number of faults within a block is kept track of. Once the number of faults gets above the BCH capability, the success of a write operation is determined based on the probability of failure as presented in Fig. 35.

Because the primary concern in this thesis is hard faults, it is assumed that the BCH capability is dedicated to masking solely hard faults. In case transient faults becomes an issue in the future, one must provision separate BCH capability, which is not considered in this work.

The write failure probability of each simulated block is tracked to determine when the block is retired. In addition, a number of spare blocks is assigned for each experimental setup.

5.3.1 Lifetime Improvement

At first, the lifetime improvement with data dependent sparing is considered. The modelled device uses BCH-20 and has 20% over-provisioning managed in accordance to the temp sparing scheme. The write failure frequency threshold is 10%, i.e., a block is retired if the probability of write failure on it reaches 10%. Fig. 38 plots the collected results: the percentage of surviving blocks in a storage device (Y axis) as a function of successful writes per block (X axis).

The result shows that when the number of surviving blocks with data dependent sparing is 100%, only 22% of the blocks survived with static sparing. In other words, data dependent sparing offers 78% point more physical storage capacity than static sparing before bad blocks happen. In terms of storage device lifetime—time until the first bad block occurs after consuming all spares—data dependent sparing's advantage is clear; it increases the lifetime by 18.1% compared with static sparing.

The two curves in the plot have a noticeably different shape. Static sparing starts to lose storage blocks sooner and keeps losing more and more blocks as they become faulty. On the

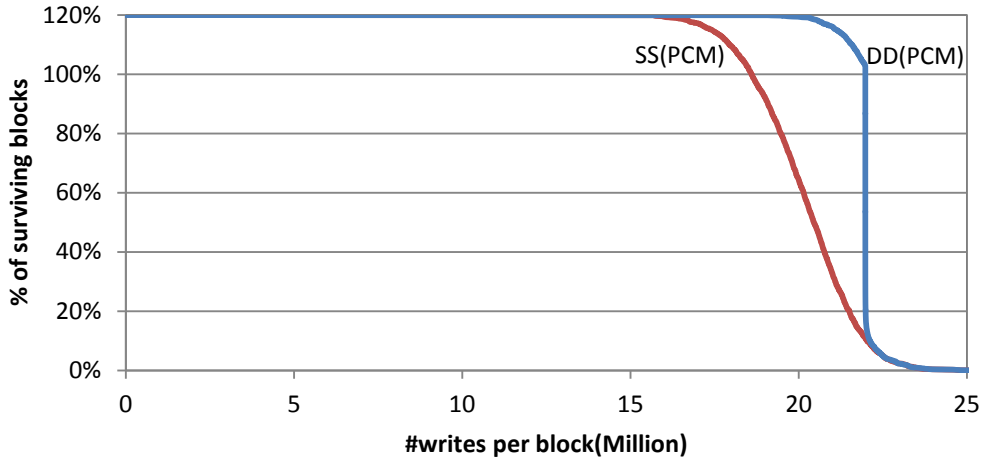

Figure 38: Lifetime of PCM blocks with BCH-20 and 10% failure frequency threshold. "DD" denotes data dependent sparing and "SS" static sparing.

other hand, data dependent sparing sheds blocks much later but loses many blocks near the end of usable lifetime. Data dependent sparing salvages otherwise bad blocks and effectively extracts more lifetime from better-than-bad blocks.

5.3.2 Sensitivity to Over-Provisioning

To study the sensitivity of data dependent sparing to the amount of over-provisioned capacity, a metric dubbed *lifetime increase* is defined and used. It is the difference between lifetime with data dependent sparing and lifetime with static sparing, divided by lifetime with static sparing. In essence, this metric expresses the relative lifetime advantage with data dependent sparing.

Four different over-provisioned capacities of 1%, 5%, 10% and 20% for data dependent sparing are examined, while keeping 20% over-provisioning for static sparing. It is assumed that the block is protected by BCH-20 along with a block failure frequency threshold of 10%.

Fig. 39 shows the result. Data dependent sparing is shown to outperform static sparing at all over-provisioned capacity levels examined. Data dependent sparing beats static sparing (with 20% over-provisioning) with only 1% over-provisioning and increases the lifetime by up to 4.5%. At 20% over-provisioning (i.e., same spare capacity for both schemes), lifetime increase reaches

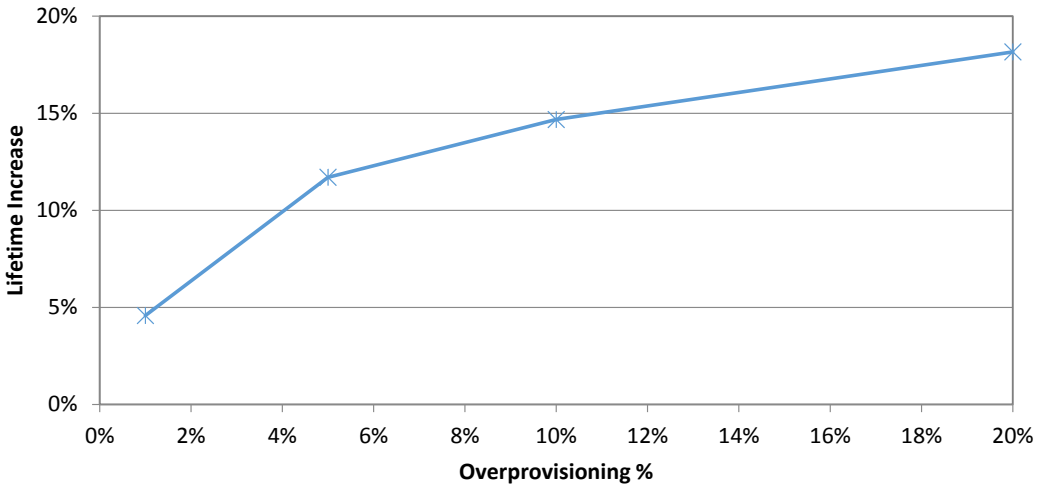

Figure 39: Lifetime increase achieved by data dependent sparing at various levels of over-provisioning compared with static sparing with 20% over-provisioning.

18.1%. The result proves the observation that errors are typically fewer than faults in a block is extremely valuable; indeed, data dependent sparing significantly increases the amount of data written to a block. This explains why data dependent sparing achieves a target lifetime with a smaller over-provisioned capacity than static sparing.

5.3.3 Sensitivity to BCH Capability

This section studies the effect of changing the capability of the error correction mechanism. It is experimented with BCH-5, BCH-10, BCH-15 and BCH-20 to revel the effect. 20% over-provisioning is assumed along with a block failure frequency threshold of 10%. The evaluation metric is lifetime increase.

Fig. 40 plots the result. It is shown that lifetime increase with data dependent sparing is larger when a weaker BCH code is used. This result may look surprising at a first glance. However, with a weaker BCH code static sparing retires faulty blocks and starts to consume spares sooner. By comparison, data dependent sparing continues using better-than-bad blocks and saves wearing of spares, which translates into a significant gain in lifetime. Our result shows that data dependent sparing is robust in improving the lifetime of a device across the capability of an error correction

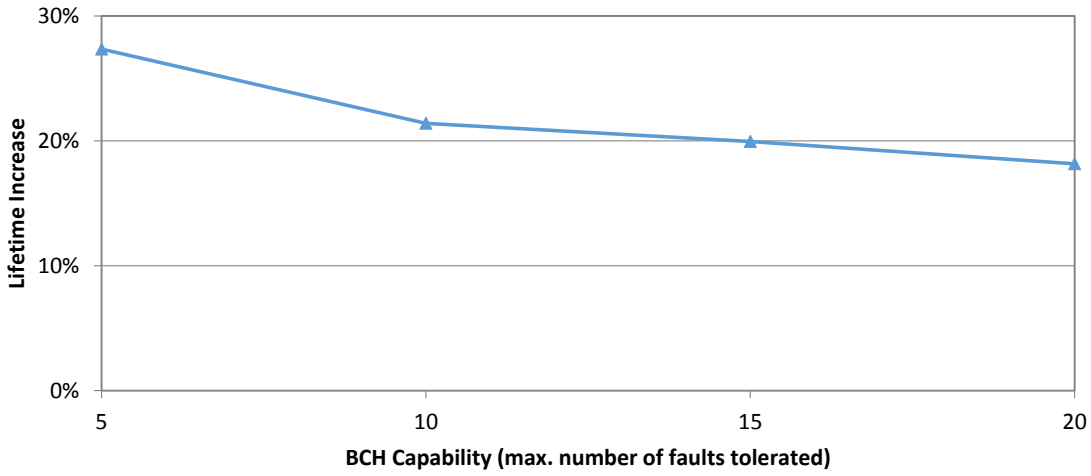

Figure 40: Lifetime increase achieved by data dependent sparing relative to static sparing for various BCH code capabilities.

mechanism used.

5.3.4 The Effect of Fail Frequency Threshold

This section studies the influence of the failure threshold upon which a block is retired. The memory blocks are protected with BCH-20 and 20% spares are provided. Two thresholds are compared—5% and 10%.

Fig. 41 shows the result. We find that a smaller threshold value leads to a shorter lifetime; this is expected because data dependent sparing discards a better-than-bad block more quickly with a smaller threshold value. Still, data dependent sparing under both threshold values achieves substantial gain in storage block lifetime. Our result confirms an interesting design trade-off—fail frequency threshold affects the lifetime of the storage device and management overheads.

