A RELIABLE, SECURE PHASE-CHANGE MEMORY AS A MAIN MEMORY

A Dissertation
Presented to
The Academic Faculty

By

Nak Hee Seong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in the
School of Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2012

Copyright © 2012 by Nak Hee Seong

A RELIABLE, SECURE PHASE-CHANGE MEMORY AS A MAIN MEMORY

Approved by:

Dr. Hsien-Hsin S. Lee, Advisor
Assoc. Professor, School of ECE
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
Professor, School of ECE
Georgia Institute of Technology

Dr. Moinuddin K. Qureshi
Assoc. Professor, School of ECE
Georgia Institute of Technology

Dr. Sung Kyu Lim
Assoc. Professor, School of ECE
Georgia Institute of Technology

Dr. Hyesoon Kim
Asst. Professor, College of Computing
Georgia Institute of Technology

Date Approved: July, 2012

To my loving wife, two dear sons, and wonderful parents.

ACKNOWLEDGMENTS

I would like to express sincere gratitude to my advisor, Dr. Hsien-Hsin S. Lee, for his guidance and support throughout my Ph.D. study. He always motivated, supervised, and encouraged me with his profound insight and extensive knowledge. Also, I would like to thank my committee members, Dr. Sudhakar Yalamanchili, Dr. Moinuddin K. Qureshi, Dr. Sung Kyu Lim, and Dr. Hyesoon Kim, for their time and valuable feedback.

I also thank all the MARS lab-mates, Dr. Dong Hyuk Woo, Dr. Dean Lewis, Eric Fontaine, Jen-Cheng Huang, Sungkap Yeo, Tzu-Wei Lin, Mohammad Hossain, Guanhao Shen, and Lifeng Nai for their help and comments. Especially, I thank Dr. Dong Hyuk Woo for thoughtful discussions to improve the quality of papers we published. Another special thanks to Sungkap Yeo for our collaboration.

I am also grateful to Dr. Seh-Woong Jeong and Dr. Jaehong Park who supervised me for more than seven years in Samsung Electronics and gave me an opportunity to pursue a Ph.D. degree.

My family was the source of my power for this accomplishment. I want to thank my wife, Sun Ah Park. She always supports me and encourages me. Her love and sacrifices made this achievement possible. Also, I want to thank my sons, Jeong Moh Seong and Kyeong Moh Seong, who always give me joy. I sincerely thank my parents, Si-Yong Seong and Yeon-Soo Lim, for their endless love and support. They are always proud of me and on my side throughout my life.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

SUMMARY xi

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 PRIOR WORK FOR PCM RELIABILITY AND THE WEAKNESSES 6

2.1 Vulnerability of Prior Wear-Out Management Schemes 6
2.2 Prior Error-Correction Schemes 7
2.3 Prior Hybrid-Memory Architecture 8
2.4 Prior Resistance-Drift Resilient Schemes 9

CHAPTER 3 SECURITY REFRESH: PROTECT PHASE-CHANGE MEMORY AGAINST MALICIOUS WEAR-OUT 10

3.1 Security Refresh 10
3.1.1 Security-Refresh Controller 10
3.1.2 The Basics of Distributed Security Refresh 11
3.1.3 Security-Refresh Algorithm 12
3.1.4 Key Selection for Address Translation 16
3.1.5 Implementing Security-Refresh Controller 17
3.1.6 Memory-Controller Design Issues 18
3.1.7 Testability 19
3.2 Implementation Trade-Off of Security Refresh 19
3.3 Two-Level Security Refresh 20
3.4 Evaluation 23
3.4.1 Robustness and Write Overhead 23
3.4.2 Hardware Overhead 26
3.4.3 Wear Leveling 28
3.4.4 Performance Impact 29
3.5 Summary 31

CHAPTER 4 SAFER: STUCK-AT-FAULT ERROR RECOVERY FOR MEMORIES 33

4.1 SAFER: Stuck-At-Fault Error Recovery 33
4.1.1 Partition Technique for Double Error Correction 33
4.1.2 Partition Technique for Multi-Bit Error Correction 36
4.1.3 Using Data-Block Inversion 38
4.1.4 Putting It All Together 40

4.2 Efficient Implementation of SAFER . . . 42
4.2.1 The Location of SAFER Logic . . . 43
4.2.2 Ideal Data Size for SAFER Effectiveness . . . 43
4.2.3 The Area Overhead of Fail Cache . . . 44
4.3 Methodology . . . 45
4.3.1 Experimental Setup . . . 47
4.4 Results . . . 48
4.4.1 Lifetime Improvement . . . 48
4.4.2 The Number of Fails Recovered . . . 50
4.4.3 Meta-Bit Overhead vs. Lifetime Improvement . . . 50
4.4.4 SAFER with Fail Cache . . . 51
4.5 Summary . . . 54

CHAPTER 5 WRITE-FREQUENCY REDUCTION METHODS . . . 55
5.1 Multi-Dimensional Classification . . . 55
5.2 Interference and Its Implication to Design . . . 59
5.3 The Implementation of Our Hybrid-PCM Architecture . . . 61
5.3.1 Overall Control Flow and Isolation Cache . . . 61
5.3.2 Decision Maker . . . 62
5.3.3 Implementation Overhead . . . 65
5.4 Impact to Wear Leveling under Malicious Attacks . . . 66
5.5 Experimental Evaluation and Analysis . . . 68
5.5.1 Sensitivity Study for Interference . . . 68
5.5.2 Wear-Out Evaluation . . . 70
5.5.3 Evaluation for Normal Applications . . . 71
5.5.4 Impact to Wear Leveling . . . 72
5.6 Related Work . . . 75
5.7 Summary . . . 77

CHAPTER 6 TRI-LEVEL-CELL PHASE CHANGE MEMORY: TOWARD AN EFFICIENT AND RELIABLE MEMORY SYSTEM . . . 79
6.1 Tri-Level-Cell (3LC) PCM . . . 81
6.2 Analytical Error Model and Validation . . . 84
6.3 Revisiting Four-Level-Cell (4LC) PCM . . . 85
6.3.1 Estimating Scrubbing Overhead . . . 86
6.3.2 Reducing Capacity to Achieve Low Soft Error Rates . . . 87
6.3.3 Using Error-Correcting Codes . . . 89
6.4 Tri-Level-Cell (3LC) PCM in Practice . . . 92
6.4.1 Binary-to-Ternary Conversion . . . 92
6.4.2 Bandwidth Enhancement . . . 95
6.4.3 Efficient $\langle 3, 2\rangle$ Conversion for Error Correction . . . 97
6.5 Evaluation . . . 99
6.5.1 Soft Error Rate of BE-3LC PCM . . . 99
6.5.2 Performance . . . 101

6.5.3 Information Density 103
6.6 Summary 105

CHAPTER 7 CONCLUSIONS 107

REFERENCES 110

LIST OF TABLES

Table 1 Notations used in the proof. 16

Table 2 SRAM fail cache overhead for an 8Gb PCM chip. 45

Table 3 Applications with more than 1M writebacks to memory. 52

Table 4 Configuration Variables of Four-Level-Cell (4LC) PCM When $t_0 = 1$ s. . 82

Table 5 Configuration Variables of Tri-Level-Cell (3LC) PCM When $t_0 = 1$ s. . . 83

Table 6 Probability of Soft Error of Four-Level-Cell (4LC) PCM by Equation (4) in Section 6.2 83

Table 7 Probability of Soft Error of Tri-Level-Cell (3LC) PCM by Equation (4) in Section 6.2 84

Table 8 Probability of Uncorrectable Errors by ECC and $SER_{combined}$ for 2GB per Bank 4LC PCM 88

Table 9 Maximum Capacity Per Bank of Four-Level-Cell (4LC) PCM by Soft Error Rates and Scrubbing Overhead 89

Table 10 An example of the $\langle 3, 2 \rangle$ number mapping method. 95

Table 11 Physical Parameters for the Second Storage Level of 3LC PCM When $t_0 = 1$ s. 100

Table 12 Soft Error Rates of Intermediate Storage Level of BE-Three-Level-Cell (BE-3LC) PCM 101

LIST OF FIGURES

Figure 1 Security-refresh terminology. 13

Figure 2 An example of one complete security-refresh round. 14

Figure 3 The block diagram of security-refresh controller (SRC). 18

Figure 4 One-level security refresh (four ranks, four banks per rank). 20

Figure 5 Two-level security refresh (four ranks, four banks per rank). 21

Figure 6 Two-level security refresh within a bank. 23

Figure 7 Single-level robustness. 25

Figure 8 Two-level robustness vs. sub-regions. 26

Figure 9 Two-level robustness vs. refresh intervals. 27

Figure 10 Two-level hardware cost per 512MB PCM chip. 28

Figure 11 The accumulated number of writes over the memory space. 30

Figure 12 Relative IPC. 31

Figure 13 An example of partitioning two fails. 34

Figure 14 An example of four-group partition. 37

Figure 15 Fail-cache organization. 41

Figure 16 The sequence of a write request in SAFER. 42

Figure 17 An example of SAFER. 42

Figure 18 DRAM architecture. 44

Figure 19 Hardware overhead for recovery schemes. 46

Figure 20 The definition of lifetime improvement. 49

Figure 21 The relative lifetime of a 256B memory block. 50

Figure 22 Fail recovery in a 256B memory block. 51

Figure 23 Meta-bit contribution for lifetime. 51

Figure 24 Fail-cache miss rate. 53

Figure 25 The relative lifetime improvement of SAFER with a fail cache. 53

Figure 26 Examples of multi-dimensional classification. 57
Figure 27 Examples of interference. 59
Figure 28 The overall block diagram of our hybrid-PCM architecture. 62
Figure 29 The block diagram of a decision maker. 63
Figure 30 The details of a hash function and a counter manager. 64
Figure 31 The effect of write distribution on wear leveling. 67
Figure 32 The rate of false-negative interference. 69
Figure 33 The rate of false-positive interference. 70
Figure 34 The maximum wear-out of attack addresses in 300 simulations. 71
Figure 35 The saved PCM writes of SPEC2006 for 100K writebacks. 72
Figure 36 Lifetime improvement using single-level security refresh. 74
Figure 37 Probability of Soft Error of Four-Level-Cell (4LC) PCM Over Time . . . 86
Figure 38 Tri-Level-Cell Utilization 93
Figure 39 Cell Distribution vs. Programming Sequence 96
Figure 40 State mapping of $\langle 3, 2 \rangle$ conversion for efficient error correction in 3LC PCM 98
Figure 41 Performance comparison with 4LC and 3LC 103
Figure 42 Sensitivity study of bandwidth-enhanced 3LC 103
Figure 43 Information density of 4LC PCM 105

SUMMARY

The main objective of this research is to provide an efficient and reliable method for using multi-level cell (MLC) phase-change memory (PCM) as a main memory. As DRAM scaling approaches the physical limit, alternative memory technologies are being explored for future computing systems. Among them, PCM is the most mature with announced commercial products for NOR flash replacement. Its fast access latency and scalability have led researchers to investigate PCM as a feasible candidate for DRAM replacement. Moreover, the multi-level potential of PCM cells can enhance the scalability by increasing the number of bits stored in a cell.

However, the two major challenges for adopting MLC PCM are the limited write endurance cycle and the resistance drift issue. To alleviate the negative impact of the limited write endurance cycle, this thesis first introduces a secure wear-leveling scheme called Security Refresh. In the study, this thesis argues that a PCM design not only has to consider normal wear-out under normal application behavior, most importantly, it must take the worst-case scenario into account with the presence of malicious exploits and a compromised OS to address the durability and security issues simultaneously. Security Refresh can avoid information leak by constantly migrating their physical locations inside the PCM, obfuscating the actual data placement from users and system software.

In addition to the secure wear-leveling scheme, this thesis also proposes SAFER, a hardware-efficient multi-bit stuck-at-fault error recovery scheme which can function in conjunction with existing wear-leveling techniques. The limited write endurance leads to wear-out related permanent failures, and furthermore, technology scaling increases the variation in cell lifetime resulting in early failures of many cells. SAFER exploits the key attribute that a failed cell with a stuck-at value is still readable, making it possible to continue to use the failed cell to store data; thereby reducing the hardware overhead for error recovery.

Another approach that this thesis proposes to address the lower write endurance is a hybrid phase-change memory architecture that can dynamically classify, detect, and isolate frequent writes from accessing the phase-change memory. This proposed architecture employs a small SRAM-based Isolation Cache with a detection mechanism based on a multi-dimensional Bloom filter and a binary classifier. The techniques are orthogonal to and can be combined with other wear-out management schemes to obtain a synergistic result.

Lastly, this thesis quantitatively studies the current art for MLC PCM in dealing with the resistance drift problem and shows that the previous techniques such as scrubbing or error correction schemes are incapable of providing sufficient level of reliability. Then, this thesis proposes tri-level-cell (3LC) PCM and demonstrates that 3LC PCM can be a viable solution to achieve the soft error rate of DRAM and the performance of single-level-cell PCM.

CHAPTER 1

INTRODUCTION

Given the grim prospect of technology scaling in DRAM, researchers recently have a growing interest of seeking for alternative memory technologies and integrating them into the main memory hierarchy of a computing system. The common and salient features of these new classes of memory include non-volatility, high density, fast access time, solid-state without slow, power-consuming mechanical operations, etc. Most importantly, these memories demonstrate better scalability with shrunk feature size than currently deployed memory technologies. Out of several emerging memory candidates, phase-change memory (PCM), which stores data based on the resistivity of material phases, is the most mature. Commercial PCM products from Samsung and Micron-Numonyx have been announced to replace NOR flash for mobile devices, and the processor research community is taking a step further to study the feasibility and the corresponding challenges to move PCM closer to the processor cores in the memory hierarchy [1, 2, 3, 4].

A PCM cell typically uses chalcogenide alloy that consists of **Ge**, **Sb**, and **Te**. The material has two distinct states, namely, a low resistive crystalline state (SET) and a high resistive amorphous state (RESET). The crystalline state can be reached by heating the material above the crystallization temperature while it can be switched into the amorphous state by melting and quickly quenching it. Furthermore, using fine-grained partitioning of the resistance range between the two states, it is possible to store multiple bits per PCM cell. Although PCM is slower than DRAM to read and much slower to write, architecture-level solutions have been explored to mitigate these high latencies and to effectively use PCM as a DRAM replacement for a main memory. However, PCM confronts a few major challenges for the universal adoption, *i.e.,* its low write endurance and resistance drift causing permanent faults and transient faults, respectively.

According to ITRS report, the current write endurance of a PCM cell is around 10^8, which is a few magnitudes lower than today's DRAM. Without considerable enhancement, thus, the weak endurance may bring about lots of reliability issues. To address these reliability challenges, effective and efficient wear-out management schemes must be designed to extend the cell's lifetime or to maintain faultless operations at the presence of dysfunctional cells.

We broadly classify these wear-out management techniques into four types. The first group of techniques simply minimizes the number of memory writes to eliminate silent stores [1, 2, 5, 4] and/or perform writes with an inverted coding method based on Hamming distance [6, 7]. Even though such techniques could extend the endurance to some certain degree, they are of no use in the face of the worst-case write scenarios or deliberately designed malicious write sequences.

The second type is to perform wear-leveling. Similar to those employed in commodity flash memory, wear-leveling techniques aim to evenly distribute the writes across the given memory address space by periodically shuffling the physical locations of memory blocks to mitigate the likelihood of write hot-spots. Given that these new memories can be updated much faster (thus failed quicker) than floating-gate flash memories, malicious wear-out attack, which is a novel security concern, must be taken into account when designing wear-leveling schemes [8, 9].

The third category is to maintain correct memory operations even in the event of permanent faults resulting from aging. Such techniques have the memory operated as if it has self-healing capability. Conventional error correcting mechanisms, commonly found in on-die SRAM and off-chip DRAM, can be classified into this category. Recent proposed architectural techniques such as ECP [10], DRM [11], and FREE-p [12] are also such a type dealing with aged faulty cells.

The final group integrates durable memory (*e.g.,* , DRAM or SRAM) into the less reliable yet bulkier resistive memories to meet the requirement of desirable lifetime. We call such design *hybrid resistive memory*. The design principle is to filter out frequent same-address writes from accessing the resistive main memory. Note that, these four solution classes are completely orthogonal. One can mix and implement them together for resistive memories to achieve synergistic results for reliability and robustness.

Another reliability issue in PCM is incurred from the phenomenon that the resistance of the cell increases over time, which is called resistance drift. Since its major cause is the structural relaxation of the amorphous phase [13], the drift barely affects both of the SET state composed of the crystalline phase and the RESET state that is already high resistive. However, multi-level cell (MLC) PCM uses partial crystalline states, *i.e.,* intermediate states between the two distinct states. Although the MLC PCM can increase the amount of information stored in a cell, the drift can shift the resistance level of a intermediate state to the next adjacent state. Thus, to reliably retrieve the stored states we must place an adequate margin between any two adjacent states to guard each state from the drift. If the margin fails to guard, it produces transient errors threatening PCM reliability.

Recently, four-level (two-bit) cell PCM has been designed and evaluated [14, 15]. However, different from the evolution of NAND flash from two-level to four-level to eight-level, it is too challenging to increase the number of levels in MLC PCM. As the number of levels in a cell increases, the distance between any two adjacent levels becomes too close to secure a reliable margin against resistance drift, which leads to undesirable errors due to the state changes. This new type of soft errors caused by resistance drift, if left unaddressed, will make MLC PCM completely useless.

Therefore, this dissertation focuses on those two reliability issues in PCM such as the limited write endurance and the resistance drift. The first contribution of this research is the finding that the limited write endurance incurs both the durability and security issues simultaneously, and thus, a secure wear-leveling scheme is required to prevent malicious

writes to PCM. In this study, we propose an efficient wear-leveling scheme called Security Refresh which can dynamically change physical address mapping with random keys.

The next observation is that as technology scales, the endurance variation of cells increases and the lifetime of the PCM memory is dictated by the weakest cells. We mitigate the growing variation impact on the PCM lifetime with a multiple stuck-at-fault error recovery scheme. The scheme called SAFER exploits two properties of stuck-at-faults caused by cell aging, *i.e.,* readability and permanency.

Another contribution for protecting PCM from malicious writes is to propose a new hybrid PCM architecture using low-cost hardware for effective wear-out management. In this architecture, a detection mechanism based on a multi-dimensional Bloom filter and a binary classifier isolates malicious writes to a small SRAM cache. This mechanism not only reduces write frequency to PCM main memory but also makes a wear-leveling efficient by conservatively sensing the existence of malicious attacks.

The last contribution of this study is to address the negative impact of resistance drift on the MLC PCM reliability. We mathematically formulate the drift-induced soft-error rates of MLC PCM. With this analytical model, we evaluate the previously proposed ideas for reducing errors and show that four-level PCM is infeasible as main memory without any device-level progress. Then, we propose tri-level-cell (3LC) PCM and shows that 3LC PCM can achieve the soft error rate of DRAM and the performance of single-level-cell (SLC) PCM.

The remainder of this document is organized as follows. Chapter 2 presents the details of prior works related with this research and demonstrates their weaknesses. In Chapter 3, Chapter 4, and Chapter 5, we introduce our proposals to overcome the weaknesses caused by the limited write endurance of PCM. Chapter 3 describes a secure low-cost wear-leveling scheme to protect a limited-write-endurance memory from malicious write attacks. Chapter 4 describes a new stuck-at fault recovery scheme exploiting the properties of stuck-at

faults to reduce hardware costs. Chapter 5 describes a hybrid memory architecture efficiently isolating frequently written memory blocks to an SRAM cache. Chapter 6 proposes a reliable tri-level-cell (3LC) PCM as a main memory and describes efficient ways to use the 3LC PCM in the conventional binary computing systems. Lastly, Chapter 7 concludes this dissertation.

CHAPTER 2

PRIOR WORK FOR PCM RELIABILITY AND THE WEAKNESSES

2.1 Vulnerability of Prior Wear-Out Management Schemes

While phase change memory is often considered as a potential replacement of DRAM, the primary roadblock for using PCM as part of the main memory is its much lower write endurance compared to DRAM. Several recent studies have attempted to address this issue by either reducing PCM's write frequency or using wear-leveling techniques to evenly distribute PCM writes. Although these techniques can extend the lifetime of PCM under normal operations of typical applications, we found that most of them fail to prevent an adversary from writing malicious code deliberately designed to wear out and fail PCM. For instance, the schemes to reduce write frequency, such as *data comparison write* [5] and *Flip-N-Write* [6] do not prevent an adversary from intentionally wearing out the target memory bits, because of their deterministic patterns that can be easily detoured.

In wear-leveling schemes [4, 9], on the other hand, a rush of writes to the same location can be dispersed to different locations by changing physical memory mappings with another address translation layer. However, the prior wear-leveling schemes have the inherent weaknesses caused by regular shuffling pattern, coarse-grained shuffling, and static randomization. From their weaknesses, an adversary can extract mapping information of the additional translation layer and focus on attacking target bits.

Furthermore, all the prior art did not consider the circumstances when the underlying OS is compromised and its security implication to PCM design. A compromised OS will allow adversaries to manipulate all processes and exploit side channels easily, which deduces useful mapping information and accelerates the wear-out of targeted PCM blocks.

2.2 Prior Error-Correction Schemes

Repeating writes to a PCM cell causes the cell to be expanded and contracted repeatedly, which leads to mechanical stress and eventually incurs a permanent stuck-at-fault failure. Furthermore, as technology scales down, the endurance variation of cells increases, which causes the early failure of many cells. In the absence of error recovery techniques, the lifetime of the PCM memory is dictated by the weakest cell. Thus, we need an error recovery scheme capable of correcting multiple stuck-at faults.

The existing error correcting code (ECC) schemes, such as the (72,64) Hamming Coding scheme, can be applied to recover from permanent stuck-at faults even though they are primarily devised for recovering from transient faults. However, unlike transient errors, the number of stuck-at faults gradually grows with time (with repeated write cycles), making it necessary to provide efficient multi-bit error correction capability.

Another important requirement for a stuck-at fault recovery technique is that the technique must operate in the presence of existing wear-leveling algorithms. Otherwise, it makes the memory system vulnerable to malicious attacks, especially when the OS is compromised. To do so, it should be lightweight enough to be embedded inside a chip, since the existing wear-leveling schemes typically have their own address translation layer in either the memory controller or the chip itself.

Recently, architectural techniques have been proposed to overcome multiple stuck-at faults in PCM [11, 10]. Ipek *et al.* proposed Dynamic Pairing scheme to reuse faulty pages [11]. In the Dynamic Pairing scheme, each byte has its own fail indication bit. If a new fail occurs, the indication bit of the corresponding byte is set and the OS adds the corresponding page to a waiting list of faulty pages. On a page allocation, the OS selects a pair of faulty pages such that their fail bits are not at the same offset within the page. One of the pages of the pair is maintained as the primary copy, and the other as a backup copy. Dynamic Pairing provides the ability to reuse faulty pages with more than one fail bit per data block. However, since the OS manages faulty pages, this scheme makes the

memory system vulnerable to malicious attacks, especially when the OS is compromised as mentioned in Section 2.1.

Error-Correcting Pointer (ECP) scheme [10] stores six fail pointers for each 512 bits of data block and replaces the fail cells with extra/spare cells. This ECP scheme is more efficient than the (72,64) code from the standpoint of both hardware overhead and fail recovery because it can recover six fails per 512 bits with 61-bit overhead. Furthermore, this technique operates in the presence of existing wear-leveling algorithms.