5.3.5 Sparing Overhead Reduction

Finally, this section revisits the question of how much over-provisioning is needed for data dependent sparing to achieve a target lifetime, compared with static sparing. Static sparing is assigned

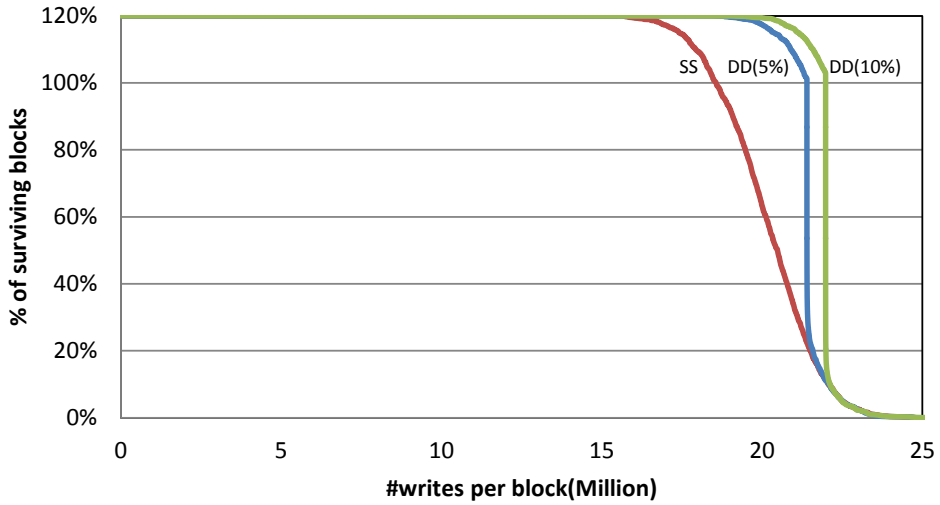

Figure 41: Lifetime of PCM blocks under two failure frequency threshold values: 5% vs. 10%. "DD" denotes data dependent sparing and "SS" static sparing.

with 20% and 10% over-provisioning and the over-provisioning for data dependent sparing is obtained. A BCH-20 code is used along with a failure frequency threshold of 10%.

Table 5.3.5 highlights the capability of data dependent sparing in reducing the amount of over-provisioning; it requires as low as 0.4% over-provisioning to achieve the same lifetime as static sparing with 20% over-provisioning and only 0.1% over-provisioning to match the lifetime of static sparing with 10% over-provisioning. Clearly, data dependent sparing has the good potential to increase the value of a storage device by achieving a target lifetime with reduced cost.

The collected results portrays the flexibility that the data dependent sparing scheme offers to

	DD Over-provisioning
20% Sparing(SS)	0.4%
10% Sparing(SS)	0.1%

Table 3: Required over-provisioning for data dependent sparing to match static sparing lifetime.

device manufactures. It helps them reduced the amount of over-provisioning while preserving the service contract agreement in terms of lifetime; thus decreasing the manufacturing cost. Otherwise, the reduced amount of spare over-provisioning could be redirect to the user space. Thus, providing more storage capacity with the same cost.

5.4 SUMMARY

Data Dependent Sparing is a new bad block management technique in the context of PCM's stuck at-fault model. The proposed technique delays the retirement of bad blocks as their block write failure probability is low due to the data dependent nature of errors. When a write to a block fails, a spare is assigned temporally and later writes are attempted on the original block. Overall,this chapter makes the following key contributions:

1. Introduces a new block classification: good, better-than-bad and bad
2. Presents Data Depending Sparing as a new physical block sparing technique that engages both good and better-than-bad blocks in the write operation.
3. Shows that Data Depending Sparing can either increase the lifetime by up to 18% over the conventional approach with equivalent spare over-provisioning or achieve a target lifetime with as few as 1% of the spare over-provisioning assigned to the conventional approach.

While deploying Data Dependent Sparing for PCM is essential for a better management of defective blocks, the scheme does not apply to PCM only. In fact, Data Dependent Sparing is relevant to other memory technologies that exhibit a stuck-at faults model. In essence, the proposed technique highlights the need of carefully investigating the write failure source as the first write failure is not indicative of the degree of defectiveness of a memory block.

6.0 EXTENSIONS TO MULTILEVEL CELLS

The difference in resistance between the RESET and the SET states in PCM is about 2 to 3 orders of magnitude. This characteristic allows PCM to store multiple level per storage cell (MLC), which achieves a higher density. Instead of considering only the RESET and SET states, MLC storage can be achieved through accurately programming cells to multiple intermediate resistance levels between the aforementioned two states [35, 60, 4, 61]. For example, storing 4 levels per cell (4LC) enables the storage of 2 bits per cell i.e. “00”, “01”, “10” and “11”.

To program MLC PCM, the widely adopted strategies are partial-RESET and partial-SET [6, 3, 57]. To program a cell to a target level, Partial-RESET puts the cell first into the SET state then a partial RESET state pulse is applied to program the cell to the intended level. Partial-SET follows the same approach, but puts first the cell in the RESET state. The two approaches offer a tradeoff between low write latency and write energy with partial-RESET being more energy efficient (putting a cell in the SET state is less energy demanding than the RESET state) and with partial-SET being more time efficient (putting a cell in the RESET state is faster than the SET state). To get the best of both worlds, a hybrid scheme that combines both strategies can be used [35]. The scheme picks one of two strategies to program a cell depending on how far the target level is from either the SET or the RESET state. Irrespective of the adopted approach, a program and verify [5, 57] approach is used on top. Since accurate programming within the level boundaries is essential for MLC, program and verify reads the programmed cell after every write pulse. If the cell is not at the target level yet, the cell undergoes a second programming round.

While achieving higher density is desirable, increasing the number of levels has a diminishing return. A major disadvantage of MLC is drift soft errors and high energy consumption [84, 34]. Actually, PCM cells are susceptible to resistance drift over time. This drift in resistance is not large enough to cause SLC cells to switch levels, but it affects MLC cells leading to a soft bit error

rate that cannot be practically tolerated by error correcting codes. To counter those disadvantages, storing as many as 3 levels per cell (3LC) achieves orders of magnitude lower cell soft error rate than 4LC, and higher level cells, making 3LC cells soft error rates comparable to SLC cells. By eliminating one or more memory levels, the distribution of the resistance range in 3LC over the remaining 3 levels becomes less tight; thus significantly reducing the soft error rate [84, 77]. Accordingly, this thesis considers 3LC due to their resistance to drift errors. If drift errors becomes an issue for 3LC or higher memory levels needs to be stored, then provisioning against drift errors through a dedicated error correcting code becomes a necessity.

This chapters explores the application of the proposed techniques at the pre-write fault avoidance, post-write fault tolerance and the post-failure recovery stages in the context of MLC storage cells. The modifications and the extensions that each technique requires to be applicable in context of MLC are highlighted.

6.1 SYMBOL SHIFTING

Both CAFO (Refer to Chapter 3) and Data Inversion (Refer to Chapter 4) relied on the concept of inversion to encode the write data. While a flip can still be applied in the context of MLC, it only offers the choice between two symbols for each cell: the un-inverted and the inverted symbol. MLC, however, offers the choice of $k-1$ symbols other than the stored symbol where k is the number of levels that a cell can store. To get to those $k-1$ symbols, this thesis presents *Symbol Shifting*. Essentially, Symbol Shifting shifts all the data symbols within a memory block by i levels such that i is less than the number of level that a cell can store which results in a new data symbol stored within each memory cell. Definition 3 formally states the concept of symbol shifting.

Definition 3. *Applying Symbol Shifting on a data block with k-level cells is a shift of every symbol in the block by i levels for* $1 \leq i \leq k-1$ *such that* $x' = (x+i)\ mod\ k$ *where* x' *is the new shifted symbol level and x is the old symbol level.*

It follows from Definition 3 that Symbol Shifting provides $k-1$ different mappings for a memory block of k-level cells. For Example, 3LC cells ($k=3$) can be shifted by either one or two

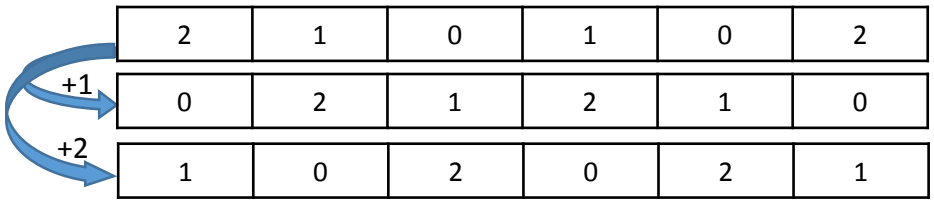

Figure 42: A shift of symbols in the context of 3LC PCM cells.

memory levels resulting in two data mappings as depicted in Figure 42.

6.2 MULTILEVEL CAFO

Applying CAFO (Refer to Chapter 3) in the context of MLC, M-CAFO, requires extending the cost model and encoding engine. The cost model is extended to capture all the possible transitions in levels that a cell can undergo. For 3LC, the cost model consists of 9 possible level transitions captured in Table 4. Setting the cost values depends on the technology being modelled and the adopted approach to program MLC cells.

CAFO's encoding consists of modelling a memory block in the 2D space and alternatively flipping rows and columns to reduce the cost of the write data. Instead of applying a data flip, M-CAFO uses the concept of Symbol Shifting. Thus each row or column can be written either un-shifted, or shifted.