2.3 Prior Hybrid-Memory Architecture

To extend the lifetime of PCM, the first priority is to reduce the absolute number of writes to the physical memory cells. Toward this effort, processor architects have suggested to enlarge the size of the last-level cache (LLC) [2] or employ a deeper delayed write queue [9]. Given the presence of data temporal locality, the larger LLC can help to collapse multiple writes to the same location, reducing the total number of writes to the PCM main memory. Essentially, the large LLC is used as a write shield to filter out write accesses with high temporal locality.

However, this simplistic solution has several drawbacks. First, the expected data recurrence in the write buffers or LLC may take a long time to be observed and captured. Worse yet, this design will not defend the worst-case scenarios or malicious attacks where an adversary can intentionally concoct a process with specific cache miss patterns to bypass the LLC and directly write to the off-chip PCM as described in Section 2.1. Therefore, we need a more effective, robust protection mechanism to guarantee usable lifetime under the circumstances of worst-case write patterns and/or malicious wear-out attacks.

As briefly mentioned in Chapter 1, one way to extending the lifetime of limited-endurance memory is to have a hybrid memory architecture by integrating durability-proof memory to harden the less durable PCM. Here we classify them into two types: serial (vertical) and parallel (horizontal). The serial approach simply inserts a DRAM cache backed up

serially by a PCM main memory in the memory hierarchy [16, 2]. The DRAM serves as a filter cache to capture high-locality writes. The parallel scheme [17] consists of a DRAM memory alongside with its PCM counterpart. In this scheme, the OS maintaining page-worn information is responsible for managing page migration between two types of memories.Although these previous works related to the two approaches would work well for normal applications, however, a reliable memory system must consider the worst-case scenario under malicious attacks. For the serial approach, the large DRAM cache schemes have deterministic patterns that attackers could exploit to bypass the cache [18]. Also, the approach relying on the OS becomes vulnerable as soon as the OS is compromised by an attacker.

2.4 Prior Resistance-Drift Resilient Schemes

The primary approach to alleviate the negative impact of resistance drift is to use a wide drift margin between any two adjacent levels. However, there is a trade-off for deciding the width of margins. Since the controllable range of PCM resistance is bounded by the SET and RESET states, using wider margins demands to make the valid range of each level narrower. As a result, more write-and-verify steps are required to finely tune the resistance level for the narrower valid range, which incurs the write endurance issue.

Therefore, recent works have proposed drift-tolerant techniques such as encoding information in the relative order of resistance levels in a codeword [19], using reference cells to indicate level boundaries with extra cells [20], and estimating the resistance drift based upon resistance statistical model [21].

Although these prior studies can be leveraged by error correction schemes, increasing the number of levels in a cell induces fast level shifts caused by the drift, which negatively affects PCM reliability after all. This explains that the chip design of three-bit (eight-level) cell PCM is immature while only experimental results from prototype two-bit (four-level) PCM chips have been reported in recent papers [14, 15, 19].

CHAPTER 3

SECURITY REFRESH: PROTECT PHASE-CHANGE MEMORY AGAINST MALICIOUS WEAR-OUT

As mentioned in Section 2.1, prior studies mainly focused on extending the lifetime of a PCM-based system that runs conventional applications but failed to protect the system against deliberately-crafted malicious attacks. A malicious application can exploit the properties of a durability solution to destruct a PCM portion easily. Although durability and security seem to be two separate issues in PCM design, they share a common goal and should be addressed at the same time. In this research, we argue that a correct, usable PCM design should consider the worst-case wear-out under malicious attacks such as side channel exploits to make PCM practical and commercially viable. In general, if PCM can sustain malicious attacks, they should simultaneously address the durability issue. To circumvent these intentional exploits, we must keep adversaries from inferring an actual physical PCM location. Furthermore, the address space must be shuffled *dynamically* over time to avoid useful information leaked through side-channels.

To achieve this goal, we proposed *Security Refresh*.[1] Similar to the concept of protecting charge leak from DRAM, Security Refresh, a low-cost hardware embedded inside PCM, prevents information leak by constantly migrating physical locations of PCM data (thus refresh) and obfuscating the actual data placement from users and system software.

3.1 Security Refresh

3.1.1 Security-Refresh Controller

First, we define one more address space, the *Refreshed or Remapped Memory Address (RMA)*, inside a PCM bank to dissociate a memory address (MA) from the actual data location. After receiving an access command (in MA) from the memory controller, each

[1] The original paper was published in the 37th International Symposium on Computer Architecture, Saint-Malo, France, 2010.

PCM bank re-calculates its own internal row and column address (in RMA). To allow such mapping, in this work, we propose *Security Refresh*. Similar to DRAM refresh that prevents charge leaking from a DRAM cell, our Security Refresh prevents address information leaked from PCM accesses by dynamically randomizing mapping between MAs and RMAs. On the other hands, rather than refreshing based on time in DRAM cell, our Security Refresh scheme refreshes a PCM region based on use, *i.e.*, the number of writes. Our Security Refresh is controlled by *Security Refresh Controller* (SRC), which is embedded inside the PCM bank. The SRC not only remaps an MA into an RMA but also periodically changes the mapping between these two address domains with extremely low-overhead hardware. The rationale and advantages of employing an SRC inside a PCM bank are as follows:

- To obfuscate the address information regarding the actual physical data placement from applications, the (compromised) OS, and the memory controller.
- To obfuscate potential side-channel leakage, if any.
- To prohibit any physical tampering, *e.g.*, memory bus probing.
- To allow a memory controller to exploit bank-level parallelism for better scheduling.
- To provide high efficiency without disturbing the off-chip bus during data shuffling and swapping.
- To enable a high-bandwidth data swapping mechanism without being constrained by limited, off-chip pin bandwidth.
- To allow PCM vendors to protect their product without relying on a third-party such as the OS or the memory controller.

3.1.2 The Basics of Distributed Security Refresh

Since our proposed SRC will be implemented inside each PCM bank that will likely be manufactured with a process optimized for PCM cell density, the hardware overhead for the SRC should be kept low to make it practical. Furthermore, as demonstrated previously,

information can leak through side channels. A sufficient amount of such information allows an adversary to assemble useful knowledge and devise a side-channel attack for target PCM locations. Simply hiding internal memory addresses alone will not address this issue properly. Thus, we need to constantly update the address mapping to obfuscate any relationship among information leaked from side channels.

Before explaining our algorithm, we first introduce our nomenclature in Figure 1. First of all, we treat one PCM bank as one region. As shown in Figure 1(a), one region is composed of many memory blocks (To simplify, we show only four in the figure). A memory block should be no smaller than a cache line to keep address lookup simple. For every r writes ($r = 2$ in Figure 1(b)), the SRC will "refresh" a memory block by potentially remapping it to a new PCM location using a randomly generated key. We will detail our algorithm in Section 3.1.3.[2] We call this number of writes, r, which denotes the *security refresh interval* analogous to DRAM's refresh rate. The refresh operations continue for all memory blocks in each region. A complete iteration of refreshing every single memory block in a region is called a *security refresh round*, similar to DRAM's refresh period. To begin another security refresh round, the SRC will generate a new random key and use it together with the key from its previous refresh round.

3.1.3 Security-Refresh Algorithm

Now we use an example to walk through our algorithm followed by its formal definition and description. Figure 2 depicts an example of one security refresh round. From Figure 2(a) to (e), we start from an initial state with eight successive security refreshes for eight memory blocks in one PCM region. In each sub-figure, the left column shows MAs (memory addresses) of these blocks with their data in capital letters while the right column shows the RMAs (refreshed memory addresses) and the actual data placement in PCM. We explain each sub-figure in Figure 2.

[2]We differentiate these two terms: refresh and remapping. A refresh will be evaluated upon the due of a security refresh interval, however, as we will show later, it may or may not lead to an address remapping in PCM space.

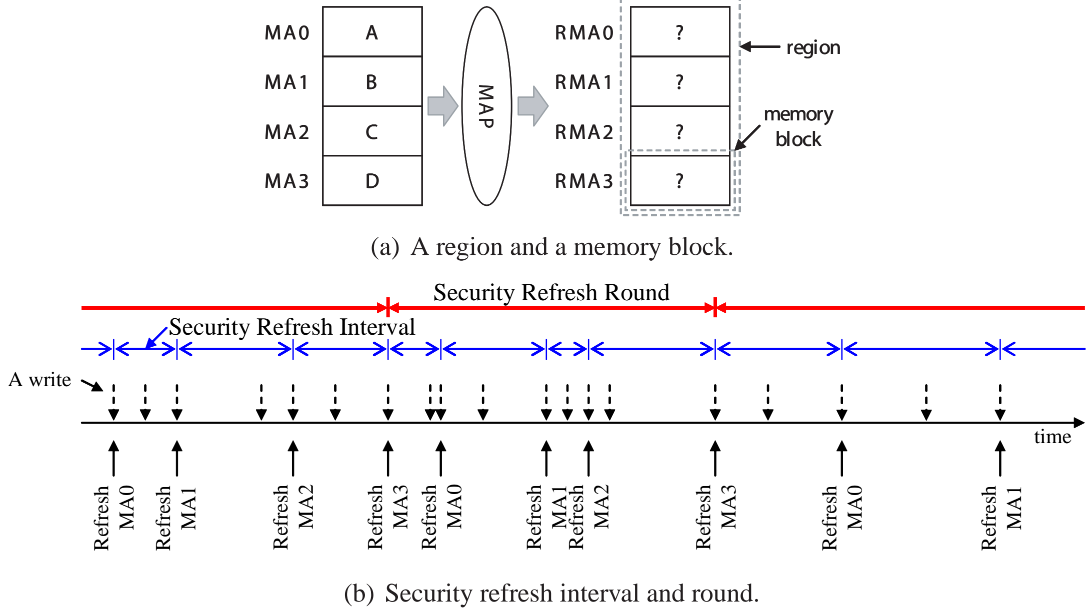
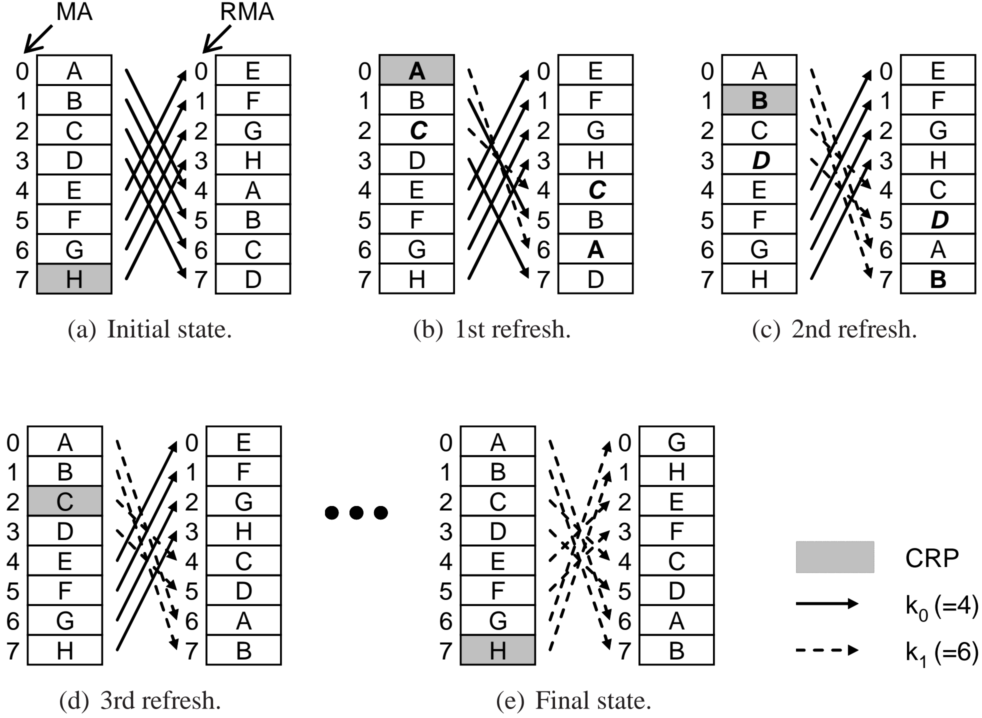


Figure 1: **Security-refresh terminology.**

1. Figure 2(a) shows the initial state in which all eight RMAs were generated by XOR-ing their corresponding MAs with a key k_0 where $k_0 = 4$. For example, the memory address MA0 (000) XOR k_0 (100) is mapped to RMA4 (100) in the physical PCM. Also note that, Figure 2(a) has reached the end of a security refresh round as all the MAs have been refreshed with k_0. Upon each security refresh, the candidate MA to be refreshed is pointed by a register called *Current Refresh Pointer (CRP)* shown as a shaded box in the figure. The CRP is incremented after each security refresh.

2. Upon the next security refresh (Figure 2(b)), a new security refresh round will be initiated because CRP has reached the first MA of a region. Consequently, a new key ($k_1 = 6$) will be generated by a hardware random number generator in the SRC for refreshing all MAs in the current round. At this point, MA0 is refreshed and remapped from RMA4 to RMA6. Since the data (A) of MA0 is now moved to RMA6 where the data (C) of MA2 used to be. Hence, C should be evicted from RMA4 and stored somewhere else. Interestingly, because of the nature of XOR, MA2 will actually be mapped to RMA4 using the new key ($2 \oplus k_1 = 4$), *i.e.,* the RMA of MA0



(a) Initial state. (b) 1st refresh. (c) 2nd refresh.

(d) 3rd refresh. (e) Final state.

Figure 2: **An example of one complete security-refresh round.**

from the previous round ($0 \oplus k_0 = 4$). This security refresh, essentially, swaps data between MA0 and MA2 in their PCM locations. We call this interesting property the *pairwise remapping property*, which will be defined and proved formally later. Note that the SRC will be responsible for reading and writing two memory blocks to physically swap the data between them.

3. Similarly, in the next security refresh (Figure 2(c)), data for MA1 and MA3 (a victim evicted by MA1) in PCM are swapped between RMA5 and RMA7.

4. In Figure 2(d), MA2 pointed by CRP is supposed to be remapped after its security refresh. However, it has been swapped previously (Figure 2(b)) in the current security refresh round. Thus, we will not swap again but simply increment the CRP pointer. To test whether an MA has already been swapped in the current round can easily be done by exploiting the pairwise remapping property. All we need to do is to XOR the current candidate MA with the key used in the prior refresh round and the key used

in the current round. If the outcome is smaller than CRP, it indicates the memory block has been swapped in the current round. For instance in Figure 2(d), we XOR MA2 with 4 (k_0) and 6 (k_1) giving a result of 0 ($2 \oplus 4 \oplus 6 = 0$). Since it is smaller than CRP (=2), it indicates that MA2 has been swapped in the current refresh round. We will show the formal proof later in this section.

5. The next five memory blocks are refreshed in the same manner. After the eighth security refresh in the current round, CRP will wrap around and reach MA0 again, completing the current security refresh round (Figure 2(e)). Upon the next refresh, a new key, k_2, will be generated and a new round starts using k_1 and k_2. k_0 will no longer be needed. Note that, for each refresh round, only the most recent two keys are needed.

Now, we formally explain the pairwise remapping property, which allows us to exchange a pair of memory blocks only with two keys. For our address remapping, assume that we use a binary operation, $\oplus$, closed on a set S, which satisfies the following properties for all x, y, and z, the elements of S where S is a set of possible addresses in a PCM region.

- Associative Property: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- Commutative Property: $x \oplus y = y \oplus x$.
- Self-Inverse Property: $x \oplus x = e$, where e is an identity element so that $x \oplus e = x$.

Basically, we find an RMA for a given MA by simply performing this binary operation between MA and a randomly generated key (k) of the same length *i.e.*, $MA \oplus k = RMA$. Here, we define several notations used in this proof as shown in Table 1.

According to associative and self-inverse properties, when A_m newly occupies A_{r_c}, B_m can be easily detected by performing $\oplus$ operation between A_{r_c} and k_p because $A_{r_c} \oplus k_p = (B_m \oplus k_p) \oplus k_p = B_m$. More interestingly, the new location (B_{r_c}) that B_m should be mapped to with k_c is the old location (A_{r_p}) that A_m used to be mapped to with k_p because $B_{r_c} = B_m \oplus k_c = (A_{r_c} \oplus k_p) \oplus k_c = ((A_m \oplus k_c) \oplus k_p) \oplus k_c = A_m \oplus k_p = A_{r_p}$. In short, we can simultaneously

Table 1: **Notations used in the proof.**

k_p	A previous key generated in the previous security refresh round
k_c	A current key generated in the current security refresh round
A_m	An MA to be refreshed in the current refresh
A_{r_p}	An RMA to which A_m was mapped with k_p (*i.e.*, $A_{r_p} = A_m \oplus k_p$)
A_{r_c}	An RMA to which A_m will be mapped with k_c (*i.e.*, $A_{r_c} = A_m \oplus k_c$)
B_m	An MA mapped to A_{r_c} with k_p, thus to be evicted by A_m
B_{r_p}	An RMA to which B_m was mapped with k_p (*i.e.*, $B_{r_p} = B_m \oplus k_p$)
B_{r_c}	An RMA to which B_m will be mapped with k_c (*i.e.*, $B_{r_c} = B_m \oplus k_c$)

map a pair of MAs into their new RMA locations by simply swapping the physical data of their old PCM blocks. Consequently, the actual swapping operations in a security refresh round will be done by one half of all security refresh operations. The simplest function that satisfies all three properties is an eXclusive-OR although we have proved that any function satisfying the above three properties can be used as the refresh/remapping function. For the rest of this chapter, we use XOR.

3.1.4 Key Selection for Address Translation

To correctly find the data location in PCM, we need to translate the given MA to its current RMA using the right key. It seems that the most straightforward way to find the right key is to add one bit in SRC for each MA to indicate whether it needs to be translated using the key in previous refresh round or the current key. Even though 1-bit per block seems small, for a 1GB PCM region with 16KB memory blocks, we will need 8KB (=2^{16} bits) extra space. In fact, hardware overhead for maintaining translation information of each block is the main reason why the prior table-based approach [4] cannot support fine-granularity segments.

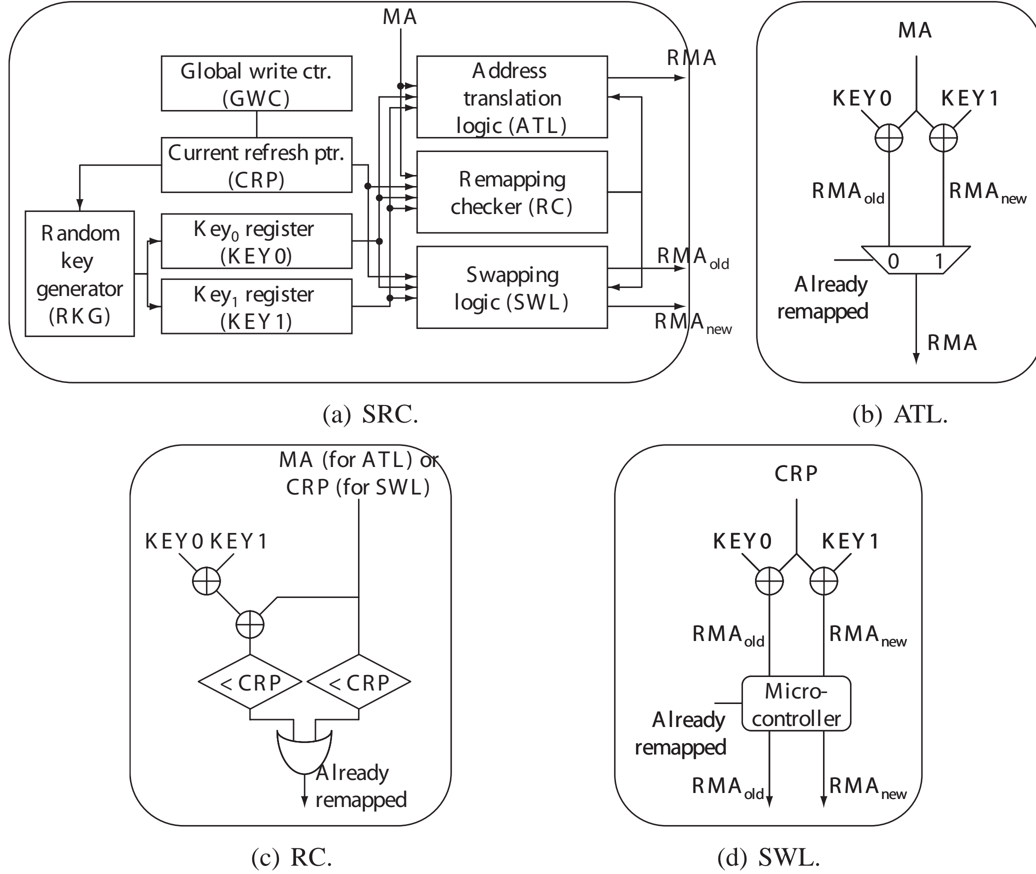
Fortunately, in our scheme, the pairwise remapping property along with the use of the linearly increasing CRP value property allows us to determine the right key without any table. In particular, when a memory controller wants to read from or write to an MA C_m, we need to use the current key (k_c) in the following two cases, otherwise, the key in previous refresh round (k_p) should be used.

- If C_m is less than the value of CRP, we should use the current key (k_c) since C_m has already been refreshed in the current security refresh round.
- If $C_m \oplus k_p \oplus k_c$ is less than the value of CRP, we should use the current key, too. This is not very intuitive, so we will describe it with a formal method. What we want to detect in this condition is whether C_m was a victim that is evicted when another MA, D_m, is remapped to the old RMA value of C_m, *i.e.*, $C_m \oplus k_p$. As explained in Section 3.1.3, we can reconstruct D_m by simply performing an XOR operation between the RMA value and the current key, which is $(C_m \oplus k_p) \oplus k_c$. If we compare D_m against the value of CRP, we can detect whether C_m was a victim that is already remapped when D_m was remapped.

3.1.5 Implementing Security-Refresh Controller

The main additional hardware for supporting Security Refresh is the Security Refresh Controller (SRC) (Figure 3(a)) per region. Each SRC consists of four registers, a random key generator (RKG), address translation logic (ATL), remapping checker (RC), swapping logic (SWL), and two swap buffers. The four registers required are: (1) KEY0 register to store a prior key ($\log_2 n$ bits where n is the number of memory blocks in a region), (2) KEY1 register to store a current key, (3) a global write counter (GWC) to count the total number of writes to a region for triggering security refresh, and (4) the current refresh pointer (CRP) that points to the next MA to be refreshed. A new key is generated by RKG in-between two security refresh rounds using thermal noise generated by undriven resistors in the SRC [22]. These keys can never be accessed or leave outside the PCM chip.

The ATL (Figure 3(b)) performs address translation. It essentially maps an MA from the memory controller to a corresponding RMA. As explained earlier, the translation process needs to understand whether a given MA has been remapped in the current round. This algorithm is implemented in the RC (Figure 3(c)), which consists of only two bitwise XOR gates, two comparators, and one OR gate. Additionally, the RC is also responsible



(a) SRC. (b) ATL.

(c) RC. (d) SWL.

Figure 3: **The block diagram of security-refresh controller (SRC).**

for finding an address to be remapped. Upon every security refresh, the RC provides the same output to the SWL (Figure 3(d)) so that SWL can decide whether the MA should be remapped or not. And if needed, the SWL performs a swap operation with a pair of swap buffers.

3.1.6 Memory-Controller Design Issues

In a conventional DRAM-based system, a memory controller understands whether a given memory request will hit in a row buffer or not. Consequently, it can schedule its commands so that the return data of those commands will not conflict in a memory bus. However, in our proposed PCM system that obfuscates internal address information, the memory controller cannot schedule the external PCM bus alone like a conventional DRAM memory controller. To utilize the bus more efficiently, we envision that future PCM chips should be

actively involved in bus arbitration. For example, a PCM chip can send a data ready signal to the memory controller once the requested data are brought into a row buffer. Based on this ready signal, the memory controller can utilize the bus more intelligently.

3.1.7 Testability

As mentioned earlier, our Security Refresh scheme is embedded inside PCM to avoid leaking useful information. However, it is also important to make the memory module testable when our scheme is applied. To suppress randomized address remapping performed by Security Refresh so the physical data locations can be determined, we can set both the key registers KEY0 and KEY1 to zero in test mode. Also, to make the access latency deterministic, the refresh asserting signal from the GWC should be masked. By doing the above, we can use existing test methods to test the memory cell array, the address decoding logic, and the data path. Lastly, a scan chain along with an isolation ring can be used to test the SRC itself. Note that this test mode must be disabled to forbid potential side-channel attacks.

3.2 Implementation Trade-Off of Security Refresh

So far, we have discussed how Security Refresh works and its advantage from the standpoint of malicious wear-out. However, there are several trade-offs in the PCM design space. For example, if the total number of writes required to start a new security refresh round is larger than the PCM write endurance limit, an adversary could wear a PCM block out before a new refresh round is triggered (**robustness**). On the other hand, extra PCM writes are induced for swapping two blocks upon remapping. Frequent swaps may unnecessarily increase the total number of PCM writes even for normal applications (**write overhead**), leading to performance degradation (**performance penalty**). Thus, we must carefully examine these design trade-offs of Security Refresh to maximize its robustness while minimizing the write overheads and its performance penalty. To quantify the trade-off, we used simple analytical models to estimate robustness and write overhead. From our analysis, we made the following observations:

1. A larger region distributes localized writes across a larger memory space.

2. A large region requires a shorter refresh interval to increase the frequency of randomized mapping changes. Otherwise, if one refresh round is too long, it may inadvertently leave a mapping unchanged for too long as well, making potential side channel attacks possible.

3. A shorter refresh interval will, nonetheless, inflict higher write overheads because of its more frequent swapping, which can lead to higher performance penalty.

Given the first observation, we first evaluated a region size as large as a PCM bank as illustrated in Figure 4. Note that the reason why we did not evaluate multiple banks in a PCM chip as a region is to allow a memory controller to exploit bank-level parallelism for better scheduling. As explained in our second and third observations, we found that the write overhead of a bank-sized region is undesirably high in this one-level scheme of Figure 4, which motivates us to investigate other techniques to mitigate them.

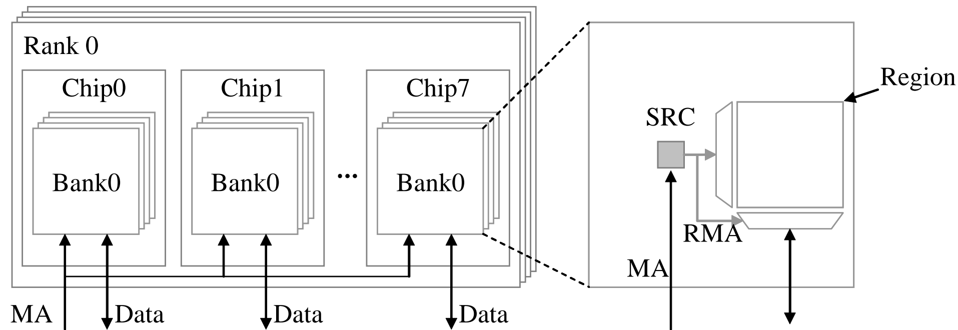


Figure 4: **One-level security refresh (four ranks, four banks per rank).**

3.3 Two-Level Security Refresh

To address the issues of write overheads and performance penalty while still taking advantage of a large region size, we propose a hierarchical, two-level Security Refresh scheme as illustrated in Figure 5. In lieu of using a very small refresh interval that increases write

overheads, we break up a region into multiple, smaller sub-regions. Each sub-region contains its own *Sub-region SRC* to perform address remapping itself based on an inner-level refresh interval. In addition, an outer-level *Region SRC* is employed to distribute writes across the entire region with its own refresh interval. The rationale behind our two-level Security Refresh scheme is that, given a refresh interval, a small sub-region effectively triggers address remapping more frequently because of a smaller number of memory blocks within each sub-region. On the other hand, an outer-level SRC occasionally remaps an MA of a given memory block across sub-regions. This additional level effectively enlarges a region size as will be detailed later.

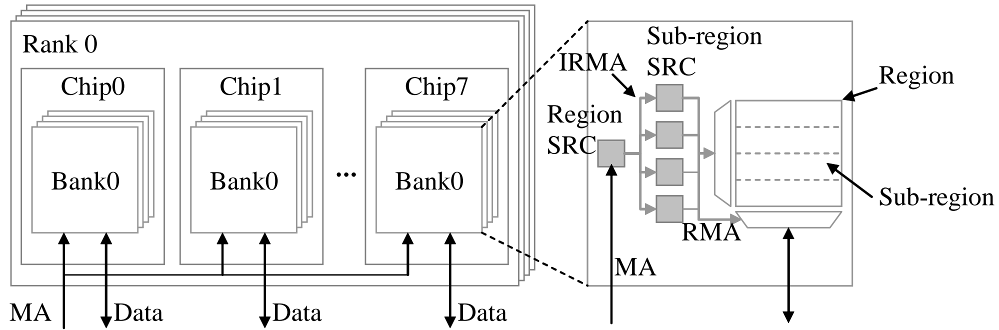


Figure 5: **Two-level security refresh (four ranks, four banks per rank).**

So far, we have laid out a logical basis for the two-level Security Refresh scheme. Now, we will explain how a security refresh of each level is performed and how it maintains the integrity of its own address remapping. Each individual Security Refresh level can be regarded as an independent layer. In other words, each level performs the Security Refresh algorithm with its own register values and settings, and the Security Refresh algorithm guarantees the integrity of the address remapping as mentioned in Section 3.1.3. Even at the same level, different regions can have different settings such as their memory block sizes and refresh intervals, though they are preset in a manufacturing phase for the maximum lifetime and the hardware feasibility.

Figure 5 depicts a block diagram of the two-level Security Refresh embedded in a PCM

bank. Basically, the two-level Security Refresh works in a recursive fashion. An outer-level Security Refresh controller (*i.e.*,, Region SRC) accepts a demand memory request from the memory controller as its input. The Region SRC remaps a memory address (MA) of the demand request to an intermediate remapped memory address (IRMA). Meanwhile, if the demand request is a write that triggers a new refresh, the Region SRC performs the demand write request and then generates a swap operation that consists of two read requests and two write requests for two IRMAs. Note that the region size of the outer-level Security Refresh is the size of a bank. Consequently, every r_o writes to a given bank (where r_o is the security refresh interval of the outer-level Security Refresh) will trigger one new refresh operation in the bank. Furthermore, to keep the integrity of its address remapping, the outer SRC should halt other requests until the swap is completed. The demand request or the swap requests generated by the outer SRC are forwarded to their own corresponding sub-regions according to a sub-region index field (Figure 6) in their IRMAs.

On the other hand, each sub-region operates the Security Refresh algorithm with its own sub-region SRC. The sub-region SRC takes a request from the Region SRC, which can be either a demand request or a swap request generated by the Region SRC. The sub-region SRC will use the IRMA of those requests to find a corresponding RMA, which is the actual physical cell location inside the sub-region. Meanwhile, if the request from the Region SRC triggers an inner-level, sub-region refresh, the sub-region SRC atomically performs a swap operation of two RMAs inside the sub-region. Consequently, every r_i writes to a given sub-region (where r_i is the security refresh interval of the inner-level sub-region Security Refresh) will trigger one new refresh operation in the sub-region. Also note that when the first write request of a swap operation from the Region SRC triggers a sub-region refresh, the second write request of the outer-level swap operation is performed after the completion of the inner-level refresh to guarantee the integrity of the address remapping in the sub-region.

Figure 6 shows an example of address remapping from MA to IRMA through the outer-level Security Refresh and that from IRMA to RMA through the inner-level Security Refresh. In this example, each 1GB bank is divided into 512 sub-regions while the memory block sizes for both region and sub-region are 256B. As shown, nine MSBs from a row address is used as a sub-region index. In other words, a row in one PCM bank is virtually partitioned into 512 sub-regions. Basically, in each sub-region, the inner-level SRC will perform the operations of Security Refresh as explained Section 3.1. Similarly, the Region SRC will perform the same operation across the entire bank. Note that the Region SRC may swap two memory blocks that belong to different sub-regions because the sub-region index is a part of output values of the XOR operation. Such swapping between distinct sub-regions triggered by Region SRC allows us to distribute localized writes across the entire bank without using a large region at the inner-level.

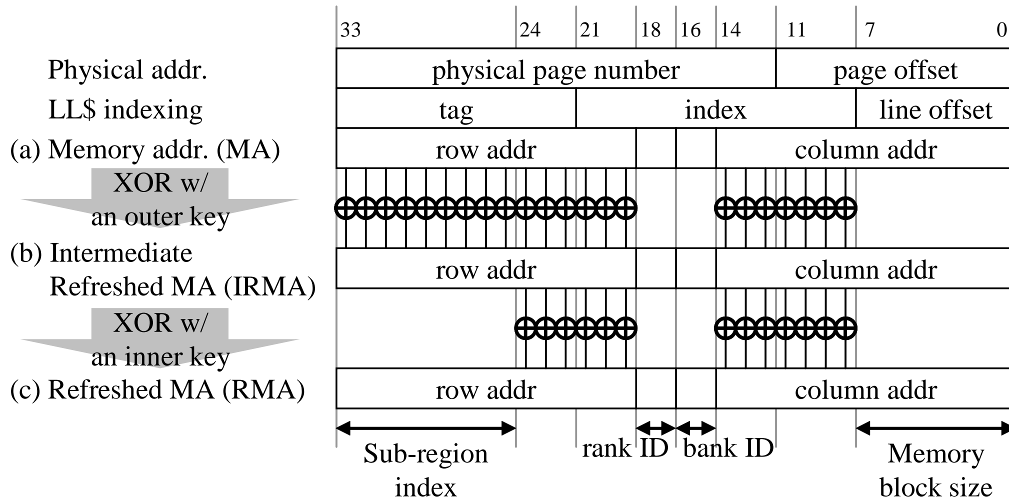


Figure 6: **Two-level security refresh within a bank.**

3.4 Evaluation

3.4.1 Robustness and Write Overhead

To evaluate the robustness, we evaluated the average lifetime for both our single-level and two-level Security Refresh mechanisms by exercising as many writes as the system can

possibly take. Birthday paradox attacks (BPA) [23] based on a randomized function efficiently fail wear-leveling schemes employing randomization with a high probability [24]. To evaluate the vulnerability of Security Refresh against BPA, we implemented our mechanisms, iteratively simulated each configuration, and calculated the average lifetime under a pinpoint attack that writes to one single logical non-cacheable address by toggling its data bits. Note that this attack method has the same effect with BPA because our Security Refresh remaps all memory addresses with a new random key for every refresh round. Throughout this subsection, we assume the same baseline architecture used in Section 2.1.

3.4.1.1 Single-Level Security Refresh

Figure 7 shows the average lifetime of the single-level Security Refresh. Here, we varied the memory block size from 256B to 8KB and the refresh interval from 1 to 128. We keep the same 1GB bank size for PCM with four banks and four ranks used in Section 2.1. The read and write latencies are 150ns and 450ns, respectively. As shown, for a given memory block size, as we refresh more frequently with a shorter refresh interval, our system is more robust. Unfortunately, such benefit comes at the cost of higher write overhead, which is calculated by $\frac{\text{the number of additional writes}}{\text{the total number of writes to PCM}}$. Note that, the extra write overheads were all accounted for when calculating the average lifetime. For example, if our refresh interval is one, the write overhead is 50%. Such additional writes can accelerate the wear-out, but we found that the additional latency caused by these additional writes effectively delays the attack as well, resulting in a longer lifetime.[3]

On the other hand, given a fixed region size, if a smaller memory block is used, we get more blocks in a region. As a result, the probability of a randomly selected block mapped to the same physical cell decreases, thus robustness is increased. However, a smaller memory block often negatively affects robustness because, given a fixed refresh interval and a fixed region size, more blocks in a region increases the required number of writes to trigger a new security refresh round. In other words, the frequency of generating a new random key is

[3]Note that our lifetime result here accounts for additional latency of performing those additional writes.

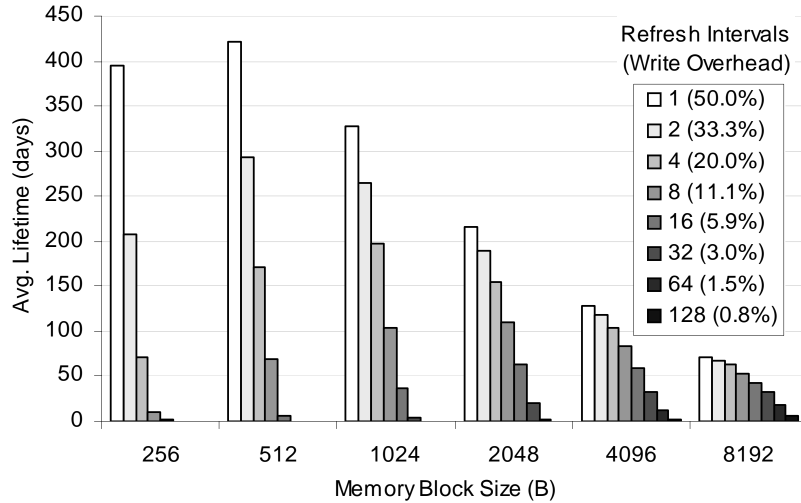


Figure 7: **Single-level robustness.**

reduced. These trade-offs are manifested in Figure 7. As shown, the average lifetime tends to increase as we reduce the memory block size down to 512B, then it decreases when we further reduce it to 256B. Note that for blocks smaller than 256B (the cache line size of the last-level cache) may require multiple PCM accesses to retrieve a single cache line, thus we did not simulate such configurations.

Overall, we found that the longest lifetime, 422 days, is achieved when we use 512B as the memory block size. This, however, may not satisfy the current average server's replacement cycle that is usually three to four years [25, 26].

3.4.1.2 Two-Level Security Refresh

Figure 8 shows the average lifetime of our two-level Security Refresh scheme when the refresh interval of an outer-level Security Refresh is 128. In this evaluation, we use the same memory block size, 256B, for both inner and outer levels. Since the last-level cache line size is 256B, it is likely that the datapath of the baseline PCM, with respect to power and performance, will be optimized for 256B as well. Furthermore, we found that the PCM with a memory block size of 256B under two-level Security Refresh demonstrated reasonably long lifetimes. Therefore, we only present results with 256B memory blocks.

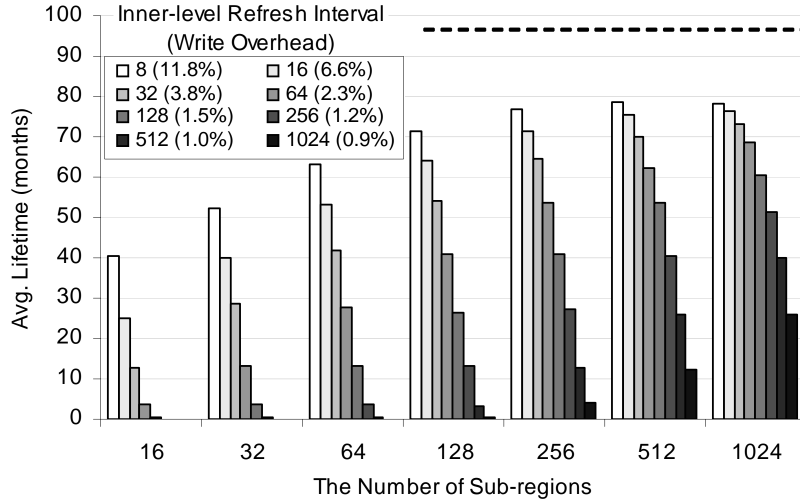


Figure 8: **Two-level robustness vs. sub-regions.**

To study the sensitivity, we varied the number of sub-regions and the inner-level refresh interval. Note that we did not simulate extremely short inner-level refresh intervals simply because they incur too much write overhead. As shown in the figure, we found that the configuration with 512 sub-regions and refreshing memory blocks every eight writes inside a sub-region can sustain around 78.8 months. This achieves 81.2% of the lifetime of the perfect wear-leveling scheme, which is 97.1 months with the same block size. It is noteworthy that this average lifetime is very pessimistic as we assume that an attacker can monopolize the entire system resources to perform a pinpoint attack continuously for 78.8 months.

Figure 9 shows the average lifetime of the two-level Security Refresh scheme with 64 or higher outer-level refresh intervals. The results suggest that the average lifetime is more sensitive to the inner-level refresh interval than the outer-level. This is explained by the following. Since a sub-region (inner level) contains fewer memory blocks, a shorter refresh interval will provide better wear-leveling.

3.4.2 Hardware Overhead

In this subsection, we describe the hardware cost of our Security Refresh. To calculate the size of registers required to implement the single-level Security Refresh, we need a detailed

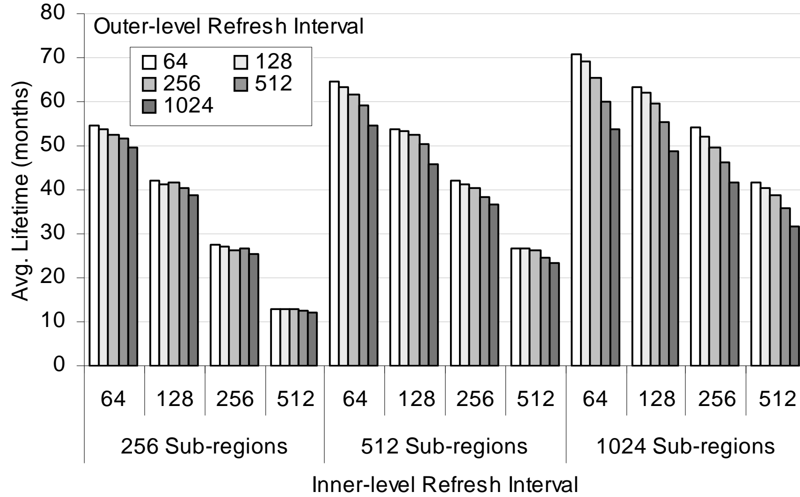


Figure 9: **Two-level robustness vs. refresh intervals.**

configuration. First, assume that a 4GB PCM rank is composed of eight PCM chips as in a conventional SDRAM DIMM while each chip consists of four banks. Then, to build a 16GB PCM system, we need 32 PCM chips. If an SRC is in charge of a PCM bank, 128 SRCs exist in the 16GB PCM system. When a memory block size is 256B and SRC's refresh rate is 64, each SRC consists of three 22-bit registers for KEY0, KEY1, and CRP, and a 6-bit register for GWC. Since eight chips are accessed in parallel to serve a 256B request, each chip has a pair of 32B swap buffers per bank. In sum, the total register size required for a chip is 292B (= $4banks \times (3 \times 22bit + 6bit + 2 \times 32Byte)$).

In case of the two-level Security Refresh, each sub-region also has a dedicated inner-level SRC. To model the area overhead, we assume the followings: 1) an outer region is divided into n sub-regions, 2) the outer region and each inner sub-region contains 2^p and 2^q memory blocks, respectively, and 3) their refresh intervals are 2^x and 2^y, respectively, then the total hardware cost per outer region without considering swap buffers can be calculated like $(x + 3 \times p) + n \times (y + 3 \times q)$ bits. On the other hand, swap buffers can be shared in the same level because a bank allows only one request to access its PCM cell array at a time, which serializes all requests. This serialization property, along with the atomicity of the inner refresh, allows all sub-regions to share physical swap buffers. That is, each level

needs one pair of swap buffers.

Figure 10 shows the hardware cost of those configurations used in Section 3.4.1.2. The hardware cost grows exponentially as the number of sub-regions increases. Thus, if more than 5 years of attack endurance is required, dividing a bank into 512 sub-regions can satisfy this requirement with around 12KB of the hardware cost. (Note that these configurations can sustain for 64.5, 63.3, and 61.5 months as indicated in Figure 9.) It is the trade-off between the cost and the high security requirement for worst-case or malicious wear-out. Unlike the conventional DRAM process, PCM fabrication process is compatible with CMOS, thus those hardware overhead will not be significant.

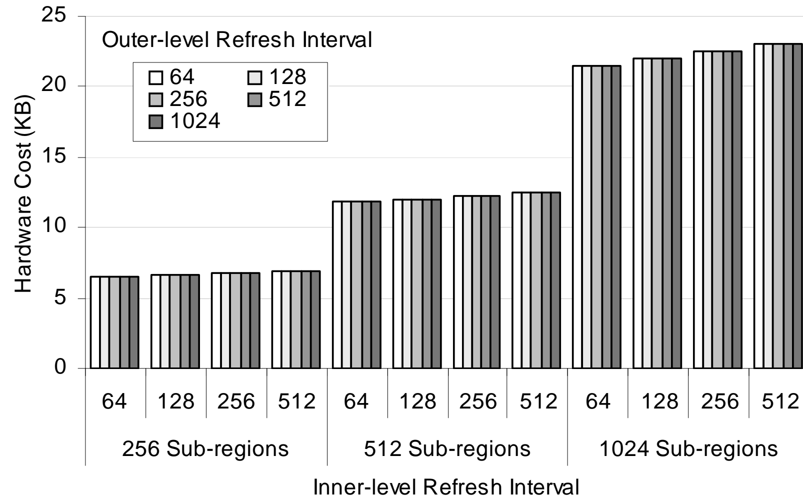


Figure 10: **Two-level hardware cost per 512MB PCM chip.**

3.4.3 Wear Leveling

In this section, we study how well writes generated by an attack are distributed across the memory space. To count the number of writes for each memory block, we use PIN [27]. In this simulation, we use the two-level Security Refresh scheme with four 1GB PCM banks, each divided into 512 subregions. Each PCM bank is one region. Furthermore, we use the same memory block size (256B) for both the region and the subregion while the refresh interval for Region SRC (outer level) is 128 writes. To study the sensitivity of inner-level

refresh intervals, we use three different inner-level refresh intervals — 32, 64, and 128 writes.

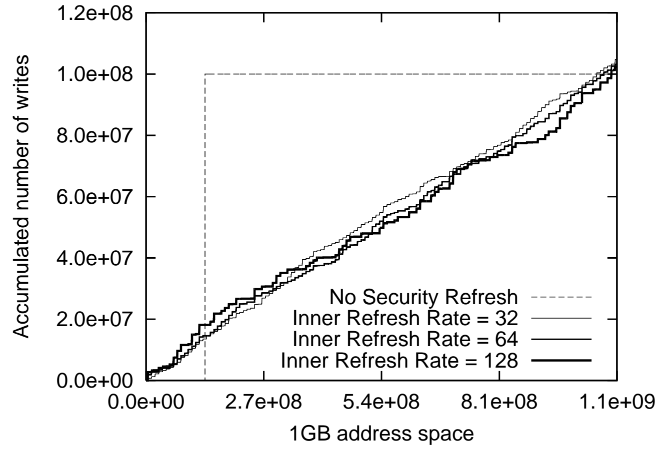
Figure 11 shows the accumulated number of writes (including swap write overhead in our scheme) for a given pinpointed physical address (134518272) for 10^8 times and 10^{11} times. The y-axis of this chart plots the *accumulated* number of writes across the memory addresses on the x-axis. To read the number of writes to a particular PCM address A, one has to obtain the values of A and (A-1) on y-axis in this chart and take a subtraction. As shown in Figure 11(a), without any wear-leveling scheme, all 10^8 writes hit the same location. With our two-level Security Refresh, these writes are distributed across the entire memory space. The more linear a curve is, the more evenly distributed the writes are. Based on this, as shown in Figure 11(a), we found that a finer-grained swap interval tends to lead to a more balanced wear-out distribution. Not surprisingly, as the number of writes is increased to 10^{11}, they are even better distributed as shown in Figure 11(b).