As in CAFO, M-CAFO needs to determine the gain of either a row or a column and decide whether that row or column should be written un-shift or shifted. For 3LC, each row or column can be written either un-shifted, shifted by one level ($shift_1$), or shifted by two levels ($shift_2$). If neither of the shifted patterns yields in positive gain, then the data is written un-shifted. Otherwise, the pattern that yields in the maximum positive gain is chosen. In order to be able to calculate the gain, the cost of symbol transitions (refer to Table 4) has to determined. This cost depends on the programming model of the 3LC cells. This thesis adopts a hybrid programming model that is

Cost	Transition
a	0→1
b	0→2
c	0→0
d	1→0
e	1→1
f	1→2
g	2→0
h	2→1
i	2→2

Table 4: M-CAFO cost model for 3LC PCM cells.

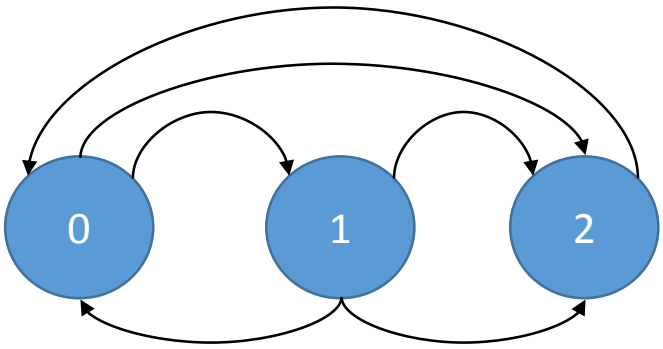

Figure 43: 3LC programming model.

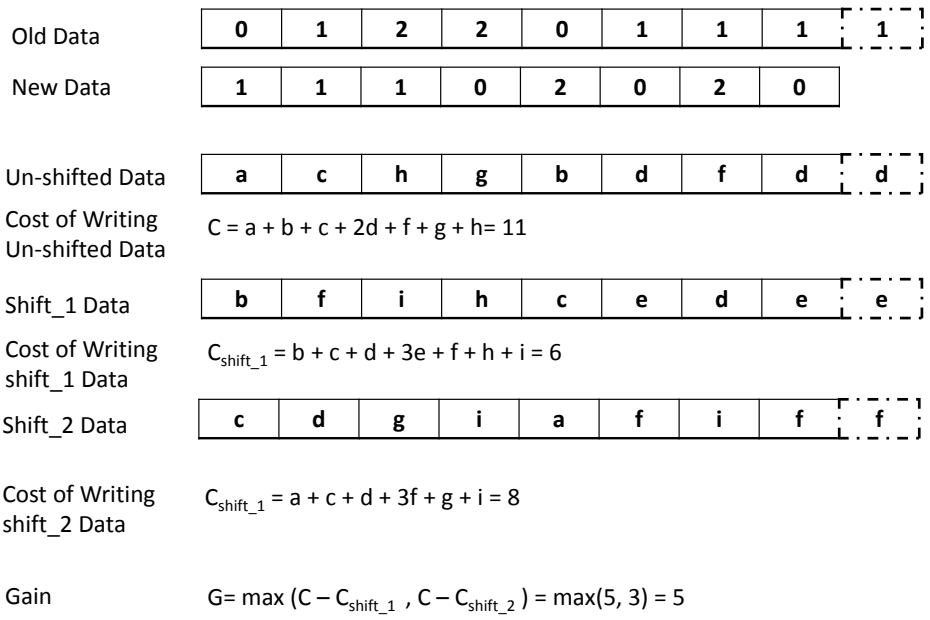

Figure 44: Gain calculation in the context of 3LC PCM cells.

depicted in Figure 43. Assuming ternary symbols of “0”, “1” and “2”, the adopted programming model can program cells upward from their current stored symbol without having to go through the RESET state e.g. a cell storing symbol “1” can be programmed to store symbol “2” without having to go through the RESET state. On the other hand, programming cell downward from their current stored symbol necessitates going through the RESET state i.e. programming a cell that stores symbol “2” to store symbol “1” has to go through the RESET state (“0” symbol state).

Figure 44 shows an example of gain calculation in the context of 3LC, where the cost of an un-matching cell than has to go through the RESET state is double the cost of an un-matching cell that does not have to go through the RESET state in accordance to the programming model in Figure 43. At first, the cost of writing the data un-shifted is computed (11). Then, the costs of writing data shifted by one memory level (6) and by two memory levels (8) are computed. At last, the gain is calculated and shows that shifting by one memory level yields a higher gain i.e. writing the data shifted by one level incurs lower endurance cost.

Besides the above mentioned changes, M-CAFO does not require additional modifications to

the encoding process. Once the gain of rows is determined, those rows with positive gain are shifted. Subsequently, the gain of the columns is determined and the columns with positive gain are shifted. The number of levels that the memory cells within a memory block got shifted by is stored within the additional auxiliary set i.e. “0” implies no shifts, “1” implies a shift by one memory level and “2” implies a shift by two memory levels. This process is repeated until no row or column show a zero or negative gain.

Section 3.1.4 in Chapter 3 introduced the possibility of achieving further cost reduction even after all the rows and columns show a zero or negative gain through applying an optimization to the encoding. The optimization consists of simultaneously flipping one row with multiple column, or vice versa. In theory, M-CAFO can apply the same optimization. In other words, multiple rows and columns can be shifted simultaneously. The only difference is that for the each of the chosen row and columns, the optimization engine has to note the shifted level, i.e. shift by either one level or two levels, which could complicate the optimization process. Nevertheless, the optimization is a design option and can be dropped to avoid complexity.

6.2.1 Evaluation

To evaluate M-CAFO, Monte Carlo simulations with syntactic data are carried and the average cost reduction is computed. M-CAFO results are compared against M-CFNW. The latter is a cost aware variation of Flip-N-Write 3 that adopts the symbol shift concepts to apply the technique in the context of MLC– FNW can be looked as a 1D CAFO. As for Flip-Min 3, the generation of cosets are based on the Reed-Muller code that is specific to cells that store binary symbols. Investigating the concept of Flip-Min with multi-symbols codes to generate cosets is beyond the scope of this thesis and is not considered. Figure 45 shows the write cost reduction achieved by M-CAFO over M-CFNW relative to differential write. The adopted costs are set are in accordance with the programming model in Figure 43. That is, each cell transition that has to go through the RESET resistance (STATE ”0”) has a higher endurance cost than the transition that does not have to go through the RESET resistance state. Accordingly, the cost of the RESET state is varied. Furthermore, the simulation is carried with various block sizes and different space overheads. It is worth mentioning that the experiments do not consider applying the encoding optimization for

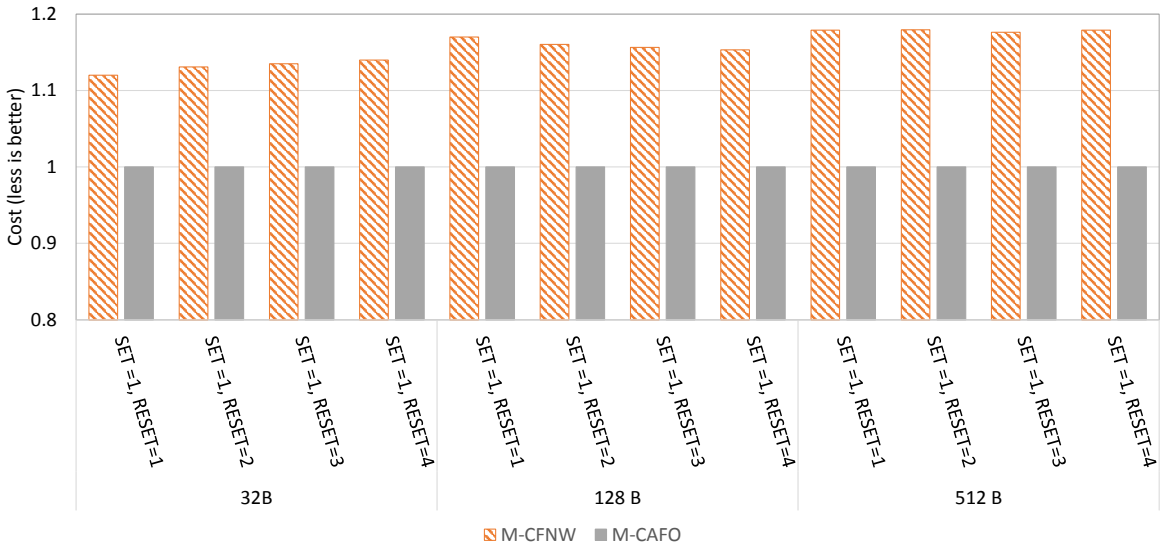

Figure 45: Cost reduction of M-CAFO over M-CFNW in the context of 3LC PCM cells.

M-CAFO.

The results show that M-CAFO reduces the cost over M-CFNW by up to 18% in the context of 3LC. Accordingly, CAFO outperforms FNW irrespective of the number of levels that are to be stored within the storage cell. One thing to note is that both M-CAFO and M-CFNW have the same computational overhead in the 3LC context as in the SLC context. Although in the 3LC context two patterns, shift_1 and shift_2, for the gain calculation have to be considered, the gain calculation for each pattern can be done in parallel. Therefore, M-CAFO and M-CFNW do not introduce an additional computational overhead in the context of MLC.