The figures also show how many writes are additionally generated by the swap operations during refreshes. For example, in Figure 11(a), the difference between the final accumulated number (on the right) and 10^8 tick on y-axis represents the extra writes contributed by swap operations. The percentage increase of writes for the three different inner-level refresh intervals are 3.8%, 2.3% and 1.5%, respectively.

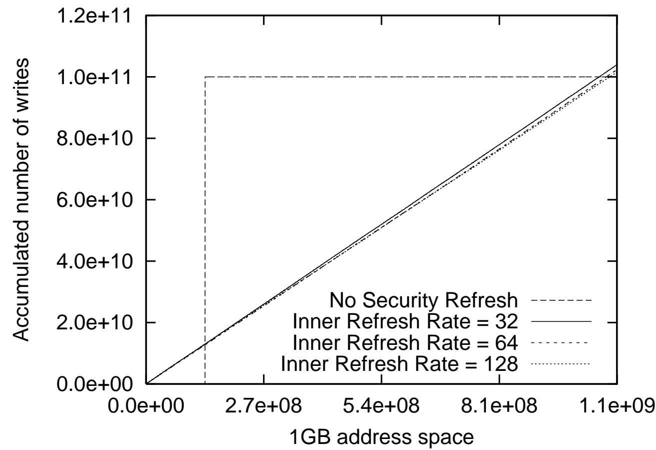
3.4.4 Performance Impact

Finally, we evaluate the performance impact of our Security Refresh scheme using SESC [28] with 26 SPEC2006 benchmark programs. Similar to previous studies [2, 9], our system employs an 8MB L3 DRAM cache for hiding PCM's relatively long read latency. Also, we modeled a memory controller that exploits bank-level parallelism and arbitrates requests to improve PCM row buffer hits. We used a two-level Security Refresh scheme with the same configuration in Section 3.4.3 to compare against a baseline without any wear-leveling technique.

As shown in Figure 12, the performance of most of the benchmark programs is barely



(a) 10^8 pinpoint attacks.



(b) 10^{11} pinpoint attacks.

Figure 11: **The accumulated number of writes over the memory space.**

affected with our Security Refresh for the three inner-level refresh intervals experimented. The two exceptional cases are 433.milc and 459.GemsFDTD, which contain not only many PCM writes but also many PCM reads. As such, the swapping operations for Security Refresh often increases the latency of the reads However, the geometric means of instruction-per-cycle (IPC) variations are found to be −1.2%, −0.7%, and −0.5% when we use 32, 64, and 128 as our inner-level refresh interval, respectively. Not surprisingly, such trend is analogous to our write overhead of those configurations, 3.8%, 2.3%, and 1.5%.

Furthermore, note that in our scheme, the nature of bitwise XOR operations allows the memory controller to utilize data locality at a row buffer. In particular, as shown in Figure 6,

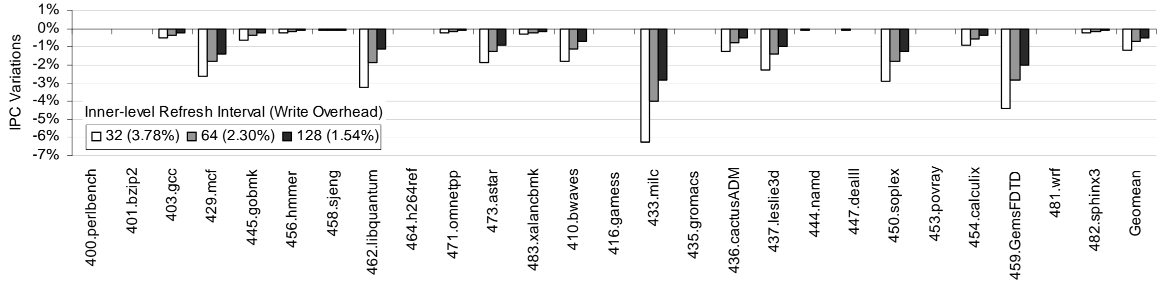


Figure 12: **Relative IPC.**

our remapping method uses a bitwise operation without shuffling address bit positions. This means that one MA row address is mapped to one RMA row address, which allows the memory controller to utilize spatial locality inside a row for better scheduling. Furthermore, the bitwise remapping allows us to send a row address of MA to a PCM chip separately from a column address of the MA similar to conventional DRAM memory commands. As a result, even though a refresh often closes a row opened by a previous demand request, our simulation results show that the row hit rates decrease by only 0.4%, 0.3%, and 0.2%, for the three inner-level refresh intervals we simulated, respectively. Overall, the performance impact with our Security Refresh scheme is negligible.

3.5 Summary

In this study, we argue that a robust PCM design must take both security and durability issues into account simultaneously. More importantly, it must be able to circumvent the scenarios of intentional, malicious attacks with the presence of a compromised OS and potential information leak from side channels. By analyzing prior durability techniques at architectural level, we demonstrated practical attacking models to wear out and fail PCM blocks. For example, prior redundant write reduction techniques do not obfuscate addresses, making a victim memory block easy to target. Some wear-leveling technique performs address randomization. However, the mapping was static at boot time, leaving open side channels for adversaries to glean and assemble useful information.

To address these shortcomings, we propose *Security Refresh*, a novel, low-cost hardware-based wear-leveling scheme that performs dynamic randomization for placing PCM data. Security Refresh relies on an embedded controller inside each PCM to prevent adversaries from tampering the bus interface or aggregating meaningful information via side channels. Furthermore, we evaluated the implementation trade-off of Security Refresh and quantified the reliability for a two-level Security Refresh mechanism. Given a 1GB PCM bank with 512 sub-regions at the inner-level, our two-level security refresh can endure more than 5 years with a 256B memory block using 128 and 64 writes for the outer- and inner-level refresh intervals. In addition, we also applied pinpoint attacks to understand the wear-out distribution using Security Refresh. We found that as the number of pinpoint writes to the same memory address is increased, our technique will distribute the data placement more uniformly, improving durability. Finally, we analyzed the performance impact of Security Refresh with normal applications (SPEC2006) and showed the average IPC degradation is below 1.2%.

CHAPTER 4

SAFER: STUCK-AT-FAULT ERROR RECOVERY FOR MEMORIES

As mentioned in Section 2.2, our objective for a new error correcting scheme is to efficiently recover the original data from permanent stuck-at faults. One of the key attributes of stuck-at faults, which prior works have overlooked, is that the cell with a stuck-at value is still readable. We exploit this property to reuse the faulty cell with the stuck-at value to provide hardware efficient multi-bit stuck-at fault error recovery. This becomes necessary because, with technology scaling of resistive memories, the non-uniform distribution of lifetime variations may be exacerbated leading to more frequent occurrences of multiple permanent stuck-at faults per data block.

4.1 SAFER: Stuck-At-Fault Error Recovery

We now describe our stuck-at-fault error recovery (SAFER)[1] technique, which enables a hardware-efficient multi-bit error recovery by dynamically partitioning the data blocks to ensure that each partition has at most one fail bit. We begin with a discussion of how to partition a data block such that each partition has at most one fail bit, and then describe how to recover from those fail bits.

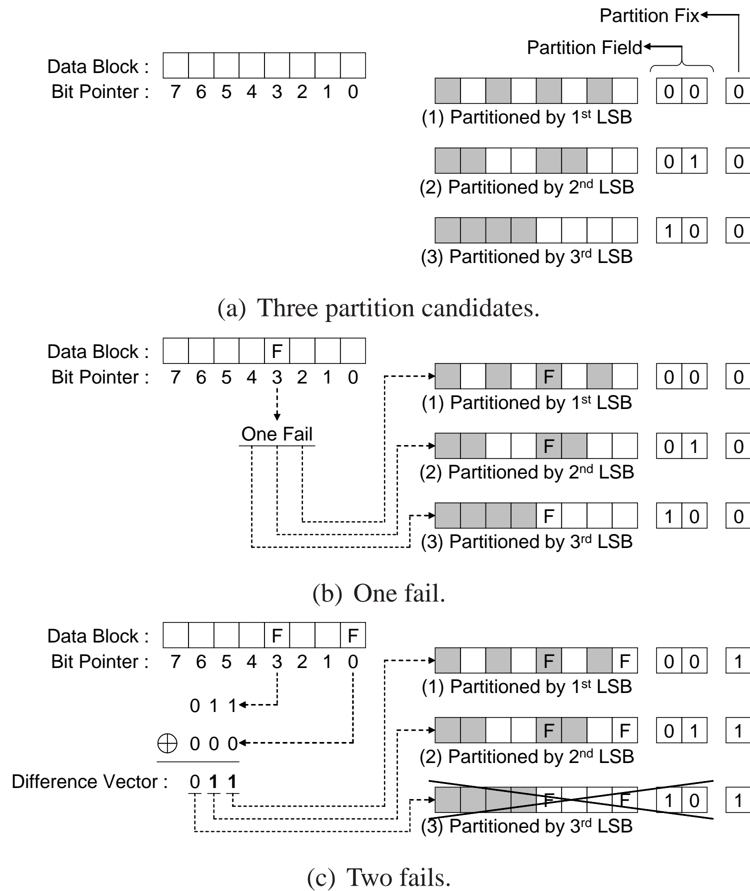
4.1.1 Partition Technique for Double Error Correction

We first explain how we partition a data block for double error correction (DEC). The key idea of SAFER for DEC is to partition a data block into two groups ensuring that the two fail bits belongs to different groups and to use single error correction (SEC) technique per group.

If we assume an n bit data block, we have $\frac{C^{n}_{n/2}}{2}$ possible ways to partition the block into two $n/2$ bit groups. However, if the goal is to only ensure that the two fail bits are not in

[1]The original paper was published in proceedings of the 43th International Symposium on Microarchitecture, December, 2010.

the same group, the number of ways to partition them into two groups is reduced to only $\lceil log_2 n \rceil$. We now describe the partition technique to handle DEC using an example shown in Figure 13.



(a) Three partition candidates.

(b) One fail.

(c) Two fails.

Figure 13: **An example of partitioning two fails.**

The partition technique of SAFER identifies the location of each data bit in a block using a bit pointer. Each data bit is assigned a bit pointer using $\lceil log_2 n \rceil$ bits. Figure 13(a) shows an example of partitioning an eight bit data block into two groups. Three bits are required to represent each bit position in this eight bit block. In this figure, each box indicates one data bit cell and gray and white boxes are used to indicate two different groups, say G and W. The data block can be partitioned into two groups in three different ways, namely, GWGWGWGW, GGWWGGWW, and GGGGWWWW, based on whether the least significant bit (LSB), the second LSB, or the most significant bit (MSB) of the

three-bit bit pointer is used, respectively. In other words, all possible 28 fail bit-pairs that are selected in the eight-bit block can be separated into two groups by using one of these three patterns.

Thus a block with at most one fail bit can be partitioned by using any arbitrary bit of a bit pointer. Figure 13(b) shows that if the first bit to fail is at bit position 3, any of the three ways of partitioning discussed above can be used.

Now, if the second bit to fail is at bit position 0 as shown in Figure 13(c), the partition should be fixed to separate the two fail bits into different groups. The partition technique uses XOR operation to determine the difference vector of the two fail pointers ($000 \oplus 011 = 011$). The number of 1s in the difference vector indicates the possible choices for partitioning the data. With two bits being 1 in the difference vector there are two ways to partition the data block. If we choose the first LSB of the difference vector, the resulting partition is shown in Figure 13(c)(1), and instead if we choose the second LSB of the difference vector, the resulting partition is shown in Figure 13(c)(2).

The "partition field" identifies which bit of the difference vector was used to partition the data block. For a n bit data block, the "partition field" uses $\lceil log_2(\lceil log_2 n \rceil) \rceil$ additional bits to identify how a block is partitioned. For our example in Figure 13, with an eight bit data block, the partition field is $\lceil log_2(\lceil log_2 8 \rceil) \rceil (= 2)$ bits with a value of either "00","01", or "10" depending on the bit position (the first, second or third LSB) of the difference vector chosen for partitioning the data. Furthermore, the partition is not fixed unless there are two fail bits. Hence, a "partition fix" bit is used to indicate whether the partition is fixed, or not. In Figure 13(b) the "partition fix" bit is set to 0, and it is set to 1 only in Figure 13(c) as soon as a second fail bit happens.

To summarize, the partition technique of SAFER identifies the two fail positions using their $\lceil log_2 n \rceil$ bit pointers. An XOR operation on the two fail pointers determines a bitwise difference vector between the two fail pointers. Finally, the technique selects a bit position with a value 1 from the difference vector, and resets all the other bits to 0. For example,

if the selected bit position is the kth LSB in the difference vector, the partition technique splits the n bit data block into two groups according to the kth LSB of the pointer for each bit inside the block. Thus, $\lceil log_2 n \rceil$ group patterns exist and only $\lceil log_2(\lceil log_2 n \rceil) \rceil$ additional bits are needed to identify how a block is partitioned.

Furthermore, for blocks with two fail bits, the partitions have to be fixed to ensure that the fail bits are in different groups. Therefore, one additional bit is required to indicate whether a partition is fixed, or not. Thus, the total storage overhead for a n bit data block is $(1 + \lceil log_2(\lceil log_2 n \rceil) \rceil)$ bits.

As shown in the above example, the partition technique of SAFER for DEC is successfully able to partition the data block such that the two fail bits are not in the same group; thereby, enabling the use of SEC per group.

4.1.2 Partition Technique for Multi-Bit Error Correction

To be able to handle more than two bit fails in a data block, the partition technique is extended to dynamically partition the data block into multiple (> 2) groups by selecting multiple bits in the difference vector. We describe the extensions of the partition technique to handle multi-bit errors using an example shown in Figure 14.

Figure 14 shows an example data block of 16 bits with four fail bits to be partitioned into four groups, which are depicted with four different gray-levels. The partition technique associates a group index for each bit in the data block using two bits from the bit pointer. When a data block is composed of 16 bits, the bit pointer is ($log_2 16 = 4$) bits, which implies that there are $C_2^4 = 6$ possible ways to choose two bits out of them. Based on the fail bit locations, one of these six possible ways is chosen to determine the four groups.

In Figure 14(a), the initial partition arbitrarily uses the third and the first LSBs. For each data bit, the concatenation of these two bits in its bit pointer represents its group index. For example, the 12th data bit has a bit pointer of "1100" and concatenating the third and the first LSBs results in a group index of "10"(2).

The "partition field" is extended to record which bit positions are used for partitioning

(a) Initial state.

(b) Dynamic partition.

Figure 14: **An example of four-group partition.**

the data. In this example, the two partition fields indicate that the third and the first LSBs are used from the bit pointer. The "partition fix" field is extended to a counter, "fixed partition counter", to keep track of the number of partitions that are fixed. In Figure 14(a), there are no fails, hence the value of the fixed partition counter is zero.

Figure 14(b) shows how a partition can be changed dynamically to account for a new fail bit. If the first fail bit occurs at bit position 8, the initial partition is still valid because none of groups has more than one fail bit. Now, if the second fail occurs at bit position 2, there are two fail bits in group 0. Thus, a new partition should be derived so that the two

fail bits are in different groups. Using the partition technique described in Section 4.1.1, the difference vector of the two fails is ($1000 \oplus 0010 = 1010$), which implies that the second and the fourth LSBs are candidates for the first partition field. Correspondingly, the first partition field is set to “11”. The fixed partition counter increases by one to account for fixing the first partition field. After this partition, groups 0 and 2 each have one fail bit. Note that even if the second fail bit was not located in group 0, the fixed partition counter would have to be incremented although the first partition field does not need to be changed.

At this point, if a third fail happens, the second partition field should be fixed with a proper value. If the third bit fails in position 0, the current partition has two fail bits in group 0. Applying the same partition technique as above, the difference vector of the two fails is ($0010 \oplus 0000 = 0010$), which implies that the second LSB position is the candidate for the second partition field. After this re-partitioning, groups 0, 1, and 2 each have one fail bit. Furthermore, the fixed partition counter is incremented and reaches its maximum value of two, making it impossible to re-partition further. Thus, in this example, we can recover from a fourth bit failure only if the failure occurs in bits belonging to group 3.

Based on the above discussion, it is clear that the hardware requirement is proportional to the number of groups required to partition the data to ensure one fail bit per group. For a n bit data block and a k group partition, the number of additional bits required is $\lceil log_2 k \rceil \times \lceil log_2 \lceil log_2 n \rceil \rceil + \lceil log_2(\lceil log_2 k \rceil + 1) \rceil$, where, $\lceil log_2 k \rceil$ is the number of partition fields, $\lceil log_2 \lceil log_2 n \rceil \rceil$ is the size of each partition field, and $\lceil log_2(\lceil log_2 k \rceil + 1) \rceil$ is the size of the fixed partition counter. For 512 bits of data block to be partitioned into 32 groups, additional 23 bits are required to represent the partition, which is still only 4.50% overhead compared to the data size.

4.1.3 Using Data-Block Inversion

The partitioning technique described in the previous section exploited the fact that stuck-at faults are permanent (not transient) to ensure that at most one stuck-at fault bit is present in each partition. In this section, we propose a recovery scheme by exploiting the fact that

if a resistive memory's cell wears out resulting in a stuck-at fault, then it is still possible to read the cell content as the permanent stuck-at value. By exploiting this readability of failed cells, the recovery scheme reduces the number of additional bits required to recover data written to a stuck-at cell.

If a data block has only one fail bit and the data being written at the fail bit position is the opposite of the stuck-at value, then the data can be stored in an inverted form with a marked flip-bit. When reading the data, the original data can be recovered by inverting the stored data if the corresponding flip-bit is marked. The idea of inverting a data block is similar to bus-inverting coding [29] and Flip-N-Write [6]. However, our objective in inverting a data block is to recover a stuck-at fail while the bus-inverting coding [29] inverts a data block to reduce I/O power and the Flip-N-Write [6] utilizes it for removing redundant writes to PCM.

The proposed technique to invert the data can be used by SAFER only after verifying that the data write has failed to store the intended value. This write verification can be performed by reading the data written and comparing it with the original data. When the verification fails, the positions of fails and its stuck value can be revealed from the comparison result. Note that iterative write techniques that require a write verification phase are already needed for resistive memories using multi-level cells [30].

The proposed data inversion technique uses only one additional bit per partition to indicate that the data value has to be inverted prior to a read. However, the drawback is that the decision to invert and store the data can be made only after a first write fails the verification, resulting in two writes to store the data, thereby affecting the endurance of the cell.

To alleviate this problem, we propose a relatively small direct-mapped cache called "fail cache", to keep track of data blocks with recent stuck-at fails. For these blocks recent fail positions and their stuck-at values are maintained in the cache. Figure 15 shows

the fail cache organization that is composed of 16 banks. When storing new fail information, its block address and fail pointer are used to calculate the corresponding cache entry by separating into a cache tag, an index and a bank address. If new fail information is detected during the write verification phase, the fail position and the stuck-at value are known. Therefore, they can be stored with its tag portion in the corresponding cache entry. The tag, the cache index, and the bank address can also be calculated from its block address and the fail pointer. On every write request from the memory controller, all fail information for the corresponding n-bit data block should be extracted from the fail cache. To do so, the 16 banks are simultaneously accessed for $n/16$ iterations. For example, 32 iterations are required for a 512-bit data block. As a result, two n-bit vectors are generated – a fail indication vector and a stuck-at value vector. These two vectors for each write request can be exploited to avoid the additional write. If a fail indication vector indicates errors, the corresponding bits to be written are suitably inverted and stored according to their stuck-at values and partition information. Note that a read request to the same data block precedes a write request to eliminate redundant writes, and the partition information is collected during the read. Thus, if all fail information for a write data block is found in fail cache, the second write can be avoided. Also, since the preceding read can be used to gain enough time to access fail cache for $n/16$ iterations, the performance impact of the fail cache will be insignificant.

4.1.4 Putting It All Together

SAFER comprises two techniques, namely, the dynamic multi-group partition and the data block inversion. The dynamic multi-group partition ensures that each group includes at most one fail bit by partitioning the data into different groups. With each group now including at most one failed cell, the data block inversion scheme can be applied to recover from the stuck-at fault for that group. The total hardware bit budget of SAFER, to recover from a maximum of k failures, is $\lceil log_2 k \rceil \times \lceil log_2 \lceil log_2 n \rceil \rceil + \lceil log_2 (\lceil log_2 k \rceil + 1) \rceil + k$, where n is the size of a data block and k is the number of partitioned groups.

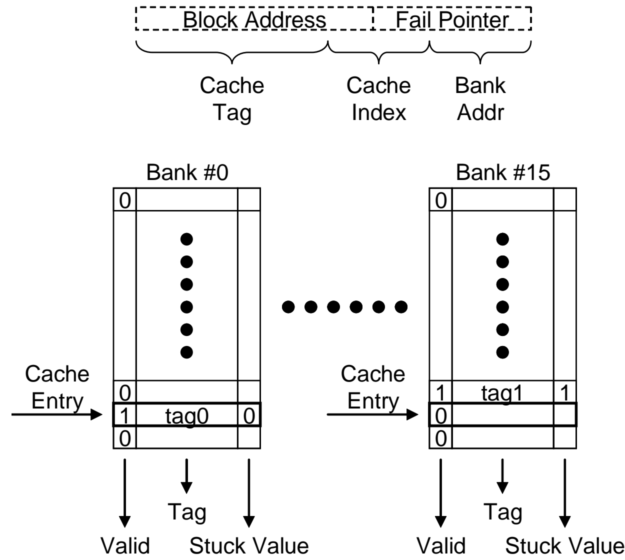


Figure 15: **Fail-cache organization.**

Another hardware overhead is bit manipulation logic for data block partition and data inversion. As our partition technique is based on the knowledge of fail positions, detecting a new fail position in a write verification phase is important. It can be implemented with simple combinational logic, an n-to-$\lceil log_2 n \rceil$ priority encoder for each partition group. If a priority encoder generates a valid fail pointer in the first verification phase, the corresponding group will be re-written in an inverted form. If a priority encoder still generates a valid fail pointer after the inversion write, it indicates the occurrence of a new fault in the corresponding group. Then, the data block is re-partitioned with the two fail pointers revealed at the two verification phases. That is, re-partition can be performed with the priority encoders and a simple FSM described in Figure 16. For both read and write data inversion, a partition decoder is required to select corresponding bits to be inverted.

Figure 17 shows an example of SAFER for a 16 bit data block and a four group partition. Additional six bits are required for the four-group partition and four flip bits are used to indicate whether the data in the corresponding groups is stored in an inverted form or not. Note that the six bits used to describe the partition are updated only when a new fail bit occurs. On the other hand, the four flip bits will be updated on every write that tries to store

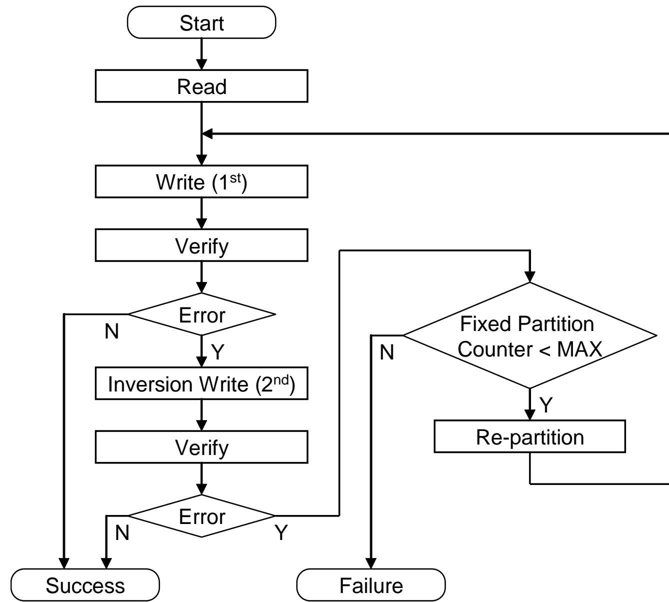


Figure 16: **The sequence of a write request in SAFER.**

to the fail bit a value that is the opposite of the stuck-at value. In this example, fail bits are present in group 0, group 1, and group 2. Thus, the flip bits for those groups may be changed on every write. However, the flip bit for group 3 will still be zero until a new fail happens in group 3.

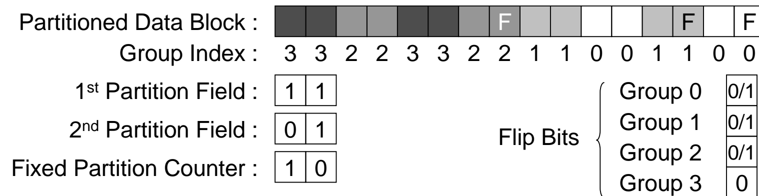


Figure 17: **An example of SAFER.**

4.2 Efficient Implementation of SAFER

In this section, we address three key issues necessary for efficient implementation and use of SAFER, namely, where to place SAFER logic, what is the ideal data block size to maximize SAFER effectiveness, and how to limit the overhead of the "fail cache".

4.2.1 The Location of SAFER Logic

Fail recovery schemes are mainly used to prolong the lifetime of resistive memory especially after fails occur, but they are not geared towards decreasing the number of writes to improve the lifetime. Hence, fail recovery schemes must be used in concert with other schemes that delay the occurrences of failures, such as redundant write reduction schemes [1, 2, 5, 4, 6] and wear-leveling schemes [4, 9, 18]. These schemes are typically implemented in the memory controller or in the memory chip itself. For example, the wear-leveling schemes maintain their own address translation layer to evenly wear out the entire memory space, and our *Security Refresh* logic described in Chapter 3 is located inside the memory chip to protect against malicious attacks. To use fail recovery schemes in conjunction with these other schemes, it is necessary that they be embedded in the memory chips. Thus, we propose to locate the SAFER logic inside the memory chip.

4.2.2 Ideal Data Size for SAFER Effectiveness

SAFER dynamically partitions a data block into multiple groups according to fail locations and supports one bit correction for each group. Therefore, the larger the data block, the more efficient the fail recovery. For example, a double error correction per 16 bytes is more efficient than a single error correction per eight bytes. Similarly, four bit error correction per 32 bytes is more efficient than the two bit error correction. However, the upper bound of the size of a data block will be decided by the memory chip design, which is optimized to increase the density of the memory cell.

Figure 18 shows an example of a typical 4Gb 8 bank DDR3 DRAM architecture that is highly optimized for density. We expect the new resistive memory architecture to be similar to that of the DRAM because of the density issue. In the example, each bank is composed of 2048 sub-arrays whose size is 512×512 bits [31, 32, 33].

Here, the column decoder generates column selection signals to sub-arrays, and pass-transistors, which act as column multiplexers, are located near by each sub-array. This is important so as to minimize area for long wires from the sense amplifiers to interface

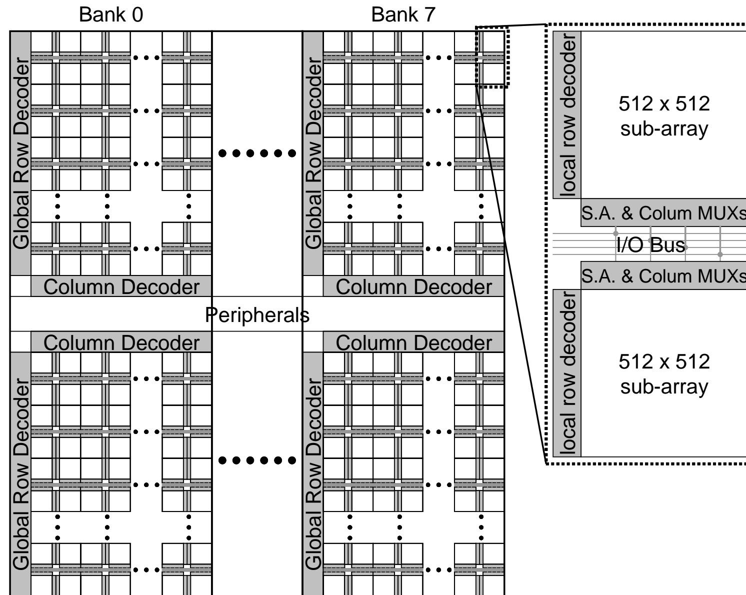


Figure 18: **DRAM architecture.**

peripherals by multiplexing them. Thus, the size of data bits that are transferred to interface peripherals at any time is equal to the minimum burst length that the chip supports. For instance, the DDR3 interface has a fixed burst length of eight. If the I/O data bus width is 16, 128 bits can reach to the peripherals.

To minimize the area overhead SAFER is best located in the peripherals, which implies that the size of a data block can be at most 128 bits in this case. However, historically the interface size continues on an upward trend from SDR to DDR, to DDR2, and to DDR3. Thus, we can safely assume that the size of data reaching the peripherals may be 512 bits in the near future. We evaluate the effectiveness of SAFER varying the block size from 64 to 512 bits in Section 4.3.

4.2.3 The Area Overhead of Fail Cache

Since we decided to embed SAFER logic inside the memory chip, the fail cache should be located inside the memory chip with other peripherals. Fortunately, a PCM process is CMOS compatible, and it poses no process technology hurdles to implementing an SRAM cache. Hence, one of the major concerns is the area overhead of the "fail cache". According

to ITRS projection [34], the cell sizes of SRAM and PCM, in 2024 will be $140F^2$ at 10 nm and $6F^2$ at 8 nm, respectively, implying that 36.46 times cell area difference may exist between SRAM and PCM. Table 2 shows the area overheads of a direct-mapped SRAM “fail cache” considering the 36.46 times cell area difference. For example, if we assume an 8Gbit PCM chip with a fail cache with 128K entries, in which each entry is composed of 16 bits of tag, a valid bit and a stuck-at value, then the total size of the cache is 2.25M bits, which is only about 1.00% area overhead relative to the 8Gbit PCM. In Section 4.4, we show the effectiveness of the “fail cache” varying the number of entries from 1K to 128K.

Table 2: **SRAM fail cache overhead for an 8Gb PCM chip.**

Number of Entries	Tag Size (bits)	Entry Size (bits)	Cache Size (bits)	Area Overhead
1K	23	25	25.6K	0.01%
2K	22	24	49.2K	0.02%
4K	21	23	94.2K	0.04%
8K	20	22	0.18M	0.08%
16K	19	21	0.33M	0.15%
32K	18	20	0.63M	0.28%
64K	17	19	1.19M	0.53%
128K	16	18	2.25M	1.00%

4.3 Methodology

In this section, we present the methodology for evaluating SAFER and for comparing it against two existing techniques, namely the ideal *Hamming Coding* [35] and the *ECP* [10] technique. We compare against *Hamming Coding* because it represents a theoretical limit of memory lifetime for existing ECC schemes designed to correct transient errors. The number of bits required for the Hamming Coding implementation is provided by the Hamming Bound: $l \leq n - \lceil log_2 \Sigma_{k=0}^{t} C_k^n \rceil$, where l is the size of data, n is the size of the hamming code including meta-data for correction, and t is the number of correctable bits [36]. For example, a 512 bit data block needs 58 additional bits to be able to correct eight fails. Again, these 58 bits may serve only as a lower bound and a practical implementation may

require more bits. In addition, Hamming Coding has a high toggle rate for the meta-data. Hence, an additional bit is needed to determine if the meta-data is valid. The indication bit also helps avoid cells for meta-data from failing earlier than data cells. In our evaluation, the Hamming Coding scheme is referred to as *IdealECC*.

Since our focus is to implement SAFER inside the memory chip, limiting the area overhead is important. We define area overhead as $\frac{\text{the size of meta-data}}{\text{the size of data}}$. For instance, the area overhead of the (72,64) hamming code is 12.5% (= $\frac{72-64}{64}$). For comparison, we use the area overhead of the (72,64) hamming code as the upper bound for our evaluation and exclude all configurations of SAFER, *ECP* and *IdealECC* that exceed this area overhead.

Figure 19 shows the hardware overheads for the different configurations for *IdealECC*, *ECP*, and SAFER. The configuration names for each of the techniques include the maximum number of fails that can be recovered. The number above each bar in the graph shows the size of meta-data for the corresponding configuration. For example, for the 512 bit data block, ECP6 represents the ECP technique with six fail pointers that can recover up to six fails, and uses 61 bits for the meta-data; IdealECC8 represents the ideal eight bit Hamming Code correction technique, which requires a minimum of 59 bits; and the SAFER32 can correct up to 32 fails with an additional 55 bits.

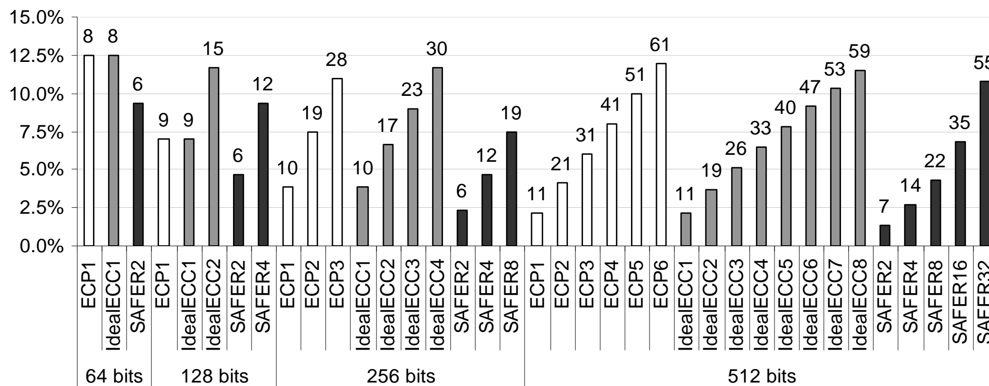


Figure 19: **Hardware overhead for recovery schemes.**

4.3.1 Experimental Setup

We use Monte Carlo simulations to evaluate SAFER and compare against *IdealECC* and *ECP*. Since PCM is the closest to mass production among resistive memories, our evaluations are based on ITRS projections for PCM endurance. We use the following assumptions for the Monte Carlo simulations:

1. We assume the lifetime of each memory cell to follow the normal distribution with a mean lifetime (μ) of 10^8 and without any correlation between neighboring cells [11]. Our experiments with different standard deviation (σ) values (10^7, $2 \cdot 10^7$, and $3 \cdot 10^7$) did not show significant variation in lifetime patterns. Hence, we use a standard deviation (σ) of 10^7 for our evaluations.

2. We assume a perfect wear-leveling scheme so as to focus only on the impact of the fail recovery scheme on the lifetime. The wear-leveling scheme evenly wears out the entire memory space at a block granularity equal to the line size of the last level cache as in the *Randomized Region-based Start-Gap* [9] and the *Security Refresh* [18]. We use 256 bytes for the last level cache line size, which implies that all the 256 byte memory blocks have the same number of writes because of the perfect wear-leveling scheme. Based on this, we measure the lifetime of one 256 byte data block.

3. A write request to memory is converted to a sequence of a read, a write, and a read request. The first read eliminates *silent writes* to memory by comparing the memory data read with the data to be written. We assume that 50% of the writes are *silent writes*. The second read verifies that the data written to memory matches the intended write data, which allows us to recognize cell failures. SAFER requires another write with necessary bit inversions if cell failures are detected during write verification. However, a hit in the fail cache will avoid the second write because the bits are already suitably inverted to account for the cell failures based on the information stored in the fail cache.

4. We assume that four x16 memory chips compose a x64 DIMM memory module so that each chip can deliver 512 bits of data.

In the Monte Carlo simulation, each configuration is run 50000 times, and the average result is reported. For each run, our simulator allocates the array equivalent to the number of required cells including the 256 byte data block and its meta-data corresponding to each configuration. A random write endurance value according to the aforementioned normal distribution is assigned to each array element. For each write to a cell, we considered the toggling rate of the value to determine the available lifetime. Simulation continues until a given configuration cannot recover from a failure any longer. We take into consideration that all the meta-data do not have the same toggling rate. For example, the fail pointer in the ECP scheme and the partition fields in SAFER are updated only once when a new fail occurs. On the other hand, the meta-data for Hamming Coding (excluding the bit indicating the validity of the meta-data), the replacement cells in the ECP scheme, and the flip bits in SAFER are written with the same toggling rate (i.e., 0.5) as the data. For SAFER, the simulation also accounts for an additional write that is needed if the write verification detects a failure.

4.4 Results

In this section, we describe the simulation results focusing on the following figures of merit: lifetime improvement resulting from fail recovery, number of fails recovered for a given size of data block, and the cost of meta-bits for the observed lifetime improvement. Finally, we show the effectiveness of the fail cache in eliminating the additional writes and correspondingly improving the lifetime.

4.4.1 Lifetime Improvement

Our simulations assumed that the lifetime of memory cells follows a normal distribution $N(\mu, \sigma)$, where μ is 10^8 writes and σ is 10^7 writes. Furthermore, we assumed that each bit toggles with a probability $T = 0.5$. However, for reliable analysis, we present the lifetime

improvement as a function of σ.

Figure 20 describes the method used to determine the relative lifetime improvement. In the example shown in Figure 20, the first fail shown as F occurs at 131.1 million writes. If SAFER were to increase the lifetime to L, then the relative lifetime improvement is calculated as $(L - F)T/\sigma$ to account for the dependence of the observed lifetime improvement on both σ and T. If SAFER were to increase the lifetime to the mean lifetime, then the relative lifetime improvement is $((2 \cdot 10^8 - 1.311 \cdot 10^8) \cdot 0.5/10^7) = 3.44$.

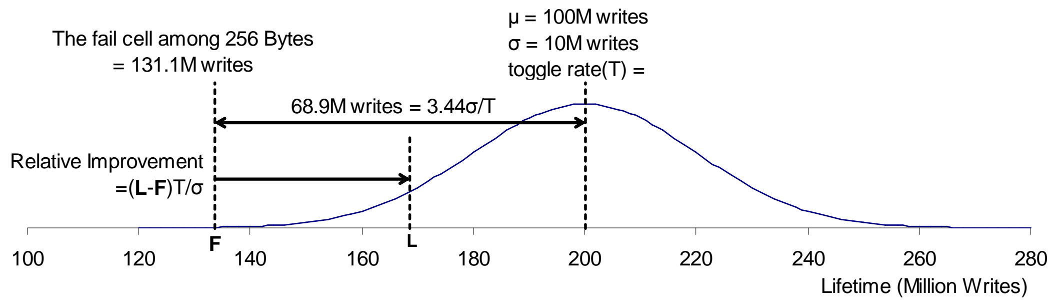


Figure 20: **The definition of lifetime improvement.**

Figure 21 shows the relative lifetime improvement for each configuration with different data block sizes. For these results, SAFER does not use the fail cache thereby requiring the additional overhead of a second write if the write verification detects a failed cell. We observe that, even without the fail cache, SAFER improves the lifetime more than ECP for all the configurations. For a 512 bit data block size, SAFER32 increases lifetime by 21.6 million ($= 1.08 \cdot 10^7/0.5$) writes, and ECP increases lifetime by only 21.1 million ($= 1.05 \cdot 10^7/0.5$) writes while still using 10% more meta-data (Figure 19) than SAFER.

Also, each bar of the IdealECCn represents the lifetime improvement at the time of occurrence of the $(n + 1)^{th}$ fail. For example, for a 512 bit data block, the lifetime improvement by using IdealECC2 is 12.8 ($= 0.64 \cdot 10^7/0.5$) million writes when the third failure occurs.

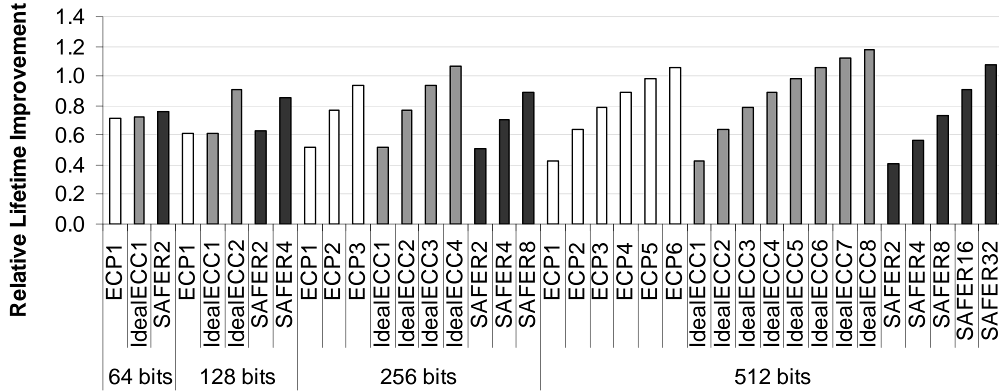


Figure 21: **The relative lifetime of a 256B memory block.**

4.4.2 The Number of Fails Recovered

Figure 22 shows the average number of fails recovered per memory block for each configuration. It appears that ECP and IdealECC show linear increment in fails recovered with increase in the maximum number of recoverable fails. On the other hand, SAFER shows an exponential improvement. It is important to note that the maximum number of recoverable fails for SAFER increases exponentially as the number of partition fields increases linearly.

As shown in Figure 22, for a 512 bit data block, SAFER32 recovers from 22.94 fails whereas ECP6 recovers from only 17.08 fails. However, the relative improvement in lifetime with SAFER is only 2% better than the improvement with ECP. The key reason why the 34% improvement in the fail recovery of SAFER is not translated to larger improvement in lifetime (compared to ECP) is the additional write required by SAFER if the write verification phase identifies a failed cell when we do not use a fail cache. We show in Section 4.4.4 that using a fail cache with SAFER significantly removes the additional writes and shows gains in lifetime improvement even relative to IdealECC8.

4.4.3 Meta-Bit Overhead vs. Lifetime Improvement

Another important figure of merit of a recovery technique is the cost of meta-data for the observed lifetime improvement. Figure 23 shows the contribution of each meta-data bit to the overall lifetime improvement for a memory block size of 256 bytes. From Figure 23,

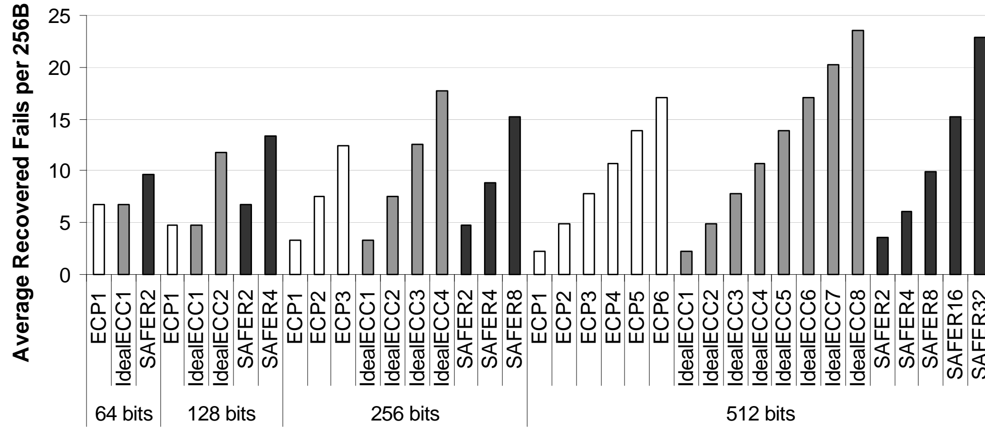


Figure 22: **Fail recovery in a 256B memory block.**

we observe that, for a data block of 512 bits, SAFER32 has a 13.4% better utilization of the additional meta-data relative to ECP6.

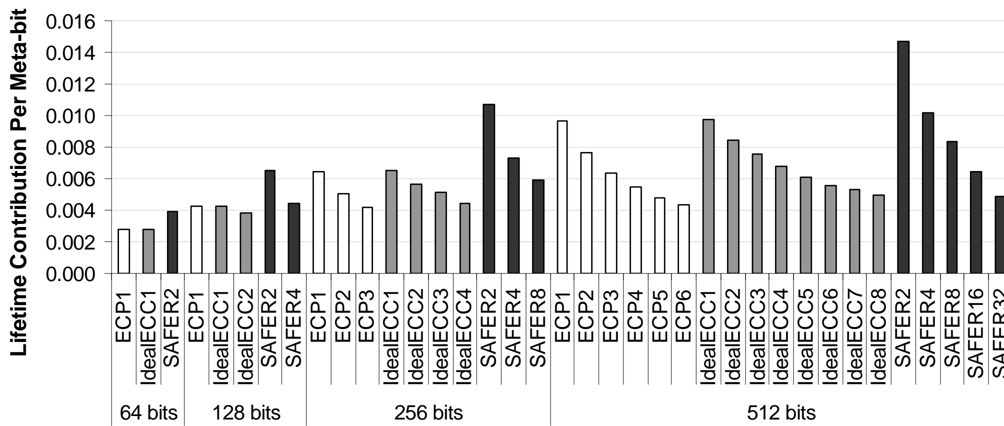


Figure 23: **Meta-bit contribution for lifetime.**

4.4.4 SAFER with Fail Cache

So far, we have evaluated lifetime improvement and meta-bit efficiency of SAFER without fail cache. By using the fail cache, however, the lifetime can be extended even longer. Fail cache enables SAFER to avoid the additional write by providing information about the fail bits so that the data to be written can be suitably inverted. We use the miss rate of the fail cache as a measure of its effectiveness in reducing the additional write to the memory.

To determine the fail cache miss rate to enable *n* bits of data to recover from a maximum

of k fails, we randomly set the fail bits in a memory of size 1GB, such that each n bits of memory had at most k failures. As soon as any n bits of data block has more than k fails, the fail insertion was terminated.

Using this set-up, we simulated 26 applications from SPEC2006 suite using the PIN instrumentation tool [27]. The following memory hierarchy was simulated: 32KB 8-way set-associative L1 data cache, 1MB 8-way set-associative unified L2 cache, and 8MB 8-way set-associative L3 DRAM cache, and finally a 1GB main memory. Out of 26 applications, we only used ten applications that have more than one million writebacks to the memory (Table 3). In this set of simulations, we simulated five billion instructions.

Table 3: **Applications with more than 1M writebacks to memory.**

Application	Number of Writebacks
410.bwaves	3.92M
429.mcf	8.17M
433.milc	7.72M
436.cactusADM	1.29M
437.leslie3d	4.75M
450.soplex	3.87M
458.sjeng	1.13M
459.GemsFDTD	9.37M
462.libquantum	7.62M
473.astar	2.45M

The geometric mean miss rate of the above applications are shown in Figure 24 for different cache sizes for different maximum recoverable fails in a 512 bit data block. Note that different bars represent fail caches with different numbers of entries. From Figure 24, we observe that cache miss rate not only increases as we decrease the cache size, but also increases substantially as we increase the number of maximum recoverable fails. However, as the number of recoverable fails increase, the contribution to lifetime improvement by each additional bit continues to decrease. For example, from Figure 21, we observe that, for 512 bit data block, IdealECC2 achieves 54.6% of the relative lifetime improvement of IdealEEC8 by correcting up to only two errors. From Figure 24, we observe that, to correct

up to two errors, the fail cache miss rate is only 5%.