6.3 INCREASING ERROR CORRECTING CAPABILITIES IN MLC THROUGH DATA SHIFTING

Data inversion was presented in Chapter 4 as a technique to increase the number of faults that an error correcting code can cover within a protected block before turning defective. Integrated

Protection and Un-integrated Protection were introduced as two variations of data inversion that can tolerate a number of faults that is up to double the nominal capability of the deployed ECC in the context of SLC. This chapter presents the application of data inversion in the context of MLC.

While data inversion in the context of SLC consists of inverting the data post to a write failure, symbol shifting offers the possibility of applying $k-1$ shifts on a k-level cell (Refer to Figure 42) in the context of MLC cells. Hence, MLC requires moving from the concept of data inversion to *data shifting* using the concept of symbol shifting [49].

6.3.1 Theoretical Foundation of Data Shifting

This section studies the effect of shifting the data on k-level memory block containing stuck-at faults. Particularly, the impact on the classification of stuck-at faults between SA-W and SA-R after a shift is highlighted.

Proposition 1. *Given a memory block with ω SA-W cells, applying Symbol Shifting can make at least ω/(k-1) cells to become SA-R, where k is the number of levels that a memory cell can store.*

Proof: For k levels, a symbol within a memory cell can be shifted by i levels for $1 \leq i \leq k-1$. Define a cell to be SA-W$_l$, for $1 \leq l \leq k-1$, if the symbol that is to be written on that cell needs to shifted l times to match the stuck-at value. let ω_l be the number of SA-W$_l$ cells. Accordingly, ω_1 + . . . + ω_{k-1} = ω. For $l = i$, shifting the data pattern by i levels would make all the SA-W$_l$ cells to become SA-R. In the worst case, ω_1 = ω_2 = ... = ω_{k-1}. Thus, applying a shift can bring at least $\omega/(k-1)$ cells to become SA-R.

It follows from Proposition 1 that shifting the data can make $\omega/(k-1)$ cells to turn SA-R in the worst case. For example, consider a 3LC block with 6 stuck-at cells that all end up SA-W after a write operation i.e. $\omega = 6$. Let us assume that three of SA-W cells are stuck at a symbol value that is one level away from the intended symbol value to be written and the remaining 3 SA-W cells are stuck at a symbol value that is two levels away from the intended symbol value to be written. Accordingly, shifting the data symbols by either one or two levels can make at most 3 SA-W to turn SA-R.

Proposition 2. *Given a memory block with ρ SA-R cells, applying Symbol Shifting turns all the ρ SA-R cells to SA-W.*

Proof: For k levels, every symbol within a memory block can be shifted by i levels for $1 \leq i \leq k-1$. Hence, it is obvious that all the ρ SA-R cells will be shifted to a level that does not match with the symbol to be written on those cells (shifting the cells by k levels would bring the cells to the same current level). Accordingly, a shift of the data pattern turns all the SA-R cells SA-W.

Propositions 1 and 2 show the effect of applying a Symbol Shifting on the classification of the faulty cells between SA-R and SA-W. Clearly, deciding to apply a shift on the data pattern to be written depends on the number of SA-R and SA-W within a block as shown next.

As a memory block gets written, faults starts to occur due to the limited endurance of the cells. For each write request, the accrued faults are classified as either SA-W or SA-R depending of the data pattern to be written. In fact, there exists an upper bound on the number of cells that can be SA-W when the data can be shifted.

Theorem 3. *Given a memory block with ϕ faulty cells, the number of faulty cells that can be SA-W is at most $\omega = \phi(k-1)/k$ when a shift on the data symbols can be applied, where k is the number of levels that a memory cell can store.*

Proof: Given ϕ faults within a memory block, writing on such a block would result in $\omega + \rho = \phi$ cells where ω is the number of SA-W cells and ρ is the number of a SA-R cells. A Symbol Shifting should be applied if it ends up in ω' SA-W cells such that $\omega' < \omega$. In accordance to Propositions 1 and 2, we know that applying a shift on the data renders all the ρ SA-R cells to turn SA-W and $\omega/(k-1)$ of the SA-W cells to become SA-R . Thus, $\omega' = \omega - \omega/(k-1) + \rho$. Accordingly, there is need to determine the value of ρ that would make Symbol Shifting capable of reducing the initial number of SA-W cells i.e. $\omega' < \omega$. Solving $\omega - \omega/(k-1) + \rho < \omega$ for ρ results in $\rho < \omega/(k-1)$. Thus, a Symbol Shifting is to be applied only if $\rho < \omega/(k-1)$. Consequently, we know that if $\rho \geq \omega/(k-1)$ then a shift on the data is not effective in reducing the initial number of SA-W. Accordingly, for $\rho \geq \omega/(k-1)$ the maximum number of SA-W cells occurs when ρ is at is minimum possible value i.e. $\rho = \omega/(k-1)$. Substituting ρ by $\rho = \omega/(k-1)$ results in $\phi = \omega + \rho = \omega + \omega/(k-1)$. Solving the latter equation for ω ends up in $\omega = \phi(k-1)/k$.

For example, consider a 3LC memory block ($k = 3$) where $\phi = 6$. The maximum number of cells that can be SA-W if a shift on the data is applied is 4. Hence, Theorem 3 sets an upper bound on the number of stuck-at cells that can be SA-W when data shifting can be applied.

6.3.2 Data Shifting effect on Block Defectiveness

This section studies the effect of complementing an error correcting coding with Data Shifting in the context of MLC in the two settings of Integrated and Un-integrated protection that were presented in Chapter 4. Both settings augmented the protected block with an extra polarity bit to flag the inversion step. In the context of MLC, the protected block is augmented as well with an extra cell that serves as a shift status cell (SSC) to mark the number of memory levels that a memory block got shifted by.

6.3.2.1 Integrated Protection

The application of Integrated Protection in the context of MLC is similar to that of SLC. When a write request is submitted, the data cells are augmented with the SSC set to 0. Subsequently, the error correcting code computes the auxiliary information to protect against errors in the original data cells and the SSC. Next, the codeword (data cells + SSCs + auxiliary cells) is physically written. If the codeword manifests a number of SA-W cells larger than the error correcting capability of the code, then Symbol Shifting kicks in. The codeword is recomputed with shifted data symbols and the SSC is set to reflect the number of levels the data symbols were shifted by.

Applying a shift has the potential effect of increasing the number of stuck-at faults that can be tolerated within the data cells, while preserving the non-defectiveness of the memory block as described in the following theorem.

Theorem 4. *Given a memory/storage block with k-level cells protected by an error correcting code that can correct up to t-symbol errors, applying Symbol Shifting with integrated protection extends the correction capability of the code to $\phi_d + \phi_a$ faults such that $(\phi_d(k-1)/k + \phi_a) = t$, where ϕ_d is the number of stuck-at faults in the data cells and ϕ_a is the number of stuck-at faults in the auxiliary cells. That is, the block is defective only if $(\phi_d(k-1)/k + \phi_a) > t$.*

Proof: According to Theorem 3, the maximum number of SA-W cells, ω, that the ϕ_d faults can manifest is $\omega = (k\phi_d - \phi_d)/k$, where k is the number of levels that can be stored within a cell. When it comes to the ϕ_a faults in the auxiliary cells, ϕ_a errors can be manifested in the worst case as recomputing the auxiliary information could make all the stuck-at cells in the auxiliary cells to

turn SA-W. Hence, a memory/storage block becomes defective only if $(\phi_d(k-1)/k + \phi_a) > t$.

Corollary 1. *For 3LC (k=3), a block turns defective if* $(2\phi_d/3 + \phi_a) > t$.

Similarly to the SLC context, it follows from Theorem 4 and Corollary 1 that Data Shifting makes the defectiveness of memory blocks correlated with the distribution of the stuck-at faults between the data and the auxiliary cells within a block. It is highly likely that the distribution of faults allows increasing the number of faults within a block before it turns defective. Hence, Symbol Shifting can be looked at as increasing the capability of the error correcting code. Consider a 3LC memory block with $\phi_d = 6$ and $\phi_a = 2$ protected with an error correcting code of capability $t = 6$. Such a block is not defective as $(2\phi_d/3 + \phi_a) = 4 + 2 = 6 \leq 6$.

6.3.2.2 Un-integrated Protection

The Un-integrated protection setting separates the SSC from the codeword. This setting guarantees doubling the number of faults within the context of SLC. Theorem 5 states the effect on augmenting an error correcting code with Data Shifting on an MLC block.

Theorem 5. *Given memory/storage block with k-level cells protected by an error correcting code that can cover up to t-symbol errors, applying Symbol Shifting with un-integrated protection turns the block defective only after* $\phi = k(t+1)/(k-1)$ *stuck-at faults are accrued, where* ϕ *is the number of faults within the block.*

Proof: Having ϕ stuck at faults within a memory block, we know from Theorem 3 that the maximum number of SA-W cells that the ϕ faults can manifest is $\omega = \phi(k-1)/k$. For a write failure to occur, ω must be greater than t i.e. $t+1$. Accordingly, we need to determine the value of ϕ that can make $\omega = t+1$. This value can be determined though solving the following equation $\omega = t+1 = \phi(k-1)/k$ for ϕ which ends up in $\phi = k(t+1)/(k-1)$.

Corollary 2. *For 3LC (k=3), a block turns defective if* $\phi = 1.5t + 1$.