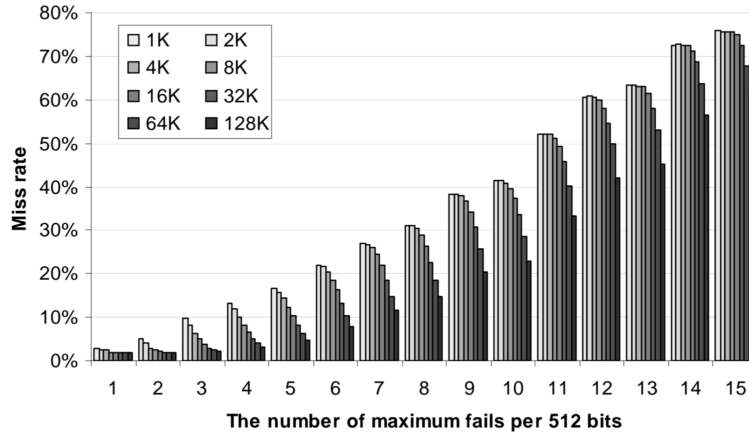


Figure 24: **Fail-cache miss rate.**

Figure 25 depicts the relative lifetime improvement when we use a fail cache. Based on the above discussion, we observe that even a small fail cache with 1K entries has comparable lifetime improvement as a 128K entries cache. Furthermore, we observe that SAFER32 has better lifetime improvement relative to even IdealECC8 with just a fail cache of 1K entries.

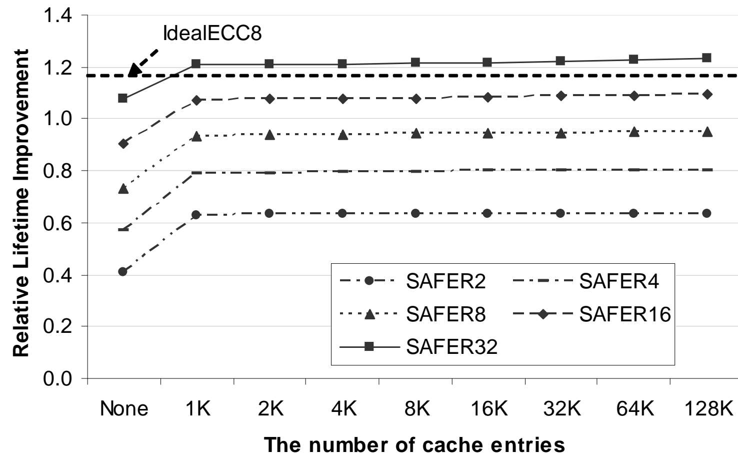


Figure 25: **The relative lifetime improvement of SAFER with a fail cache.**

4.5 Summary

Existing ECC mechanisms are geared towards correcting transient errors in DRAM memories and are not suitable to correct permanent stuck-at faults. As the cells continue to age, permanent stuck-at faults increase because of wear-out. The aging rate is particularly severe for several emerging non-volatile memory technologies. Furthermore, with process technology scaling, the lifetime variation of the cells increase, which leads to early multiple cell failures. We proposed and evaluated SAFER, a stuck-at fault error recovery technique for memories, which efficiently recovers from multiple stuck-at faults and which works in conjunction with existing wear-leveling techniques.

SAFER handles the growing stuck-at-fault errors by dynamically partitioning a data block into multiple groups and by ensuring that each group has at most one failed cell. SAFER reduces hardware overhead by exploiting the property that failed cells with a stuck-at value are still readable and uses the failed cell to continue to store data. Our evaluation based on phase-change memories shows that SAFER has 11.91% and 11.52% better hardware efficiency relative to ECP and ideal hamming coding schemes, respectively. Furthermore, SAFER achieves 14.75% and 3.07% better lifetime improvement relative to ECP and ideal hamming coding scheme, respectively.

CHAPTER 5

WRITE-FREQUENCY REDUCTION METHODS

In this study, we concentrate on a hybrid-memory design for filtering out frequent writes from resistive memories typically having limited write endurance, *e.g.,* , phase-change memory (PCM). We proposed a new hybrid-PCM architecture using low-cost hardware for effective wear-out management. The proposed architecture integrates a small, durability-proof, static random-access memory (SRAM) to filter out the frequently written addresses. To do so, we propose a multi-dimensional classification design derived from the Bloom filter technique, which is used to decide the frequently written addresses and to isolate them into the SRAM. By combining this scheme with prior wear-leveling methods, we can achieve a synergistic result in the operational lifetime of the hybrid-PCM main memory.

5.1 Multi-Dimensional Classification

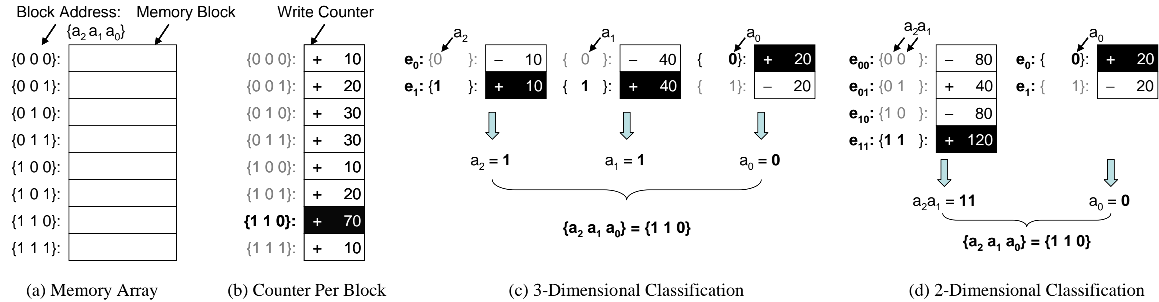
Temporal locality can be indicated by the write frequency of a certain period. For typical program phase behavior, one can record the write frequency for a recent execution phase to predict that of the future phases. A typical way to measure write frequency for each memory block is to count the number of writes accessing the memory block during a period. This scheme was employed for *segment swapping* [4], where each segment uses a counter to indicate its degree of wear-out and the information is used for wear leveling. The advantage of this counter-per-block scheme is that it can precisely measure the degree of wear-out for each memory block, but at huge storage costs. For instance, given a 1GB memory with 256B memory blocks, the counter-per-block scheme requires more than four million (2^{22}) counters. This prohibitive overhead is the main reason why the segment-swapping scheme suggested using 1MB as the segment size but no smaller. Unfortunately, the 1MB segments are too large to handle. First, when a 1MB segment is swapped with another selected segment, it simply takes a long time to transfer the entire data. During the time

of swapping, the memory controller will stop dispatching new memory requests, which affects the performance. Furthermore, with the segment-level counters, it is impossible to identify the write distribution at a finer granularity than a segment, *e.g.,* , the cache-line size. In other words, it is impossible to identify whether writes are highly concentrated on a few addresses or they are evenly dispersed across the segment. Hence, we need an efficient way to measure write frequency at a fine-grained level.

Although it may appear to be difficult to measure the exact write frequency for all fine-grained memory blocks without incurring large hardware overhead, it would be very helpful if we can estimate the outliers that show much higher write frequency than the others. Toward this, we propose a concept of *multi-dimensional classification* that can efficiently estimate which memory blocks show aberrant behavior. In this scheme, a memory-block address is projected multiple times onto all different dimensions, each of which employs its own hash function to project the memory-block address onto one of its elements. In this scheme, our dimension contains two, four or eight elements. Each element in a certain dimension has its own counter that is updated on every PCM write access. When a write address is projected onto the element, the element's counter is increased by a value equal to the number of elements in the dimension minus one. If a write address is projected onto another element, the counter is decremented by one. For example, if the number of elements in each dimension is four, the counter of the element that a write address projected onto will be increased by three and the three counters of the other elements are decremented by one. Thus, if all elements have the same probability to be projected onto, each counter value will stay around zero. However, if an element is more frequently accessed than others in a dimension, its counter value will stand out positively. Therefore, the most frequently written address can be probabilistically estimated and identified by picking out the most frequently written element projected in each dimension.

In Figure 26, the counter-per-block scheme is compared against our multi-dimensional classification. For simplicity, in this example, the entire memory space comprises eight

memory blocks shown in Figure 26(a). For the most recent 200 writes, the counter-per-block scheme in Figure 26(b) counts the exact number of writes for each block using eight counters. As shown, the seventh counter of the memory block 6 has the highest value of 70 writes, *i.e.,* the most frequently written block.



(a) Memory Array (b) Counter Per Block (c) 3-Dimensional Classification (d) 2-Dimensional Classification

Figure 26: **Examples of multi-dimensional classification.**

Figure 26(c) depicts a three-dimensional classification mechanism for the same example. First, the hash functions for the three dimensions are simply defined as $H_2(x) = a_2$, $H_1(x) = a_1$, and $H_0(x) = a_0$, respectively, where x is a three-bit block address, $\{a_2a_1a_0\}$. In other words, each bit position in a memory block address is assumed to represent a dimension. Because all elements in one dimension should be represented with one bit, each dimension has the following two elements: 'e_0' and 'e_1', accounting for one of the binary value. Since each element requires its own counter, the three-dimensional classification contains six counters in total. For example, a write access to the address {110} increments three counters corresponding to e_1 for the a_2 dimension, e_1 for the a_1 dimension, and e_0 for the a_0 dimension; and decrements the other three counters.

After the same 200 writes in Figure 26(b), the final counter values are shown in Figure 26(c). In the a_2 dimension, the e_1 counter has a bigger value than the e_0 counter. It indicates that the number of writes to the block addresses {**1**00}, {**1**01}, {**1**10} and {**1**11} are higher than the number of writes to the other addresses. Similarly, the e_1 counter for the a_1 dimension and the e_0 counter for the a_0 dimension have higher values than the others in the same dimension, respectively. Therefore, the dimensional result indicates that the memory

block {110} has the highest probability to be the most frequently written block.

Figure 26(d) shows another example for a two-dimensional classification scheme. Different from Figure 26(c), the left dimension in Figure 26(d) has four elements. Thus, in the left dimension, the counter of an element that a write address is projected onto increases by three on every write access while the other three counters decrease by one. After the same 200 writes, the final counter values of the left dimension indicates that one of the two block addresses {**11**0} and {**11**1} are the likely candidates of the most frequently written block. By combining the result with that of the right dimension, the two-dimensional classifier obtains the same result with the previous three-dimensional classifier.

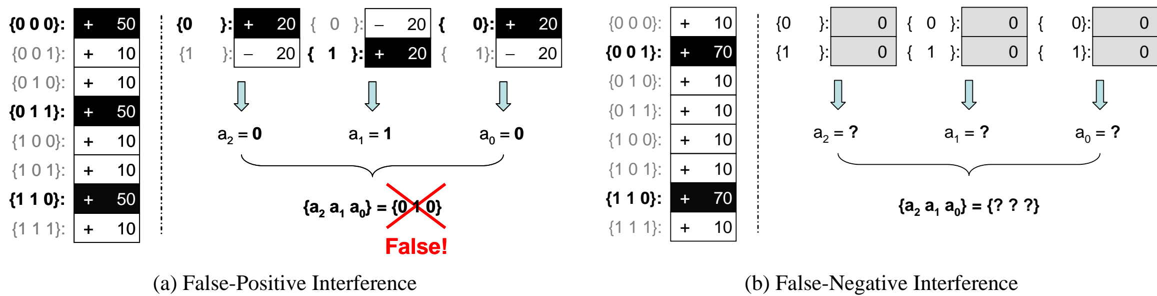
As shown in these examples, with the multi-dimension classification we can efficiently estimate (based on probability) the most frequently written block. The total storage requirement for this scheme is (*the number of dimensions*) × (*the number of elements for each dimension*). For instance, when assuming a 1GB memory composed of 256B memory blocks, instead of having more than four million counters in the prior segment-swapping scheme, we only need 44 counters for 22 dimensions represented by each bit of the 22-bit block address. Even with the low hardware overhead, we can detect the outlier addresses, *i.e.,* the memory blocks being repeatedly written within a given period. If we can accurately filter out these outliers, we can slow down the wear-out of limited write endurance memory cells.

Note that in our real design we did not restrict ourselves to only isolate one single most frequently written memory block for an entire application. Rather, the outlier addresses identified by a threshold mechanism are continuously isolated to a separate and durable memory structure during the course of an execution. The implementation of our hybrid-PCM architecture and the decision mechanism will be detailed in Section 5.3.

5.2 Interference and Its Implication to Design

Similar to other fast approximation methods, the multi-dimensional classification scheme also suffers from inaccuracy because of the fact that more than one memory block address can be projected onto the same element (*i.e.,* counter), thereby leading to address aliasing or interference. This interference is an outcome of using a hash function for each dimension and using a smaller number of counters for bookkeeping. The interference could be mitigated by increasing the number of dimensions (*i.e.,* hash functions) or the number of elements in each dimension.

The main issue of incrementing the number of dimensions or the number of elements, however, is that it incurs high storage overhead for counters. To address this issue and implement an appropriate number of counters, we need to know what are the possible types of interference present in our scheme. Figure 27 shows the examples of the interference that is classified into *false-positive interference* and *false-negative interference*. In the example of false-positive interference, three memory block addresses {000}, {011}, and {110} were written 50 times and the other five were written 10 times. However, according to the counter results of our three-dimension classifier, the memory block {010} is identified as the most frequently written block, which is not even within the top three frequently written ones. The reason is that in each dimension, two of the three most frequently written addresses happen to map to the same element.



(a) False-Positive Interference (b) False-Negative Interference

Figure 27: **Examples of interference.**

To reduce false-positives, more dimensions should be implemented to have more distinction of different addresses, and then the sum of the counter values across dimensions is used to obtain the decision by the majority. If the sum is larger than a certain threshold value, then our scheme classifies the block address as aberrant. For example, the memory block {000} is projected onto e_0 (20) for the a_2 dimension, e_0 (−20) for the a_1 dimension, and e_0 (20) for the a_0 dimension. Thus, its sum of the corresponding counter values is 20. Likewise, the sums for the other memory blocks from {001} to {111} are −20, 60, 20, −20, −60, 20, and −20. Even though this majority decision cannot avoid false-positives, at least it can detect all true-positives by adjusting the threshold value. In this example, if the threshold value is set to 10, then the memory blocks {000}, {010}, {011}, and {110} are classified as aberrant. We will discuss our implementation toward this in Section 5.3.

Similarly, because of the address aliasing, the counter results can have another type of interference caused by false-negatives. In this scenario, frequently written memory addresses could remain undetected by our estimation scheme. In another example illusterated in Figure 27(b), the memory addresses {001} and {110}, are more frequently updated than the others. However, all counters end up with the same accumulated values since the two frequently accessed addresses {001} and {110} happen to be antipodal in the three dimensions. These antipodes located in the exact opposite positions can conceal themselves completely from our multi-dimensional classifier.

For normal applications, the probability of locating the exact antipodes diminishes quickly as the number of dimensions increases, because the normal applications do not intentionally pair up addresses to generate false-negative interference. Given malicious attacks threatening PCM reliability, nevertheless, this false-negative interference is a severe weakness. If the dimension projection mechanisms are fixed or deterministic, an adversary can reverse-engineer the projection mechanisms with side-channels using latency differences, and then the corresponding false-negative interference patterns can be easily generated. For example, assume that it is revealed that each dimension has two elements and

the addresses detected by the multi-dimensional classifier are isolated to an SRAM cache, which has much shorter latency to access than that of a PCM memory. Using this latency difference, an adversary can search for a false-negative interference pair. To prevent adversaries from using the side-channels, therefore, we need to hide the projection mechanisms or to keep changing them. From these considerations about interference, we propose a secure multi-dimensional classification scheme to be described in the next section.

5.3 The Implementation of Our Hybrid-PCM Architecture

Thus far, we have described the basics of our proposed multi-dimensional classification technique. In this section, we propose a novel and efficient implementation to realize our technique for a hybrid-PCM architecture.

5.3.1 Overall Control Flow and Isolation Cache

The essential idea of improving the write endurance for a PCM main memory is to filter out the frequent memory writes from being written back to the PCM main memory. Based on the decision made by our multi-dimensional classification scheme, when an incoming write address is classified as the most frequent written block (at this point of time), it will be transferred to a small SRAM cache, called *isolation cache*. The isolation cache is fully-associative and uses the least-recently-used (LRU) policy for its line replacement.

Figure 28 depicts the overall block diagram of our hybrid-PCM architecture. When a new writeback address arrives, our proposed mechanism checks whether the address is a hit in the isolation cache. At the same time, a *decision maker* also evaluates and determines if it is worthwhile to transfer the block to the isolation cache based on the counter values in our multi-dimensional classifier. Upon a cache hit, the corresponding cache line is updated accordingly while the PCM memory block has stale data. If the address results in a miss and the decision maker indicates that the address should be transferred, then the writeback address and its data will be inserted to the isolation cache. When an insertion occurs, the LRU cache line in the isolation cache will be written back to the PCM main memory.

Otherwise, the address and its data will bypass the isolation cache and go to the PCM main memory directly. Note that a read request also requires to look up the isolation cache. In case of a hit, the isolation cache returns the read data because the corresponding PCM memory block is out of date. However, regardless of the cache look-up result, the read request neither looks up the decision maker nor changes the state of the decision maker.

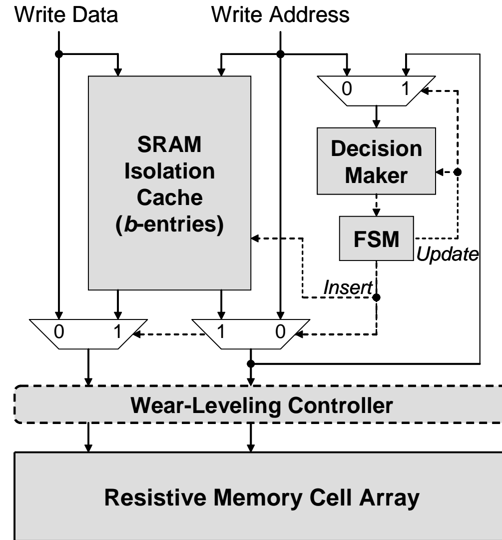


Figure 28: **The overall block diagram of our hybrid-PCM architecture.**

The decision maker is updated whenever a write address is sent to the PCM main memory, *i.e.,* either during an eviction from the isolation cache or during a direct writeback that bypasses the isolation cache. The update primarily increases or decreases the counter values in each dimension. The flow control of the address and its data for updating the decision maker is managed by a small finite state machine (FSM) as shown in Figure 28. The design of decision maker is detailed in Section 5.3.2.

5.3.2 Decision Maker

The decision maker is responsible for isolating frequently written memory blocks and represents the most critical part of our proposed design. Figure 29 depicts the block diagram of the decision maker consisting of d dimensions. Each dimension employs its own hash

function to project an m-bit block address onto one of its 2^n elements (*i.e.,* counters).[1]

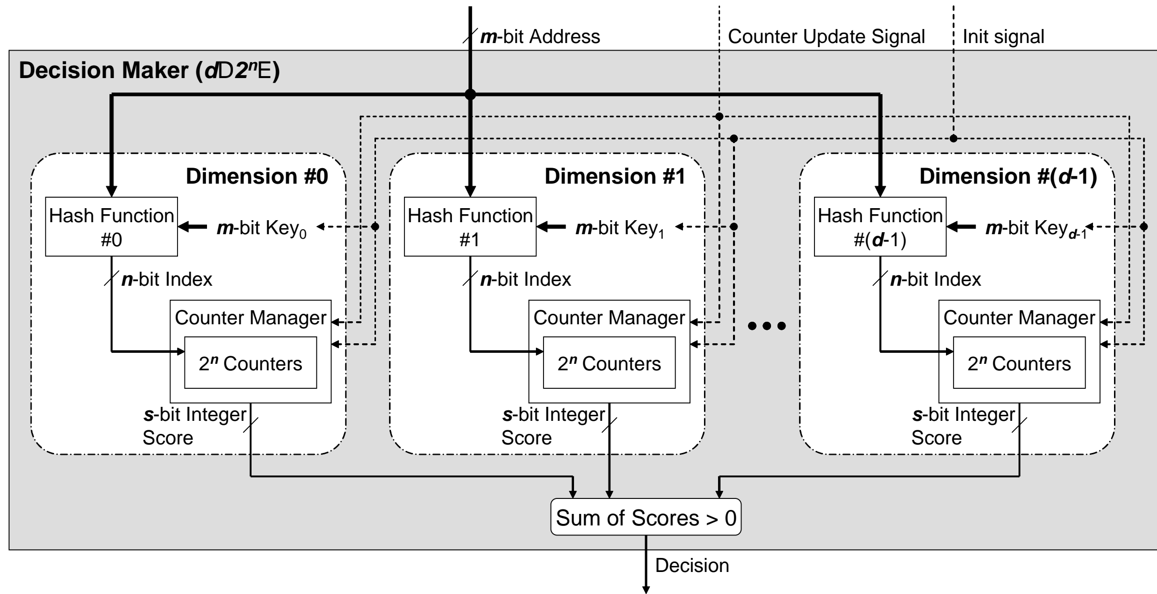


Figure 29: **The block diagram of a decision maker.**

As illustrated in Figure 30(a), each hash function generates an n-bit counter index from an m-bit block address and an m-bit randomized key. Thus, using a different randomized key for each dimension can differentiate it from the other dimensions. Even in the same dimension, changing the key can project a block address onto a different element. As mentioned in Section 5.2, a static, unaltered key may render our scheme vulnerable. By changing the key values dynamically, therefore, our scheme can be protected from malicious side-channel attacks. Note that our scheme employs a simple hash function with low-cost hardware, which enables to implement our scheme inside a PCM chip. For instance, the hash function described in Figure 30(a) performs a bitwise AND operation between an input address and a key, and then chops the result into many n-bit pieces. Lastly, to generate an n-bit index, it performs a bitwise XOR operation among the chopped n-bit pieces.

To keep track of the information of write recurrence behavior, each dimension updates its 2^n counters whenever a write data is transferred to the PCM main memory. To measure

[1]The indexing method is similar to a counting Bloom filter proposed for web caching [37]. However, the way each counter is updated is quite different, which will be discussed in Section 5.6.

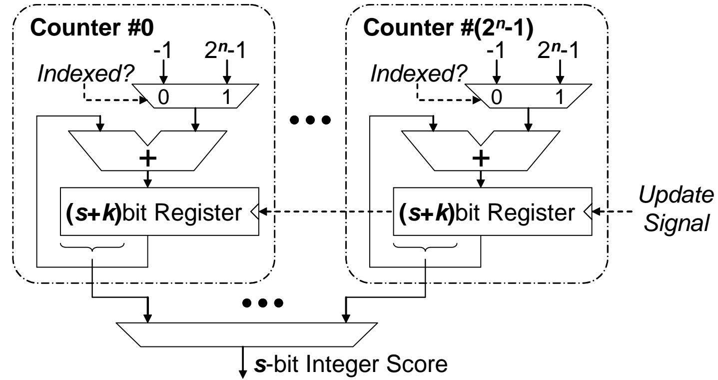
```
int n; /* the size of a counter index */

int hash (int addr, int key) {
    int index = 0;
    int mask = (1 << n) – 1;
    int temp = addr & key;

    while (temp != 0) {
        index = index ⊕ (temp & mask);
        temp = temp >> n;
    }

    return index;
}
```

(a) A hash function.



(b) A counter manager.

Figure 30: **The details of a hash function and a counter manager.**

the degree of write frequency with the counters, we chose to *reward* the counter indexed by the output of the hash function while the other counters in the same dimension are *penalized*. In other words, the indexed counter is increased by a value of $2^n - 1$ and the other $(2^n - 1)$ counters in the dimension are decremented by one. For instance, if each dimension has four counters (where $n = 2$), then the indexed counter is increased by three and the other three counters are decremented by one. Therefore, unless the writes to the PCM main memory are evenly projected onto all elements, the frequently indexed counters keep increasing their values. Figure 30(b) depicts the block diagram of the counter manager in each dimension. Each counter has an $(s + k)$-bit register to store the current counting value. The upper s bits will be used as a score in the decision-making process while the least significant k bits are used as a fluctuation margin. By tuning the value of k, we can

determine how much deviation is required to affect the score.

When receiving a new writeback block address, the decision maker should decide whether to isolate the memory block or not. To do so, the decision maker forwards the block address to all dimensions, and then each dimension outputs an s-bit score value representing how frequently the memory block is updated. To make the final decision, the decision maker employs the process of a binary classifier. If the sum of all the dimension scores is larger than a threshold value, the decision maker decides that the current address is an outlier and will isolate it to the isolation cache. Otherwise, the writeback is sent to the PCM main memory. For our experiments in Section 5.5, we set the threshold value to zero, because a positive value of the summed up score of all dimensions indicates the current writeback address shows certain deviation above the fluctuation margin.

After migrating an outlier block to the isolation cache, all the counter values will be reset back to zeros. Although it is required to go through another learning phase even for the addresses that have already reached close to the boundary of the fluctuation margin, the reset and re-initialization can reduce the probability of false-positive interference by eliminating the current counter values biased to the just-isolated address. Since a new insertion to the isolation cache evicts one cache line, these counters, after reset, will be updated with the address of the evicted cache line. At the same time, the random keys for hash functions will be re-generated.

5.3.3 Implementation Overhead

Since we advocate a tightly integrated design embedding our scheme within a PCM chip, we should consider the hardware costs for the integration. As explained in Section 5.3.2, we employs a lightweight hash function composed of several XOR and AND gates. Thus, the isolation cache and the counters of the decision maker occupy most of the hardware overhead. However, given a PCM *dual in-line memory module* (DIMM) consisting of 8 chips, the data array of the isolation cache can separate into 8 chips. Thus, assuming a 256B 32-entry isolation cache and a 64-counter decision maker, the total area overhead per

chip is about 1KB, which is affordable for the uncompromising reliability of the PCM main memory.

5.4 Impact to Wear Leveling under Malicious Attacks

Our isolation cache and multi-dimensional estimation scheme can, in fact, leverage wear leveling more efficient. Ideally, a b-entry isolation cache can completely filter out b malicious addresses, even though interference may generate false positives. Thus, when a malicious process tries to subvert our system, it should target and attack more than b addresses. Assume that an adversary attacks $(b+1)$ address targets. Filtering out only b addresses and letting one address bypass will strike the weakness of wear leveling. In this case, it appears as if only one single target address is being attacked repetitively, representing the worst-case attack (*e.g.,* , birthday paradox attack [24]) to a wear-leveling system as stated in prior literature [8, 18, 24]. To mitigate this scenario, the $(b+1)$ addresses should be equally sent to and observed by the wear-leveling controller and the PCM main memory.[2] For example, assume that an isolation cache is composed of four cache lines and a malicious process attacks five target addresses by writing to each address 100 times. For the naïve filtering that isolates b fixed addresses out but lets the last address slip through, the wear leveling will observe this address 100 times, fulfilling the worst-case attack that keeps hitting the same address block 100 times. Our proposed scheme, nonetheless, will be able to cache four of the five targeted addresses by taking turns. As a result, the wear-leveling controller will observe each of the five addresses 20 times each, representing an ideal situation for wear leveling that the 100 writes are evenly distributed to the five addresses. Our scheme can operate close to the ideal write distribution because it always detects and inserts the most frequently written address based on the current write frequency to the PCM main memory.

Figure 31 shows how the distribution of attack writes affects a PCM lifetime under a wear-leveling scheme. To remap the entire memory space (composed of 4M memory

[2] As shown in Figure 28, our wear-leveling controller sits in-between our proposed architecture and the underlying PCM main memory.

blocks in this evaluation), secure wear-leveling schemes based on randomization, such as *security refresh* [18] and *start-gap* [9], typically require a certain amount of writes called a remap period. The mapping from an address to a physical memory block does not change during the remap period and thus a malicious process can attack a single target memory block during that period. For example, the left-most bars show relative lifetimes[3] under a single-target-address attack (1 TAddr in the legend) when varying the remap period from 4M writes to 512M writes. As shown, although a shorter remap period is critical to extend the lifetime, it also increases the write overhead[4] specified in the parentheses. To both shorten the remap period and avoid high write overhead, previous wear-leveling schemes divide the entire memory space into multiple regions and perform intra-region wear leveling and inter-region wear leveling, simultaneously. However, the multi-region wear leveling also increases hardware costs for maintaining the wear-leveling status of all regions. On the other hand, our scheme prevents all malicious writes from targeting a single address. Thus, the malicious writes are dispersed to multiple target addresses. As shown in Figure 31, doubling the number of target addresses has a similar effect of reducing the remapping period by half with the same write overhead. In other words, our scheme can help the wear-leveling schemes increase the lifetime with low hardware costs by forcing a malicious attack to disperse their writes to multiple target addresses.

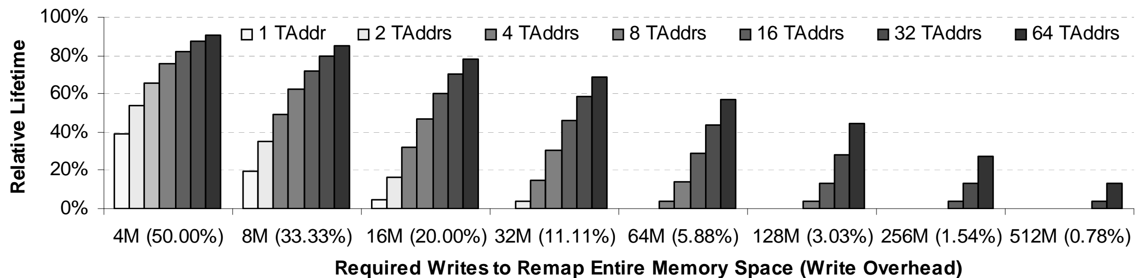


Figure 31: **The effect of write distribution on wear leveling.**

[3]The relative value is normalized to the theoretical maximum number of writes calculated as the number of memory blocks (4M) × write endurance cycles (10^8).

[4]Write overhead is defined as the number of additional writes for address remapping over the total number of writes to memory arrays including the additional writes.

Another merit of our scheme is that the insertion signal for the isolation cache can be used to control the rate of address remapping in wear leveling. Controlling the rate is beneficial to improve the overall lifetime [8]. In our scheme, frequent insertions indicate that an attack is present and the wear-leveling controller needs to speed up the rate of address remapping. On the contrary, a low insertion rate into the isolation cache can be interpreted as current write patterns are uneventful, and thus no need for the wear-leveling control to shuffle the addresses fast. Note that even though attacks are present, if the frequently written addresses fit into the isolation cache, then we do not need to accelerate the address remapping rate. Furthermore, the hit rate of the isolation cache can be used as a threshold for warning the current attack situation to a system operator. We will quantify the benefit of our scheme for wear leveling in Section 5.5.4.

5.5 Experimental Evaluation and Analysis

To achieve the high reliability of a hybrid PCM main memory system, our scheme should effectively filter out the high-deviation write addresses. To evaluate that, we devise an attack model where a loop body consists of t target addresses (TAddrs) and r random addresses (RAddrs). The TAddrs are changed only when a wear-leveling scheme, if any, finishes remapping the entire memory space. It is noteworthy that when $t = 1$ and $r = 0$, the attack model is equivalent to the *birthday paradox attack* (BPA) that is the best known attack method to wear leveling employing randomization [8, 24]. The size of a memory block is 256B in our experiments and there are 2^{22} memory blocks in each memory chip. Note that our management scheme is embedded within the chip.

5.5.1 Sensitivity Study for Interference

To attain high accuracy for our decision maker, it is critical to minimize interference. Figure 32 shows the occurrence rate of false-negative interference for the different number of TAddrs from two to 64. Each configuration was simulated 450 times with different TAddrs. The number of dimensions (D), the number of elements in one dimension (E) and

the total number of counters ((#)) are varied for each configuration. As shown in Figure 32, more TAddrs lead to higher false-negative interference while using more counters can decrease it. By using more than 128 counters for the decision maker, the false-negative interference is completely gone. Note that we performed our evaluation up to 64 target addresses because given a wear-leveling technique, increasing the number of target addresses makes the malicious attack less efficient. In case of the 64 TAddrs, even the right-most wear-leveling scheme in Figure 31, incurring 0.78% additional writes for shuffling the entire address space, can achieve 13.25% of the theoretical lifetime, which is around twelve months. Thus, it is pointless to attack more than 64 target addresses.

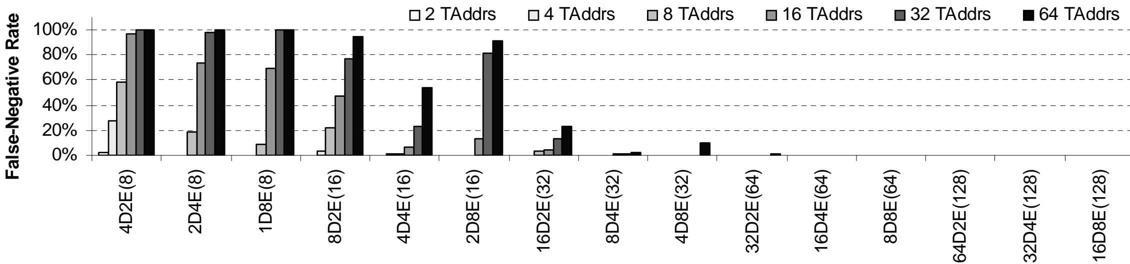


Figure 32: **The rate of false-negative interference.**

Distinct from the false-negative interference that makes our scheme to fail to capture target write addresses in the isolation cache, false-positive interference deceives our scheme into capturing the wrong addresses and evicting true-positive ones from the isolation cache. To measure how often false-positives happen, we use a 16-entry isolation cache and an attack model with 16 TAddr followed by one RAddr for each iteration. Figure 33 shows the ratio of the number of RAddr inserted to the 16-entry isolation cache to the total number of RAddr when using a $(3+k)$ counter scheme, *i.e.,* the upper 3 bits for scoring and the other k bits for the fluctuation margin. Obviously, increasing counters reduces the rate of false-positive interference. However, with the same number of dimensions, increasing counters for each dimension has an adverse effect. For example, when $k = 6$, the false-positive rate of 8D8E(64) is twice higher than that of 8D4E(32). It is due to employing the reward and penalty mechanism of the counter manager. Since increasing the number of elements

in one dimension also increases the reward for an indexed counter, the big reward raises the chance for the counter value to pass the threshold ($64 = 2^k$ in this example). Thus, to reduce the false-positive interference, it is desirable to increase the k value or the number of dimensions as shown in Figure 33.

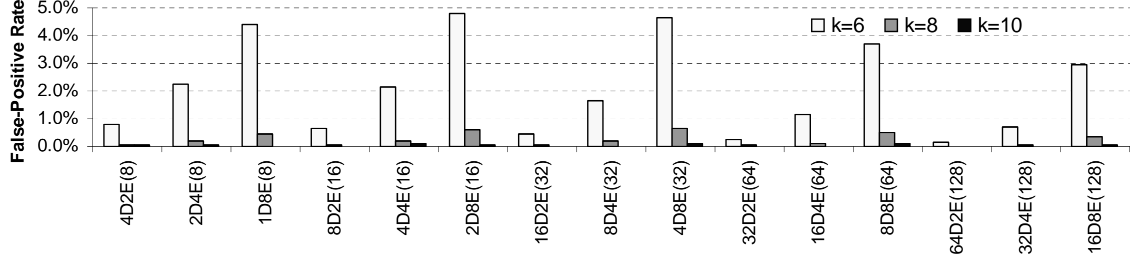


Figure 33: **The rate of false-positive interference.**

5.5.2 Wear-Out Evaluation

From the experiment of false-negative interference, we observed that 128 or more counters can almost eliminate the false-negative interference. If it is not possible to concoct a malicious code to create false-negative interference, then another efficient attack method against our scheme is to force capacity misses in the isolation cache. To do so, an adversary should attack more memory blocks than the number of entries of the isolation cache. If the number of isolation cache entries is b and the number of target addresses is t, ideally the total writes reaching the PCM main memory will be (the total number of writes $\times \frac{t-b}{t}$). Therefore, the ideal wear-out of each target memory block is (the total number of writes $\times \frac{t-b}{t})/t$, as mentioned in Section 5.4.

Figure 34 shows the ratio of the worst-case wear-out in our simulation against the ideal wear-out for total 2^{22} writes. During all the writes, no wear leveling is performed. We varied the isolation cache size from four to 32 entries. As mentioned earlier, the minimum number of write addresses that an adversary can use to attack our system is $(b + 1)$, where b is the number of entries in the isolation cache. From Figure 34, it is observed that as increasing the number of target addresses, the worst-case wear-out also increases because of

the initial learning phase before any of the target addresses is isolated. For example, when 33 target addresses begin to be written, the last one (*i.e.,* the 33rd address) must go through 33 learning phases to be isolated into the isolation cache. This long learning period of the last target address results in the worst-case wear-out in the PCM main memory. However, the long initial learning phase is not significant to the lifetime of the PCM main memory since it happens only once. Lastly, during our wear-out experiments, the configuration 32D2E(64) using a 32-entry isolation cache suffered from one false-negative interference. Thus, the two target addresses were not detected and the memory blocks show 33 times higher wear-out than the ideal case.

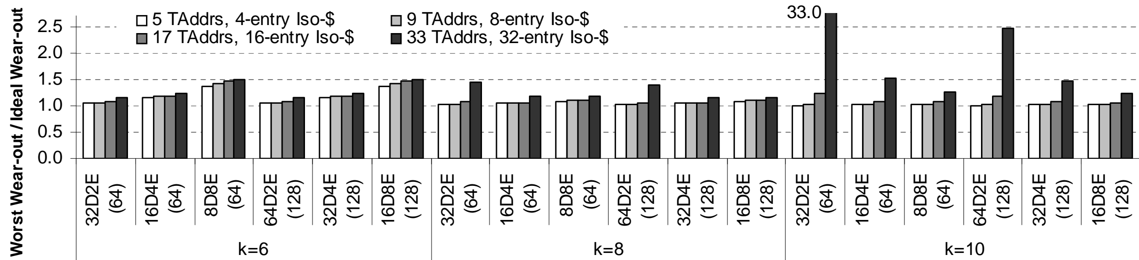


Figure 34: **The maximum wear-out of attack addresses in 300 simulations.**

5.5.3 Evaluation for Normal Applications

So far, we have shown our scheme successfully detects a malicious attack. Even though our scheme pursues a highly-reliable resistive memory system against the worst-case scenario, our scheme is helpful for normal application as well. Now we will evaluate our scheme for normal applications running on a tri-level cache system. We used PIN tool [27] and simulated selected SPEC2006 benchmark applications. The memory hierarchy includes a 32KB L1 data cache, a 1MB L2 unified cache and an 8MB L3 unified cache, all eight ways. The cache line size is 64B for L1 and L2 and 256B for L3. Four SPEC2006 applications that show the highest writeback rate from an 8MB L3 cache were chosen including 429.mcf, 471.omnetpp, 482.sphinx and 483.Xalan. We simulated a decision maker of 16 dimensions and each dimension has four counters.

Figure 35 shows the number of hits in isolation cache for an epoch of 100,000 L3 writebacks by varying the isolation cache size from 4 to 32 entries and the *k* value of the decision maker from 6 to 10. Note that, the number of hits represents the number of resistive memory writes saved. Also shown is the scenario without the decision maker, in which the isolation cache acts like a tiny L4 cache where all the writes pass through. As shown, the number of hits are much increased by applying our scheme. It indicates that even after filtering of the L3, there is still temporal locality and our scheme can detect it to reduce the write frequency to the resistive memory. An interesting observation is that using a small isolation cache requires a large *k* value to get a higher hit rate, whereas a large isolation cache can obtain a high hit rate with a small *k* value. That is, using a small isolation cache requires a more precise decision. In our scheme, the *k* value is important to detect the recurrence of writebacks. For example, a large *k* value will take a long time to train and detect a frequently recurred address, leading to lost opportunities for write reduction in the resistive memory. In all cases of our simulations, using a small *k* (= 6) shows much higher hits than using the isolation cache without a decision maker.

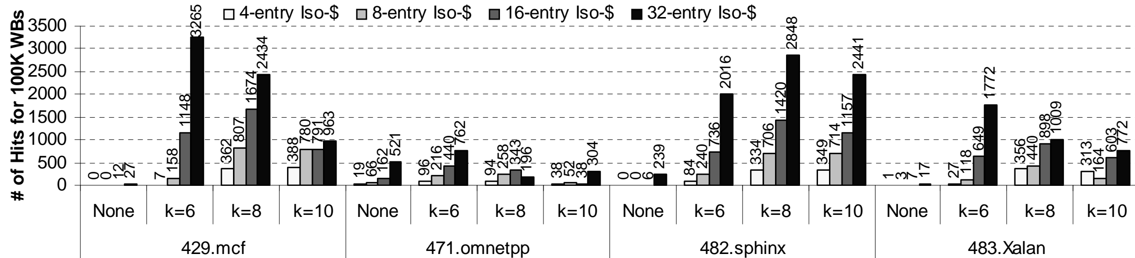


Figure 35: **The saved PCM writes of SPEC2006 for 100K writebacks.**

5.5.4 Impact to Wear Leveling

To study the impact of our scheme to wear leveling, we modeled it with the secure wear-leveling scheme, *security refresh* [18]. The security-refresh scheme remaps (or refreshes) two addresses in a randomized fashion upon every refresh interval. A two-level security-refresh scheme shows a more than five year lifetime under a continuous write attack. In the

two-level security refresh, an outer security refresh wear-levels a 1GB memory bank with 512 sub-regions inside the memory, while each sub-region is simultaneously wear-leveled by an inner security refresh.

Although the two-level security refresh extends the lifetime of a single-level security refresh by more than four times, its major setback is the overhead. Even though each security-refresh controller uses only four registers, the two-level security refresh with 512 sub-regions will require around 12KB of hardware overhead. In our scheme, the isolation cache occupies most of hardware requirements while our decision maker consists of at most tens of counters. Given a 256B cache-line writeback from LLC, a 32-entry isolation cache requires 8KB for data storage. Thus, we evaluate the application of our scheme to a single-level security refresh. Nonetheless, we found we can achieve even higher endurance with lower hardware overhead than the two-level security refresh. We will discuss the results in Figure 36 subsequently.

Moreover, to combine our scheme with a single-level security refresh, we propose a rate control mechanism for the refresh rate of the single-level security refresh. The refresh rate is measured as the reciprocal of the refresh interval. In the original security refresh, the refresh rate is controlled by a counter based on the number of writes to the memory. The counter is incremented by one for each write to the PCM main memory. Then, an address remapping takes place whenever the counter is about to overflow. In our scheme, we increase the counter by a value larger than one upon every cache line eviction from the isolation cache. In other words, the evictions expedites the address-remapping process. We call this stride value an *expediting factor*. The rationale is that a line eviction from the isolation cache results from the insertion of another attacked line, *i.e.,* a capacity miss. This indicates that the the number of attack addresses exceeds the capacity of the isolation cache, which poses a threat to PCM reliability. In our scheme, furthermore, as the attack writes show high deviation, the counter values for their target addresses cross the fluctuation margin boundary fast. Thus, their short learning phases increase the frequency of evictions

from the isolation cache. Therefore, by using the eviction rate, we can expedite the wear leveling to protect our PCM system from the intensive attacks.

Figure 36 shows the lifetime of each configuration when combining our scheme with a single-level security refresh using a 7-bit counter, *i.e.,* remapping two memory blocks upon every 128 writes. Note that without our scheme this configuration of the single-level security refresh endures only a few minutes under a malicious write attack. In the graph, all the lifetimes of configurations are depicted as a relative value to the theoretical maximum lifetime of 97.1 months under a perfect wear-leveling scheme. In each bar, the lower part depicts the lifetime spent for demand writes and the upper part is for write overhead. We used a 16D4E(64) configuration for the decision maker with $k = 8$. As the expediting factor is increased from one to 2048, the lifetime for each configuration keeps increasing even though its write overhead also increases. Thus, even the configuration using a 4-entry isolation cache (1KB) endures 60.8 months including 25.7 months for additional writes. Given the refresh rate (R) of a single-level security refresh, the rate of write overhead is calculated by the expediting factor (F) and the eviction rate ($E = \frac{\text{the number of evictions}}{\text{the number of writes to PCM}}$). In the graph, the eviction rate of each configuration is specified in parentheses below the configuration name. Then, the rate of write overhead can be calculated by $\frac{(1-R)+R\times F}{R}$.

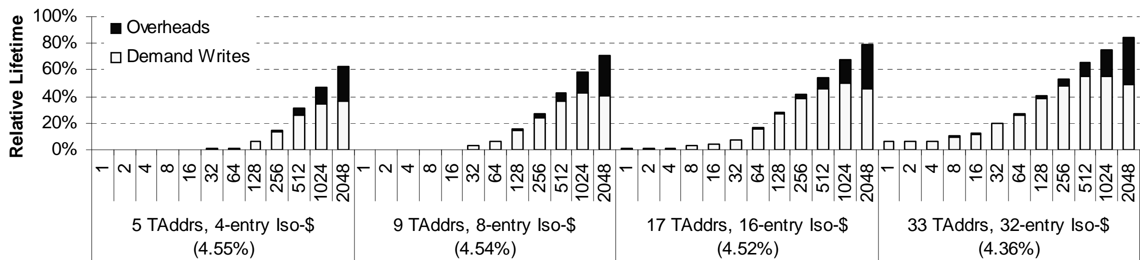


Figure 36: **Lifetime improvement using single-level security refresh.**

However, the high write overhead under a malicious write attack is affordable to protect the PCM main memory, while it must be mitigated in normal application behavior. To evaluate the write overhead for normal applications, we simulated SPEC2006 benchmark applications. In the simulations, the memory hierarchy includes a 32KB L1 data cache, a

1MB L2 unified cache and an 8MB L3 unified cache, which are all eight ways. The cache line size is 64B for L1 and L2 and 256B for L3. Our 32-entry isolation cache and the 16D4E(64) decision maker are located after the L3. Among the applications, 483.Xalan shows the highest eviction rate of 0.24%. Thus, by using an expediting factor of 128, we can restrict the write overhead for normal applications down to 1.0%. With the expediting factor of 128, the configuration using a 32-entry (8KB) isolation cache can endure 39 months under a malicious attack. Note that when restricting write overhead at around 1.0% in the two-level SR scheme with 512 sub-regions (12KB), its lifetime is 26.0 months.