In accordance to the SLC findings, It follows from Theorem 5 that separating the SSC out of the codeword allows an error correcting code to be capable of extending the non-defectiveness of a block irrespective of the distribution of the faults within the block. For example, consider a 3LC memory block with $\phi = 9$ stuck-at faults protected by an error correcting code of capability 6.

This block is not defective irrespective of the distribution of the stuck-at faults within the block as $\phi < 1.5t + 1 = 9 < 10$ in accordance to Corollary 2

6.3.3 Execution Flow

The flow of execution of data shifting does not require a substantial modification to that of data inversion in the context of SLC. In the event of a write failure, an additional write operation is attempted with the codeword symbols shifted and the Symbol Shifting Cell (SSC) set to a value that reflect the number of levels that the symbols got shifted by. For a 3-level memory cells, the symbols in a block can be shifted by i level for $1 \leq i \leq 2$. Those shifts can be executed in parallel; Thus incurring the same delay as the flip operation in the SLC context. Subsequently, the SSC is set to the value of i that will yield the least number of SA-W as it is most likely to make the second write attempt successful. At read time, the codeword is read un-shifted if the value of the SSC is different than 0. It is worth mentioning that switching to the MLC context requires the deployment of multi-symbol error correcting codes such as LDPC [25]. To clarify the flow of execution, Figure 46 gives an example of a (6,11) LDPC code for ternary cells capable of protecting against two symbol errors complemented with Data Shifting with un-integrated protection where ternary cells store symbols of 0, 1 and 2. The example shows that the first write attempt fails as a consequence of 3 SA-W cells. Subsequently, the codeword is encoded through shifting the symbols by one level up, i.e. 0→1, 1→2 and 2→0, as it leads to the least number of SA-W cells. Afterwards, a second write is attempted with shifted codeword and SSC set to 1. The second write is successful as it ends up with one SA-W which is within the reach of the deployed LDPC code. At read time, firstly the codeword is retrieved un-shifted as the value of the SSC is 1 i.e. shifting the level of the cells by one level down i.e. 0→2, 1→0 and 2→1. Secondly, the codeword is decoded to retrieve the data cells as where intended to be read initially.

6.3.4 Dealing with Drift Errors

At the current stage of PCM technology, SLC and 3LC memory devices are not susceptible to drift errors. In fact, the error rate of drift errors becomes problematic and requires attention for 4LC cells and higher. Nevertheless, the operation of data shifting in an environment where drift errors

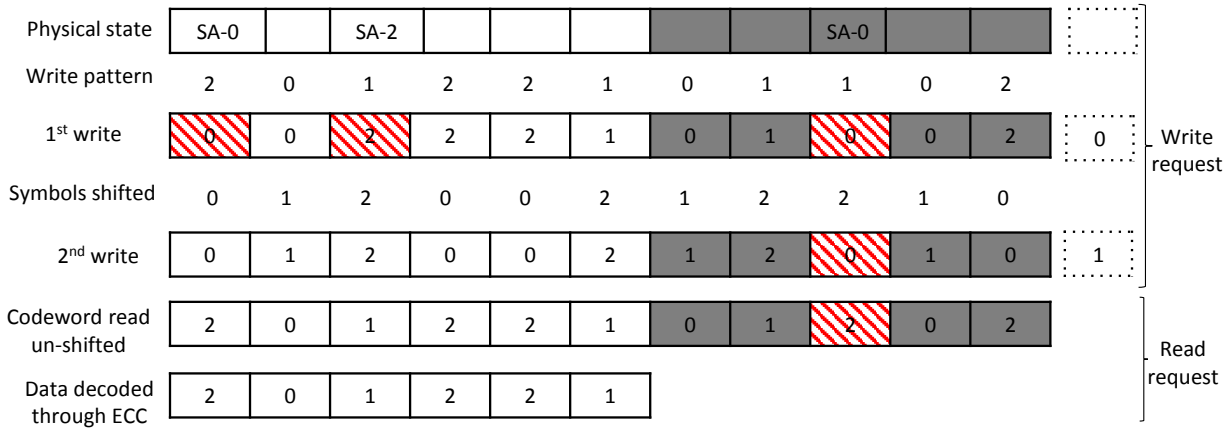

Figure 46: An example of and (6, 11) LPDC code for ternary cells of capability 2 complemented by data shifting with un-integrated protection. Dotted cells represent the SSC, grey cells represent the auxiliary cells and red patterned cells represent symbol errors.

can occur is investigated in case they become an issue for SLC and 3LC in the future. This chapter presents *Reserved Capability* and *Composite Capability* as two approaches that can deal with drift errors without intervening with the flow of execution of Symbol Shifting.

Before introducing the details of the proposed approaches, it is worth noting an important aspect of drift errors which is their time dependent nature. In fact, the probability of a cell drifting is a function of the time that has elapsed since the cell was programmed. This probability could become significant after several minutes of programming time [84, 77]. Hence, instantaneous reads after a write operations are unlikely to suffer from drift errors.

6.3.4.1 Reserved Capability

The Reserved Capability (RC) approach consists of provisioning against drift errors through the error correcting code (ECC) deployed to recover from stuck-at faults. Basically, a fraction of the capability of the ECC is reserved to cover drift errors. Given an ECC of capability t, a sub-capability c, such that $c < t$, is provisioned to cover drift errors. Accordingly, the remaining

capability, $t - c$, is dedicated to cover stuck-at faults. It is worth noting that setting the value of c depends of the bit error rate of drift errors.

Provisioning against drift errors does not change the flow of execution of Symbol Shifting, but can be applied only to the Integrated Protection approach as the ECC does not cover the SSC with Un-integrated protection. Theorem 4 defines the criterion of block defectiveness, when Symbol Shifting with Integrated Protection is used, as a function of the capability t of the ECC, where t is assumed to be dedicated to cover only stuck-at faults. Accordingly, it is enough to substitute t by $t - c$ in the theorem, where c is the capability reserved to cover drift error, in order to determine the new criterion of block defectiveness with RC. For example, an SLC block with ϕ_d stuck-at cell in the data bits and ϕ_a stuck-at cell in the auxiliary bits is defective if $(\phi_d/2 + \phi_a) > t$. With RC, such a block is defective if $(\phi_d/2 + \phi_a) > (t - c)$. For example, for $t = 6$, $c = 1$, $\phi_d = 6$ and $\phi_a = 1$, the block is not defective as $(6/2 + 2) = (6 - 1) = 5$

6.3.4.2 Composite Capability

In order for Data Shifting to achieve its premise, the data symbols must be read as they were written i.e. shifted or un-shifted. When a drift error affects a block that was written in accordance to Data Shifting, the data decoded at read time will be different than the intended data to be written. While RC can deal with drift errors only for Integrated Protection, the *Composite Capability* (CC) approach that can deal with drift errors in both setting of Integrated and Un-integrated protections.

CC consists of two separate ECCs: (1) ECC_{hard} and (2) ECC_{drift}. At first, the data is written encoded with ECC_{hard} augmented with Data Shifting (refer to Figures 22 and 23). Next, the codeword of ECC_{hard} is encoded with ECC_{drift} i.e. the codeword of ECC_{hard} represents the information bits of ECC_{drift}. When the data is to be read at a later stage, the codeword of ECC_{hard} is first retrieved through decoding using ECC_{drift}. At this stage, any drift error that occurred would be corrected. Once the codeword of ECC_{hard} is retrieved correctly, the decoding process proceeds in accordance to the mechanism of Symbol Shifting (refer to Figures 22 and 23).

Fig. 47 shows a hypothetical example of CC encoding and decoding. The data is first encoded with ECC_{hard} complemented with Symbol Shifting. Then, the codeword of ECC_{hard} is written. It results in one SA-W cell that is assumed to be within the capability of ECC_{hard}. Hence, there is no need for a second write with the data shifted. Next, the codeword of ECC_{drift} is computed

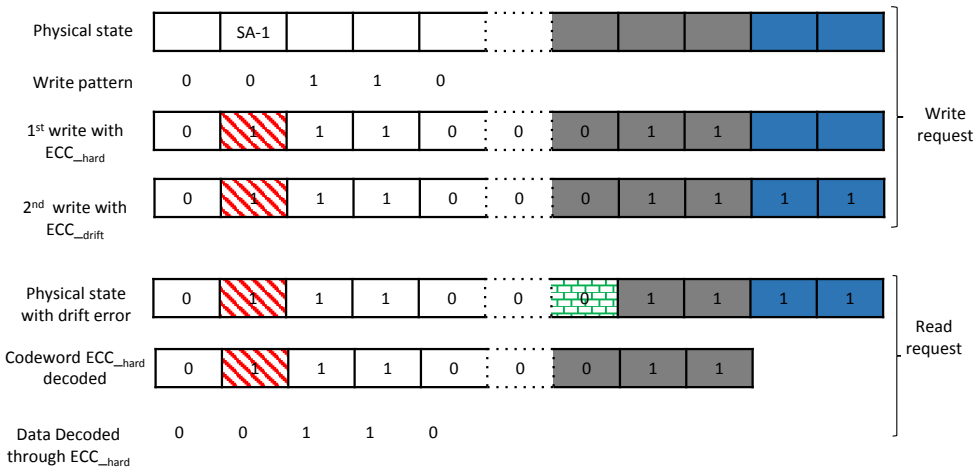

Figure 47: An example of encoding/decoding with Composite Capability. Dotted Cell represents the SSC. Grey cells represent ECC_{hard} auxiliary bits. Blue cell represents ECC_{drift} auxiliary bits. Red diagonal patterned cells represent stuck-at cell errors. Green brick patterned cells represent drift errors.

covering the codeword of ECC_{hard} which concludes the write operation. Before reading the data, let us assume that one cell drifted (green brick pattern). At read time, the codeword of ECC_{hard} is first retrieved through decoding with ECC_{drift} which is capable of tolerating the drift error. Next the data is retrieved through decoding with ECC_{hard} tolerating the SA-W cell. Thus, the data is retrieved correctly.

6.3.5 Data Shifting Evaluation

To evaluate data shifting in the context of 3LC, the same experiment setup as in Section 4.1.7.1 in Chapter 4 is followed. Instead of a BCH code, an LDPC-6 code is used to protect against symbol errors in 3LC main memory blocks. Since programming MLC cells could require multiple rounds, the endurance of MLC memory is notably lower than that of SLC. To capture this phenomenon, 3LC cells are assigned lifetime using a Gaussian distribution with a mean of 10^5 and a standard deviation of 25×10^3 for 3LC [4]. Figure 48 notes the lifetime improvement, where the Y axis

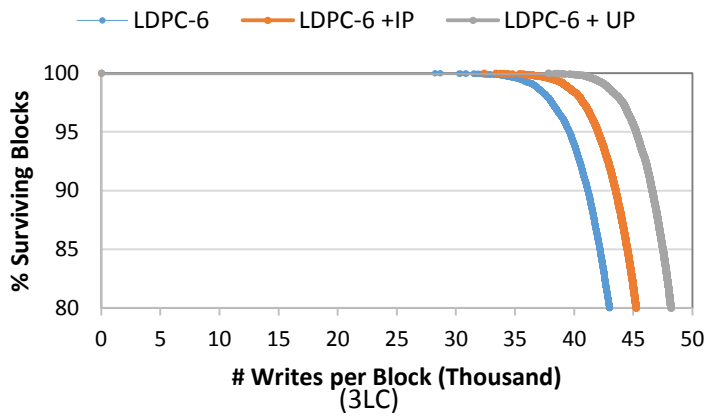

Figure 48: Lifetime of PCM main memory blocks achieved with LDPC-6 and LDPC-6 complemented by Symbol Shifting with integrated protection (IP) and un-integrated protection (UP).

represents the percentage of surviving blocks as a function of the number of writes per block represented by the X axis.

It is clear from Figure 48 that data shifting is capable of substantially improving the lifetime. After retiring the first memory block, data shifting extends the lifetime of 3LC over LDPC-6 by 14.6% and 33.8% with integrated and un-integrated protection respectively. It is notable that data shifting with un-integrated protection significantly surpasses data shifting with integrated protection in terms of achievable lifetime. This result is attributed to the fact that un-integrated protection guarantees an additional number of faults to be tolerated while the number of additional faults that can be tolerated with integrated protection depends on the distribution of faulty cells within the memory block between the data cells the auxiliary cells. Overall, protecting memory chips solely with an LDPC code leads to an earlier retirement of the blocks. Complementing the error correcting codes with Data Shifting delays the defectiveness of the memory blocks leading to a significant lifetime improvement.

6.4 MULTILEVEL DATA DEPENDENT SPARING

Chapter 5 presented Data Dependent Sparing as a new bad block management technique that exploits the data dependent nature of errors in the context of the stuck-at faults model. Conventional bad block management (Static Sparing) permanently replaces a storage block with a spare block soon after the first write failure. Conversely, Data Dependent Sparing assigns spare blocks temporary and attempts later write requests on the original blocks. Later requests are likely to complete successfully giving the low probability of failure due to data dependent nature of errors in which case the previously assigned spares are reclaimed.

The concept of data dependent errors is not specific to SLC and applies to any k-level storage cells. Nonetheless, the increase in the number of stored level increases the likelihood of errors. For instance, a stuck-at SLC has 50% probably of manifesting errors. When the number of levels to store is increased to 3, i.e. 3LC, the probability that a stuck-at 3LC cell manifesting error increases to 66.6%. This increase in probability of error manifestation directly affects the probability of failing to write on a block. Figure 49 compares the probability of failing to write on SLC and 3LC 4KB blocks protected by BCH-20 and LDPC-20 respectively. The plot reveals that the probability of failing to write increases by multiple orders of magnitude for the 3LC blocks. Accordingly, this chapter investigates whether the application of Data Dependent Sparing in the Context of 3LC is effective in squeezing additional lifetime in light of the increase in the probability of failing to write.

Applying Data Dependent Sparing in the context of 3LC does not require any changes to the flow of execution adopted in the context of SLC. The only difference with respect to that of the SLC context is the increase of the probability of failure as noted above. Accordingly, Data Dependent Sparing is applied in the 3LC context without any modifications.

6.4.1 Multilevel Data Dependent Sparing Evaluation

To evaluate Data Dependent Sparing with 3LC storage cells, the lifetime increase in comparison to conventional bad block management is studied. To study the lifetime, the experimental settings in Section 5.3 in Chapter 5 are repeated while the storage cells being modelled are 3LC. Accordingly,

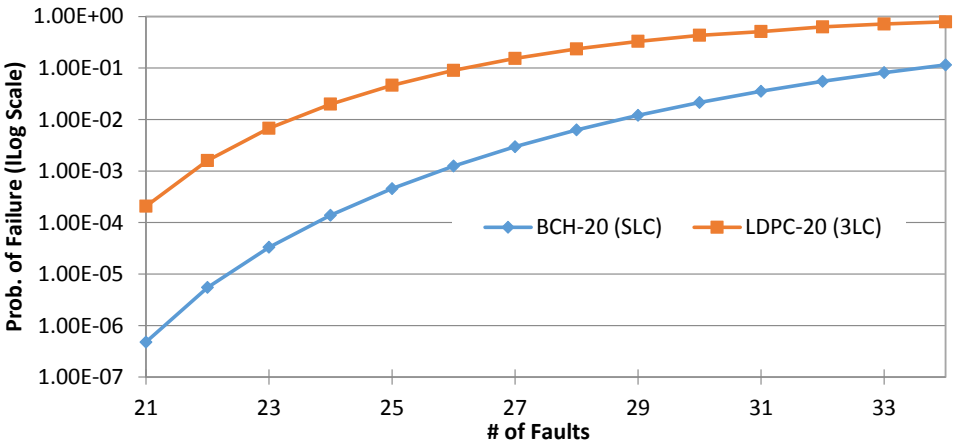

Figure 49: Probability of failing to write on 4KB SLC and 3LC blocks protected by BCH-20 and LDPC-20 respectively.

the lifetime of cells is drawn from the Gaussian distribution with a mean of 10^5 and a standard deviation of 25×10^3. Figure 50 shows the results.

Figure 50 revels that Data Dependent Sparing in the context of 3LC does not contribute to a remarkable increase in the lifetime. When 20% of the storage blocks are retired, the lifetime increase is up to 4% compared to conventional bad block management (SS). The increase is 14% less than that achieved in the context of SLC. Such a result is a direct consequence of the increase in the probability of failing to write on 3LC as was studied in Figure 49. Overall, Data Dependent Sparing is capable of increase the lifetime in the context of 3LC storage cells. Nevertheless, the achieved lifetime does not show a substantial advantage over the conventional bad block management approach.

6.5 SUMMARY

This chapter presents the modifications and changes required to apply the proposed technique at the pre-write fault avoidance, post-write fault tolerance and post-failure recovery stages in the context

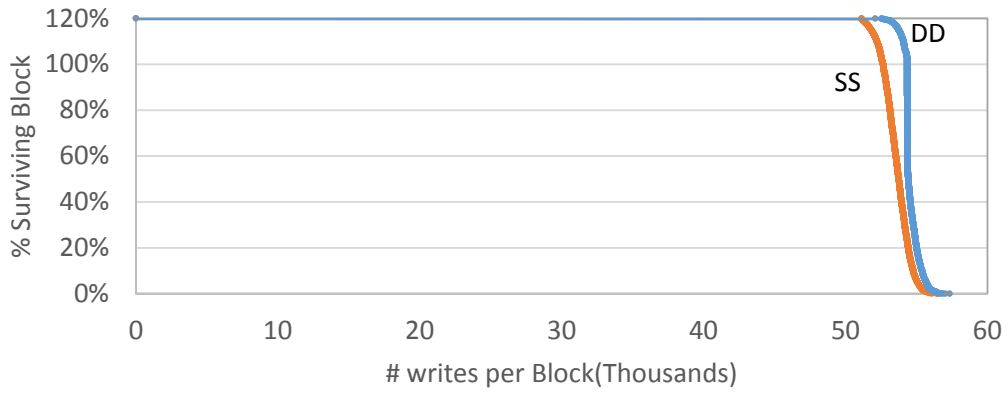

Figure 50: Lifetime Improvement with Data Dependent Sparing (DD) in the context of 3LC in comparison to Static Sparing (SS).

of multilevel cells. The main contributions of this chapter are the following:

1. Extends CAFO's cost model and encoding/decoding engine to accommodate MLC storage cells. Multilevel CAFO was shown to effectively decrease the write endurance cost in comparison to existing schemes by up to 19%.
2. Shows the effect of applying Symbol Shifting in the context of MLC. Applying Symbol Shifting to 3LC storage cells increases the number of faults that can be tolerated within a block by up to 50% of the nominal capability of the deployed error correcting code. This increase in the number of faults was shown to lead to a substantial increase in the memory lifetime by up to 33%.
3. Presents the application of Data Dependent Sparing with multilevel cell. The assessment has shown the probability of a stuck at cell manifesting errors increases with the increase in the number of levels that a cell is storing. Consequently, Data Dependent Sparing showed a relatively low impact on improving the lifetime of memory devices that amounts up to 4%.

7.0 SUMMARY AND FUTURE DIRECTIONS

As the amount of data produced yearly is projected to attain unprecedented levels, researches and scientists are faced with the challenge of offering reliable and efficient exascale computing paradigms. The memory system is expected to play a key role to sustain the development of computer systems. Unfortunately, DRAM and NAND flash are facing physical limitations putting their scalability into question. Accordingly, augmenting the memory system with new memory technologies to overcome the challenges faced by the current technologies is required. Among many candidates, Phase-Change Memory (PCM) is emerging as one of the most promising technologies due to its desirable characteristics in terms of scalability, low access latency and negligible standby power. Yet, PCM suffers from a limited endurance problem that needs to be addressed before it can be deployed within the memory stack. This thesis envisions that tackling PCM's endurance challenge can be done at three different stages: the pre-write fault avoidance, the post-write fault tolerance and the post-failure recovery. The first stage aims at preserving the endurance of memory cells through decelerating the wear-out rate. The second stage aims at tolerating the errors caused by worn-out cells. The third stage aims at managing memory blocks suffering from a large number of worn-out cells.

At the pre-write fault avoidance stage, this thesis presents *Cost Aware Flip Optimization* (CAFO) as technique capable of preserving the endurance of PCM cells. CAFO consists of a cost model that captures the endurance cost of programming a memory block cells. The cost model is aware of the endurance cost asymmetry when programming different cell values. On top of the cost model, CAFO introduce a new encoding/decoding engine that aims at reducing the endurance cost of servicing write requests. This thesis shows that CAFO can significantly decrease the endurance cost in comparison to the existing leading schemes.

At the post-write fault tolerance stage, this thesis presents *Data Inversion* (DI) as a technique

that can increase the number of faults that error correcting codes can tolerate within a memory block while incurring only one additional memory cell. When a write to a block fails, DI attempts an additional write operation after inverting the data symbols of the memory block to a new memory level. This thesis shows and proofs that such an approach can increase the number of faults that an error correcting scheme can tolerate by up to double its nominal capability. DI evaluation shows a notable potential in increasing the memory lifetime. In addition to DI, this thesis presents *extension to RDIS* which is an error correcting schemes designed specifically for the stuck-at fault model. The extension can recover from detrimental faults pattern and can decrease the space overhead incurred by RDIS.

At the post-failure recovery, this thesis presents *Data Dependent Sparing* (DDS) as a new physical sparing technique. DDS exploits the fact that write block failure is notably low after of the number of faults gets above the capability of the deployed error correcting code due to the data dependent nature of errors. DDS delays the retirement of defective memory block through a dynamic assigning of memory spares upon write block failure. When a later write to the same block comes in, DDS attempts the write on the same block and reclaim the previously assigned spare in case the write completes successfully. This thesis shows that DDS is capable of significantly extending the lifetime of memory devices and decrease the required spare over-provisioning.

All the proposed work in this thesis departs from the observation that the endurance of PCM depends on the write data. Accordingly, this thesis shows that by being aware of the endurance degradation asymmetry, by exploiting the data dependent nature of errors that stuck-at cells exhibit and by taking advantage of the data dependent write failures, the endurance of PCM can be effectively salvaged.

7.1 FUTURE DIRECTIONS

This thesis contributes to alleviating the endurance problem in PCM making it a reliable memory that can be integrated in functional systems. The memory technologies that form the building block in today's memory system have been used for decades. The memory hierarchy has been defined and evolved in light of the tradeoffs characterizing the in-use memory technologies. Simi-

larly, system software have been optimized around those tradeoffs. Incorporating PCM brings new tradeoffs in terms of performance, power and cost that makes rethinking the memory hierarchy and the software stack a necessity.

A simple replacement of current memory technologies with emerging non-volatile memories would fail to leverage the enhancements that new technologies enable. For example, system designers have for long optimized computer systems to cope with the volatility and destructive reads of DRAM. Replacing DRAM with PCM without exploiting the non-volatility of PCM would incur the cost of an un-required overhead for PCM based computer systems. Similarly, failing to exploit the byte-addressability and in-place updates of PCM would keep the high access latency imposed by the block-based access interface of storage systems. Emerging non-volatile memories enable a DRAM like interface to storage devices allowing applications to access data at a finer granularity than sector size blocks. Hence, system software should be redesigned to accommodate the enhancement offered by the emerging memory technology.

For long, the memory hierarchy has been characterized by the access time gap in between the main memory and the secondary storage. The introduction of emerging non-volatile memories such as PCM is expected to bring orders of magnitude decrease in access time to secondary storage which makes the gap between the main and secondary memories narrower than ever. This makes the emergence of a universal memory characterized by fast random access, density, non-volatility and zero idle energy and made with a single memory technology viable. Consequently, the lines separating the multi-tier hierarchical memory system becomes blurry calling for a revamped memory system hierarchy.

The work that this thesis presents salvages the endurance of PCM which makes the enhancements that PCM brings to computer system readily available. Accordingly, this thesis paves the way to rethinking the memory hierarchy and the software stack in order to allow computer systems to exploit the advancement enabled by PCM. A reliable, dense, power efficient and fast memory system would catalyze the building of future exascale systems providing the ability to harness and process the unprecedented amount of data.

BIBLIOGRAPHY

[1] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *J. Emerg. Technol. Comput. Syst.*, 2013.

[2] G. Atwood. Emerging memory impact on the memory hierarchy. In *2014 14th Non-Volatile Memory Technology Symposium*, NVMTS, 2014.

[3] F. Bedeschi, E. Bonizzoni, G. Casagrande, R. Gastaldi, C. Resta, G. Torelli, and D. Zella. Set and reset pulse characterization in bjt-selected phase-change memories. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, 2005.

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

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

[6] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande. An 8mb demonstrator for high-density 1.8v phase-change memories. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, 2004.

[7] R. Bose and D. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 1960.

[8] K. Bourzac. Memristor memory readied for production. http://www.technologyreview.com/computing/25018/, April 2010.

[9] M. Breitwisch. Phase change memory. In *Interconnect Technology Conference, 2008. IITC 2008. International*, 2008.

[10] Y. Cai, E. Haratsch, M. McCartney, and K. Mai. Current and emerging memory technology landscape. In *Flash Memory Summit*, 2011.

[11] Y. Cai, E. Haratsch, M. McCartney, and K. Mai. Fpga-based solid-state drive prototyping platform. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, 2011.

[12] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Threshold voltage distribution in mlc nand flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, 2013.

[13] J. Carlson and D. Stolaraski. The correct solution to berlekamp's switching game. *Discrete Mathematics*, 2004.

[14] E. Chen, D. Apalkov, Z. Diao, A. Driskill-Smith, D. Druist, D. Lottis, V. Nikitin, X. Tang, S. Watts, S. Wang, S. Wolf, A. W. Ghosh, J. Lu, S. J. Poon, M. Stan, W. Butler, S. Gupta, C. K. A. Mewes, T. Mewes, and P. Visscher. Advances and future prospects of spin-transfer torque random access memory. *Magnetics, IEEE Transactions on*, 2010.

[15] S. Cho and H. Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009.

[16] Y. Choi, I. Song, M.-H. Park, H. Chung, B. C. Sanghoan Chang, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Want, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J.-H. Yoo, and G. Jeong. A 20nm 1.8V 8Gb PRAM with 40MB/s Program Bandwidth. In *IEEE ISSCC*, 2012.

[17] H. Chung, B. H. Jeong, B. Min, Y. Choi, B.-H. Cho, J. Shin, J. Kim, J. Sunwoo, J. min Park, Q. Wang, Y. jun Lee, S. Cha, D. Kwon, S. Kim, S. Kim, Y. Rho, M.-H. Park, J. Kim, I. Song, S. Jun, J. Lee, K. Kim, K. won Lim, W. ryul Chung, C. Choi, H. Cho, I. Shin, W. Jun, S. Hwang, K.-W. Song, K. Lee, S. whan Chang, W. Y. Cho, J.-H. Yoo, and Y.-H. Jun. A 58nm 1.8V 1Gb PRAM with 6.4MB/s Program BW. In *IEEE ISSCC*, 2011.

[18] H. Chung, B.-H. Jeong, B. Min, Y. Choi, B.-H. Cho, J. Shin, J. Kim, J. Sunwoo, J. min Park, Q. Wang, Y.-J. Lee, S. Cha, D. Kwon, S. Kim, S. Kim, Y. Rho, M.-H. Park, J. Kim, I. Song, S. Jun, J. Lee, K. Kim, K. won Lim, W. ryul Chung, C. Choi, H. Cho, I. Shin, W. Jun, S. Hwang, K.-W. Song, K. Lee, S. whan Chang, W.-Y. Cho, J.-H. Yoo, and Y.-H. Jun. A 58nm 1.8v 1gb pram with 6.4mb/s program bw. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, 2011.

[19] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, 2009.

[20] X. Dong and Y. Xie. Adams: adaptive mlc/slc phase-change memory design for file storage. ASPDAC '11, 2011.

[21] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem. Bit mapping for balanced pcm cell programming. *SIGARCH Comput. Archit. News*, 2013.

[22] G. W. B. et al. Phase change memory technology. *Journal of Vacuum Science and Technology B*, 2013.

[23] e. a. F. Yeung. $ge_2sb_2te_5$ confined structures and integration of 64mb phase-change random access memory. *Japanese Journal of Applied Physics*, 2005.

[24] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 2008.

[25] R. Gallager. Low-density parity-check codes. *Information Theory, IRE Transactions on*, 1962.

[26] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. In *IDC iView: IDC Analyze the, Future*, 2012.

[27] B. Gleixner, F. Pellizzer, and R. Bez. Reliability characterization of phase change memory. NVMTS, 2009.

[28] R. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, (2):147 – 160, 1950.

[29] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[30] ITRS. http://public.itrs.net, 2012.

[31] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Coset coding to extend the lifetime of memory. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, 2013.

[32] L. Jiang, Y. Du, Y. Zhang, B. Childers, and J. Yang. Lls: Cooperative integration of wear-leveling and salvaging for pcm main memory. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, 2011.

[33] L. Jiang, Y. Zhang, B. R. Childers, and J. Yang. Fpb: Fine-grained power budgeting to improve write throughput of multi-level cell phase change memory. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[34] L. Jiang, Y. Zhang, and J. Yang. Er: Elastic reset for low power and long endurance mlc based phase change memory. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, 2012.

[35] M. Joshi, W. Zhang, and T. Li. Mercury: A fast and energy-efficient multi-level cell based phase change memory system. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[36] S. Kang, W. Y. Cho, B.-H. Cho, K.-J. Lee, C.-S. Lee, H.-R. Oh, B.-G. Choi, Q. Wang, H.-J. Kim, M.-H. Park, Y. H. Ro, S. Kim, C.-D. Ha, K.-S. Kim, Y.-R. Kim, D.-E. Kim, C.-K. Kwak, H.-G. Byun, G. Jeong, H. Jeong, K. Kim, and Y. Shin. A 0.1 μm 1.8-V 256-Mb Phase-Change Random Access Memory (PRAM) With 66-MHz Synchronous Burst-Read Operation. *Solid-State Circuits, IEEE Journal of*, 2007.

[37] H. Kim, M. Sah, C. Yang, and L. Chua. Memristor-based multilevel memory. In *Cellular Nanoscale Networks and Their Applications (CNNA), 2010 12th International Workshop on*, 2010.

[38] K. Kim. Technology for sub-50nm DRAM and NAND flash manufacturing. 2005.

[39] K. Kim and S. J. Ahn. Reliability investigations for manufacturable high density pram. In *Reliability Physics Symposium, 2005. Proceedings. 43rd Annual. 2005 IEEE International*, 2005.

[40] K. Kim and S. J. Ahn. Reliability investigations for manufacturable high density pram. In *Reliability Physics Symposium, 2005. Proceedings. 43rd Annual. 2005 IEEE International*, 2005.

[41] K. Kim and S. J. Ahn. Reliability investigations for manufacturable high density pram. IRPS, 2005.

[42] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013.

[43] B. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger. Phase-change technology and the future of main memory. *Micro, IEEE*, 2010.

[44] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH Comput. Archit. News*, 2009.

[45] K.-J. Lee, B.-H. Cho, W.-Y. Cho, S. Kang, B.-G. Choi, H.-R. Oh, C.-S. Lee, H.-J. Kim, J.-M. Park, Q. Wang, M.-H. Park, Y.-H. Ro, J.-Y. Choi, K.-S. Kim, Y.-R. Kim, I.-C. Shin, K.-W. Lim, H.-K. Cho, C.-H. Choi, W.-R. Chung, D.-E. Kim, Y.-J. Yoon, K.-S. Yu, G.-T. Jeong, H.-S. Jeong, C.-K. Kwak, C.-H. Kim, and K. Kim. A 90 nm 1.8 V 512 Mb Diode-Switch PRAM With 266 MB/s Read Throughput. *IEEE JSSC*, 2008.

[46] S. Lee, J. hyun Jeong, T. S. Lee, W. M. Kim, and B. ki Cheong. A Study on the Failure Mechanism of a Phase-Change Memory in Write-Erase Cycling. *IEEE Electron Device Letters*, 2009.

[47] R. Maddah, S. Cho, and R. Melhem. Data dependent sparing to manage better-than-bad blocks. *Computer Architecture Letters*, 2013.

[48] R. Maddah, S. Cho, and R. Melhem. Power of one bit: Increasing error correction capability with data inversion. In *19th Pacific Rim International Symposium on Dependable Computing*, PRDC '13, 2013.

[49] R. Maddah, R. Melhem, and S. Cho. Rdis: Tolerating many stuck-at faults in resistive memory. *Computers, IEEE Transactions on*, March 2015.

[50] R. Maddah, R. Melhem, and S. Cho. Symbol shifting: Tolerating morefaults in pcm blocks. *Computers, IEEE Transactions on*, (under review).

[51] R. Maddah, S. M. Seyedzadeh, and R. Melhem. Cafo: Cost aware flip optimization for asymmetric memories. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, 2015.

[52] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 2002.

[53] R. Melhem, R. Maddah, and S. Cho. Rdis: A recursively defined invertible set scheme to tolerate multiple stuck-at faults in resistive memory. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, 2012.

[54] I. Micron Technology. Phase change memory (pcm). http://www.micron.com/products/pcm, 2011.

[55] Micron Technology, Inc. Nand flash design and use considerations. www.micron.com, 2006.

[56] Micron Technology, Inc. Bad block management in nand flash memory. www.micron.com, 2011.

[57] T. Nirschl, J. Philipp, T. Happ, G. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. Chen, Y. Zhu, R. Bergmann, H. L. Lung, and C. Lam. Write strategies for 2 and 4-bit multi-level phase-change memory. In *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, 2007.

[58] S. Ovshinsky. Symmetrical current controlling device. In *US Patent Number 3271591*, 1966.

[59] N. Papandreou, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, H. Pozidis, and E. Eleftheriou. Multilevel phase-change memory. In *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*, 2010.

[60] N. Papandreou, H. Pozidis, T. Mittelholzer, G. Close, M. Breitwisch, C. Lam, and E. Eleftheriou. Drift-tolerant multilevel phase-change memory. In *Memory Workshop (IMW), 2011 3rd IEEE International*, 2011.

[61] N. Papandreou, H. Pozidis, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, and E. Eleftheriou. Programming algorithms for multilevel phase-change memory. In *Circuits and Systems (ISCAS), 2011 IEEE International Symposium on*, 2011.

[62] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi. A high performance controller for nand flash-based solid state disk (nssd). In *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*, 2006.

[63] M. Qureshi, M. Franceschini, A. Jagmohan, and L. Lastras. Preset: Improving performance of phase change memories by exploiting asymmetry in write times. In *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, 2012.

[64] M. Qureshi, A. Seznec, L. Lastras, and M. Franceschini. Practical and secure pcm systems by online detection of malicious write streams. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[65] M. Qureshi, A. Seznec, L. Lastras, and M. Franceschini. Practical and secure PCM systems by online detection of malicious write streams. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[66] 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*, MICRO-44, 2011.

[67] 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*, MICRO 42, 2009.

[68] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[69] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *SIGARCH Comput. Archit. News*, 2009.

[70] M. Rahman, B. R. Childers, and S. Cho. Comet: Continuous online memory test. In *Proceedings of the 17th IEEE Pacific Rim Int'l Symposium on Dependable Computing (PRDC)*, 2011.

[71] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 2008.

[72] R. Roth and K. Viswanathan. On the hardness of decoding the gale-berlekamp code. *Information Theory, IEEE Transactions on*, 2008.

[73] S. Schechter, G. H. Loh, K. Straus, and D. Burger. Use ECP, not ECC, for hard failures in resistive memories. *SIGARCH Comput. Archit. News*, 2010.

[74] 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. *SIGARCH Comput. Archit. News*, 2010.

[75] 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. *SIGARCH Comput. Archit. News*, 2010.

[76] N. H. Seong, D. H. Woo, V. Srinivasan, J. Rivers, and H.-H. Lee. SAFER: Stuck-At-Fault Error Recovery for Memories. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, 2010.

[77] N. H. Seong, S. Yeo, and H.-H. S. Lee. Tri-level-cell phase change memory: Toward an efficient and reliable memory system. *SIGARCH Comput. Archit. News*, 2013.

[78] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. HPCA, 2011.

[79] D. Strukov. The area and latency tradeoffs of binary bit-parallel BCH decoders for prospective nanoelectronic memories. In *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, 2006.

[80] D. Tang, P. Carruthers, Z. Totari, and M. Shapiro. Assessment of the effect of memory page retirement on system ras against hardware faults. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, 2006.

[81] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. FAST'11, 2011.

[82] W. Wong. A chat about micron's clearnand technology. *electronic design*, 2010.

[83] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme. ISCAS, 2007.

[84] D. H. Yoon, J. Chang, R. S. Schreiber, and N. P. Jouppi. Practical nonvolatile multilevel-cell phase change memory. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[85] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. Jouppi, and M. Erez. FREE-p: Protecting non-volatile memory against both hard and soft errors. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011.

[86] J. Yue and Y. Zhu. Accelerating write by exploiting pcm asymmetries. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, 2013.

[87] W. Zhang and T. Li. Characterizing and mitigating the impact of process variations on phase change based memory systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[88] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *SIGARCH Comput. Archit. News*, 2009.