5.6 Related Work

The front-end of our decision maker is a variation form of the original *Bloom filter* [38], which also employs multiple hash functions to map the outcomes of an input set to a bitvector to create a signature for the given set. A *counting Bloom filter* [37] replaced each bit of the bitvector with a counter to enable the deletion of an evicted element. Unlike our multi-dimensional counters, the counting Bloom filter indexed all hashed results into a unified counter array. Ghosh *et al.* described a *segmented counting Bloom filter* [39] that has both the bitvector and the counters with duplicated hashes in one Bloom filter for reducing energy and expediting lookup for a match in the bitvector. Note that the use model and update mechanism of the counters in our scheme are drastically different from those of prior Bloom filters. The prior use of counters was to enable the insertion and deletion of elements without rehashing given no saturation occurs in the counters. In contrast, our counters are used to train the decision maker for scoring the write frequency of observed addresses. Then the sum of the counters above threshold of all dimensions is used as a binary classifier, in some sense, similar to a single-layered *perceptron neural network* studied in branch predictors. Basu *et al.* proposed a *skewed Bloom filter* to count the L2 cache misses [40]. Even though the skewed Bloom filter used multiple dimensions, its counter management and decision are very different from ours. With the counter scheme, it is hard

to differentiate aberrant write patterns from normal ones because the scheme used the absolute counter values rather than the degree of deviation. Dharmapurikar *et al.* proposed *parallel Bloom filters* (PBF) for deep packet inspection [41]. Each of the PBF contains a signature of a particular string length to identify suspicious network traffic. Recently, Xiao and Hua [42] proposed a generalized PBF based on counting Bloom filters that keeps multiple attributes of a given element in the PBF. The PBF design is somewhat similar to the design principle of our multi-dimensional Bloom filters. However, the organization and use model of our counters are very different from prior art.

The previously proposed PCM architectures [1, 4, 6, 43] advocated PCM to be used as a main memory or a last-level cache with little or no consideration of write-endurance issues. These proposals leave PCM vulnerable and unusable in practice under the worst-case scenarios or malicious write attacks. Park *et al.* [16] studied a vertical hybrid-DRAM/PCM architecture from power management perspective but ignored PCM's reliability. Qureshi *et al.* [2] suggested a vertical hybrid hierarchy using a DRAM buffer as a filter with wear leveling. Their scheme will incur large overhead because of the sheer size of the DRAM buffer and the hardware storage (4MB) for the wear-leveling counters. Zhang and Li proposed a horizontal hybrid-PCM/DRAM using OS to migrate hot pages [17] to a parallel DRAM based on page-worn information. This scheme will not work in tandem with a PCM employing wear leveling, which OS has no control over. As discussed in Section 2.3, a filter-based DRAM cache scheme proposed by Jiang *et al.* [44] and a multiple-buffer scheme proposed by Lee *et al.* [1] are vulnerable to malicious attacks because of their deterministic behavior.

A *delayed write queue* (DWQ) scheme was proposed by Qureshi *et al.* [9], where a PCM write queue delays the writes to PCM until the number of pending writes exceeds a cetain threshold called a delayed write factor. If the write queue supports a function to merge write requests to the same address, then the DWQ has the same effect with our hybrid architecture in terms of the capability of helping wear-leveling by forcing adversaries to

attack multiple targets. A main difference is that the DWQ fails to reduce the number of writes to PCM. For instance, when using a n-entry DWQ, an adversary can easily shift out the writes in the DWQ to PCM by sequentially attacking $(n + 1)$ target addresses. In other words, after k iterations of attacking the $(n + 1)$ targets, the total number of writes to PCM will be $k \cdot (n + 1)$. On the other hand, our n-entry isolation cache can reduce it to near $\frac{k \cdot (n+1)}{n+1}$ as described in Section 5.5.2.

Also, Qureshi *et al.* proposed a *practical attack detection* (PAD) scheme which can estimate the degree of attack severity, named attack density [8]. The PAD scheme makes it possible to achieve an efficient wear-leveling scheme that can control wear-leveling overhead depending on the attack density. Moreover, since a frequency-based PAD (F-PAD) composed of 16 entries has 16 frequency counters for each entry, the frequency information can be used for isolating frequently written addresses, just as the degree of deviation in our detection scheme. However, the difference is that the F-PAD estimates the degree of deviation only for the addresses stored in the 16 entries while our multi-dimensional classification scheme contains the information of all addresses recently written. This limited information of the F-PAD may cause inefficient isolation.

5.7 Summary

To address the reliability requirement for making phase-change memory a reality in the main memory hierarchy, we proposed a hybrid-memory architecture that integrates a small SRAM called isolation cache with a detection mechanism inside the PCM main memory to identify and isolate the frequent writes into the durability-proof isolation cache. We argue that the reliability of phase-change memory should be guaranteed by memory vendors, and therefore the entire wear-out management hardware must be embedded within each memory chip. We proposed the design of a multi-dimensional Bloom filter along with a binary classifier to detect suspicious memory writes and confine their future writes to the SRAM. Our technique, when combining with wear leveling, will create a synergistic

improvement for the operational lifetime. As our experimental results showed, lifetime can be extended to 81.5 months out of a theoretical limit of 97.1 months under the worst-case scenario or malicious write attack with small hardware overhead.

CHAPTER 6

TRI-LEVEL-CELL PHASE CHANGE MEMORY: TOWARD AN EFFICIENT AND RELIABLE MEMORY SYSTEM

A multi-level PCM cell can store more than one bit by defining the intermediate states between set and reset states [30]. The resistance of a PCM cell is as low as 10^3 ohms in the set state and 10^6 ohms in the reset state. By further exploiting the resistance difference, a PCM cell can have two or even more intermediate states in addition to set and reset to increase data density per-cell. For example, four-level-cell (4LC) PCM stores two bits per cell by exploiting two additional intermediate states, while eight-level-cell (8LC) PCM stores three bits per cell with six more intermediate states. Such multi-level-cell (MLC) PCM requires the following operations to function correctly. Firstly, an MLC PCM cell needs iterative write-and-verify steps to verify its written value. When the resistance fails to fall into a predefined range, a PCM chip needs to repeat the write-and-verify step. This iterative writing process takes up to eight times longer than a typical write in single-level-cell (SLC) PCM [45]. Secondly, when the resistance of an MLC PCM cell is drifted and crosses the storage level boundary, a soft error (*i.e.*, bit flipping) occurs and needs to be recovered by error correcting mechanism, which can be costly. Unfortunately, the soft error rate (SER) due to resistance drift in MLC PCM is fairly high. With a detailed model of resistance drift [21], we calculate the probability of the SER of an MLC PCM cell. As we show in Section 6.3, the SER increases over time because the resistance of the MLC PCM cell increases over time. In other words, the chance of crossing the storage level boundary increases over time along with the resistance. More importantly, Section 6.2 shows that the SER of MLC PCM is significantly higher than that of DRAM. To address such shortcomings, Xu *et al.* [21] proposed a time-aware error correction scheme, which employs extra cells for storing predefined reference resistance values. The reference cells are adjusted to the predefined values whenever the other cells in its corresponding data

block are written. When reading the data block, the resistance of the reference cells are used to compensate the drifted resistance in other cells. By using such a technique, the SER (called raw bit error rate in [21]) could be reduced from $10^{-3} \sim 10^{-1}$ to $7\times10^{-4} \sim 10^{-2}$. On the other hand, Awasthi *et al.* proposed an efficient scrub mechanism for MLC PCM [46]. The mechanism effectively reduced 99.6% of uncorrectable errors; however, the lowest possible SER for long-term writes [1] of 4LC PCM was 6.74×10^{-5}.

DRAM also experiences soft errors caused by particle strikes. Its SER is known to be an average of $25,000 \sim 75,000$ FIT (failures in time per billion hours of operation) per Mbit, *i.e.,* $25\times10^{-12} \sim 75\times10^{-12}$ per bit-hour [47]. For example, 16GB of DRAM is expected to have 3.43 to 10.31 soft errors every an hour. In contrast, 4LC PCM with SER of 6.74×10^{-5} (the lowest SER for long-term writes in [46]) is expected to incur 9.26×10^6 errors, near 10^6 times more errors than DRAM. Moreover, in this comparison, an eight-bit correction BCH ECC is assumed [46] whereas no ECC was assumed in DRAM. Even so, 4LC PCM shows several orders of magnitude higher SER than DRAM even with sophisticated ECC support.

The downside of 4LC PCM is more than its high SER and the requirement for ECC support. If we adopt redundant PCM cells for storing reference values and compensate the increase in resistance, a block of PCM cells must share the redundant cells for reducing the capacity overhead [21]. As such, any small change must read and rewrite the entire block. This strategy triggers more writes to the cells, reduces their lifespan, consumes more power, and degrades performance. On the other hand, if the scrub mechanism is used for reducing soft errors [46], the memory controller will spend more time in scrubbing than DRAM, which degrades the overall performance of the memory subsystem. However, in both cases, the performance impact of those overheads were not discussed. In summary, 4LC PCM not only has higher SER than DRAM even with ECC support but also requires extra overheads that have not been quantitatively evaluated. The motivation of this research

[1] The original paper [46] defined a long-term write as follows. Some PCM cells experience sufficiently high timing gap between writes. These types of writes are called long-term writes.

stems from these observations. As we will show in later sections, if we reduce the number of storage levels from four to three, a PCM cell shows fewer errors than DRAM, and thus eliminates the need of ECC, reference cells, and scrubbing. We compare our proposed tri-level-cell (3LC) PCM over 4LC PCM to demonstrate that 3LC PCM is the only feasible solution for putting multi-level cells into practical use.

6.1 Tri-Level-Cell (3LC) PCM

For 3LC PCM design, the most straightforward approach is to remove the most error-prone state from 4LC PCM. We first discuss the physical parameters of 4LC PCM. By measuring the resistance drift of reset and set states from iterative experiments, Ielmini *et al.* [48, 49] showed that the drift can be represented by a power-law model shown below:

$$R_{drift}(t) = R \times \{\frac{t}{t_0}\}^{\alpha} \tag{1}$$

where R and t_0 are normalization constants and α is a drift exponent. Because the main cause of the drift is the structural relaxation of the amorphous state, the drift exponent of the reset state is much larger than that of the set state in the experiments. In other words, the drift exponent will increase as the portion of the amorphous state in a cell increases.

As mentioned earlier, the resistance drift causes soft errors in the MLC PCM. To estimate the reliability impact of resistance drift, we make the following assumptions for the normalization constants and the drift exponent for each storage level. According to the experiments of Nirschl *et al.* [30], the iterative write-and-verify sequence adjusts the programmed resistance R to be located within a desired resistance range for a given storage level, where $\log_{10} R$ follows a normal (Gaussian) distribution. Thus, we assume that the logarithm of a normalization resistance, $\log_{10} R$, will follows a normal distribution of $N(\mu_R, \sigma_R^2)$. In addition, a desired programmed resistance range for a given state is set to the range within $10^{\mu_R \pm 2.75 \times \sigma_R}$ Ω and the upper and lower sensing boundaries for the state are set to $10^{\mu_R \pm 3 \times \sigma_R}$ Ω. The value of a drift exponent is also assumed to follow a normal distribution of $N(\mu_\alpha, \sigma_\alpha^2)$. The parameters we use in our drift analysis are based on the previous

works [46, 21] and described in Table 4.

Table 4: **Configuration Variables of Four-Level-Cell (4LC) PCM When t_0 = 1 s.**

Storage Level	Data	$\log_{10} R$		α	
		μ_R	σ_R	μ_α	σ_α
0	01	3.0	$\frac{1}{6}$	0.001	$0.4 \times \mu_\alpha$
1	11	4.0		0.02	
2	10	5.0		0.06	
3	00	6.0		0.10	

A soft error occurs when the resistance of a MLC PCM cell is drifted above the upper boundary of its programmed state. From the state-boundary settings described above, the condition of a soft error can be represented as follows.

$$R_{drift}(t) > 10^{\mu_R + 3\times\sigma_R} \quad (2)$$

In other words, when considering the values in Table 4, the target resistance values for the four storage levels are 10^3, 10^4, 10^5, and $10^6\Omega$, respectively, and the three sensing boundaries between two adjacent levels are $10^{3.5}$, $10^{4.5}$, and $10^{5.5}\Omega$. For instance, when the resistance of a cell programmed for storage level 2 drifts larger than $10^{5.5}$ Ω, the cell is sensed as the next storage level, generating a soft error.

By using the assumption that $\log_{10} R$ and α follow normal distributions as shown in Table 4, we can calculate the probability of such soft error type. The detailed development of formula is presented in Section 6.2. As we show in later sections, the most error-prone state in 4LC PCM is the third storage level for the following reasons. Firstly, the fourth storage level (amorphous state) in the highest resistance range does not generate errors. Secondly, because α is proportional to R, the third storage level experiences the rapidest resistance drift among all levels. If we remove the third storage level, this will not only remove the errors generated by itself but also reduce most of the errors generated by the second storage level. For instance, the majority of errors generated by the second storage level occurs on the boundary between the second and the third storage levels, which can be avoided by not using the third storage level. Table 5 shows our design points for 3LC PCM.

Table 5: **Configuration Variables of Tri-Level-Cell (3LC) PCM When $t_0 = 1$ s.**

Storage Level	$\log_{10}(R)$		α	
	μ_R	σ_R	μ_α	σ_α
0	3.0	$\frac{1}{6}$	0.001	$0.4 \times \mu_\alpha$
1	4.0		0.02	
2	6.0		0.10	

Given the physical parameters in Table 5, we calculate the SER of 4LC PCM and 3LC PCM. Note that we use the analytical model discussed in Section 6.2 and present the results in Tables 6 and 7. Table 6 shows the SER of two intermediate storage levels of 4LC PCM based on time intervals since they were written while Table 7 shows the SER of the first two storage levels of 3LC PCM. For example, if a 3LC PCM cell is written to the second storage level at $t = 0$, the SER of the cell is 5.93×10^{-14} at $t = 2^{45}$. Note that we mark "too small" in the tables when Mathematica 8.0 outputs zero due to lack of precision. As Table 7 shows, there is no error in 3LC PCM up to 2^{34} seconds or more than 500 years. Because of such low SER, scrubbing will be unnecessary for 3LC PCM in the time range of interest. For the same reason, ECC or other similar techniques can be waived. In summary, the SER of 3LC PCM is even lower than that of DRAM. It does not require scrubbing nor ECC to achieve the satisfactory level of reliability. To further justify the use of 3LC PCM over 4LC PCM, we quantitatively compare and evaluate these two design options in the subsequent sections.

Table 6: **Probability of Soft Error of Four-Level-Cell (4LC) PCM by Equation (4) in Section 6.2**

Time (s)	Storage Level 1	Storage Level 2
2	(too small)	5.85E-06%
2^2	1.59E-12%	0.02%
2^3	5.85E-06%	0.12%
2^4	7.45E-04%	0.28%

Table 7: **Probability of Soft Error of Tri-Level-Cell (3LC) PCM by Equation (4) in Section 6.2**

Time (s)	Crystalline State	Intermediate State
$2 \sim 2^{34}$	(too small)	(too small)
2^{35}	2.28E-16%	(too small)
2^{40}	1.59E-14%	(too small)
2^{45}	5.71E-10%	5.93E-14%

6.2 Analytical Error Model and Validation

In building an analytical error model for both 4LC PCM and 3LC PCM, we continue discussion on top of Table 4 and Table 5. First, we define two more variables, $m = \log_{10} R$ and $n = \log_{10} t$. By substituting Equation (1) with m and n, we obtain the following.

$$\log_{10}(R_{drift}(t)) = \log_{10} R + \alpha \log_{10} t = m + n\alpha$$

Thus, the condition of a soft error can be rewritten as follows.

$$m + n\alpha > \mu_R + E$$

$$n\alpha > \mu_R + E - m,$$

$$\text{where } E = \begin{cases} 0.5 \text{ for storage level 0, 1, and 2 of 4LC PCM} \\ 0.5 \text{ for storage level 0 of 3LC PCM} \\ 1.5 \text{ for storage level 1 of 3LC PCM} \end{cases}$$

As α follows $N(\mu_\alpha, \sigma_\alpha^2)$, $n\alpha$ follows $N(n\mu_\alpha, (n\sigma_\alpha)^2)$. The probability for $n\alpha$ to be more than $\mu_R + E - m$ can be calculated as follows.

$$\text{Probability of soft error for a given } m = 1 - \Phi\left(\frac{\mu_R + E - m - n\mu_\alpha}{n\sigma_\alpha}\right), \quad \text{where } \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-x^2/2} dx \tag{3}$$

Here, we also take the effect of the iterative writing into account. As mentioned earlier, cell programming iterates a write-and-verify sequence until $\log_{10} R$ is less than $\mu_R + 2.75\sigma_R$ or larger than $\mu_R - 2.75\sigma_R$. It means the probability density function of a random variable

m, $f(m)$ is as follows.

$$f(m) = \begin{cases} \frac{1}{K}\phi(\frac{m-\mu_R}{\sigma_R}) & \mu_R - 2.75\sigma_R < m < \mu_R + 2.75\sigma_R \\ 0 & \text{otherwise,} \end{cases}$$

$$\text{where } K = \int_{\mu_R+2.75\sigma_R}^{\mu_R-2.75\sigma_R} \phi(\frac{m-\mu_R}{\sigma_R})dm, \text{ and } \phi(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$$

Therefore, we can obtain the probability of soft error as a function of time ($t = 10^n$) by integrating Equation (3) with a random variable m for $\mu_R - 2.75\sigma_R < m < \mu_R + 2.75\sigma_R$.

$$\text{Probability of soft error} = \int_{\mu_R+2.75\sigma_R}^{\mu_R-2.75\sigma_R} (1 - \Phi(\frac{\mu_R + E - m - n\mu_\alpha}{n\sigma_\alpha}))f(m)dm \quad (4)$$

We evaluate Equation (4) for 4LC PCM and also run Monte Carlo simulations to verify these equations. In the simulation, we randomly picked R and α from their corresponding normal distributions in Table 4 and calculate the drift resistance, $R_{drift}(t)$, to determine if it generates any soft error. For each storage level, the simulator executes one billion trials. Figure 37 shows the results side by side. We omit the soft error rates for set and reset states, *i.e.,* the storage level 0 and 3, because (i) resistance drift in level-3 states does not lead to a soft error, and (ii) the error rates of level-0 states are too small to be evaluated and can be ignored. For example, Mathematica 8.0 shows the first non-zero error rate for level-0 states when $t = 2^{35}$ or 1090 years, and the error rate is 2.3×10^{-18}. Similarly, note that three data points for storage level 1 and 2 are missing because either (i) Mathematica 8.0 returns zero for Equation (4) or (ii) Monte Carlo simulation found no error in one billion trials. By comparing results from two independent sources, we validate the accuracy of our theoretically derived Equation (4) by simulation.

6.3 Revisiting Four-Level-Cell (4LC) PCM

Given the soft error rates in Table 6, it is clear that without any mechanism for reducing the soft error rates, 4LC PCM is unusable as main memory. Researchers have proposed several drift-tolerant approaches such as error correction schemes [46, 50, 19, 21], data encoding schemes using relative resistance difference [19, 50], a reference cell scheme [20],

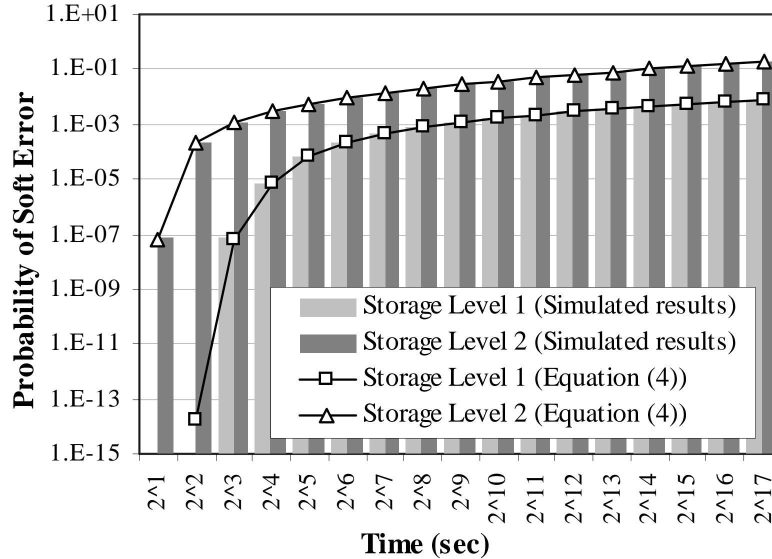


Figure 37: **Probability of Soft Error of Four-Level-Cell (4LC) PCM Over Time**

a time-aware drift estimation mechanism [21], and most recently an efficient scrubbing scheme [46]. Among them, we focus on the most recent work by Awasthi *et al.* [46] who described an architectural mechanism combining a memory scrubbing scheme with a strong error-correction method to lower soft error rates aiming to use PCM as main memory. However, as we will show, even with the most efficient scrubbing mechanism, the soft error rate of 4LC PCM is still much higher than that of DRAM.

6.3.1 Estimating Scrubbing Overhead

In this section, we compare the SER of 4LC PCM to that of contemporary DRAM and argue that 4LC PCM is impractical for main memory due to reliability concern. First, we assume a 16GB PCM main memory with eight banks (*i.e.,* 2GB per bank) using a 256B data block[2] as a basic access unit as assumed in prior literature [51, 52]. According to recent work by Choi *et al.* [53], the read and write latencies in SLC PCM are 120*ns* and 150*ns*, respectively. Considering iterative write-and-verify steps are required for MLC

[2] A large last-level DRAM cache is typically used to compensate for the relatively slower PCM access latencies. Its cacheline size is assumed to be 256B

PCM, we assume that scrubbing one cacheline takes at least $1.15\mu s$. Also, we assume that each storage level has the same probability of occurrences.

The first column in Table 8 shows the scrubbing overhead decreases as the scrubbing period increases. Here, the scrubbing overhead is defined as $\frac{\text{Time used for scrubbing}}{\text{Scrubbing period}}$. A 2GB PCM bank has 8M cachelines. Thus, about 9.65 seconds ($\simeq 8 \times 2^{20} \times 1.15\mu s$) are required for scrubbing the entire physical PCM even if the eight PCM banks are scrubbed in parallel. As shown in Table 6, even when the memory controller performs nothing but scrubbing (100% overhead, *i.e.,* the memory controller will not have time to respond to any memory request), the SER of storage level 2 in 4LC PCM is 0.12% which is significantly higher than that of DRAM. Moreover, if we use the scrubbing period of 45 minutes as in the DRAM memory system for real servers [47], the SER of a PCM cell programmed to storage level 2 will escalate to 5%, which is intolerable. Clearly, 4LC PCM with scrubbing mechanisms cannot guarantee the most basic reliability by any standard. To reach a very low SER and reduce the scrubbing overhead simultaneously, the maximum PCM capacity per bank must be limited. Our next section will show the largest capacity of 4LC PCM the scrubbing mechanism can support for different combinations of target SER and scrubbing overhead.

6.3.2 Reducing Capacity to Achieve Low Soft Error Rates

Another way of lowering SER of 4LC PCM is to limit the maximum capacity. We assume the capacity of 4LC PCM as 2GB per bank in Section 6.3.1 when estimating the scrubbing overhead. Because the scrubbing overhead proportionally increases with the capacity, assuming 1GB per bank of capacity results in halving the overhead. If we further reduce the capacity, 4LC PCM can achieve lower SER. Table 9 shows the results.

In Table 9, we calculate the maximum capacity of 4LC PCM for different combinations of SER and scrubbing overhead. The leftmost column represents the scrubbing period for each 256B memory block. The next column represents the combined SER, which is an average of SER of all four states in 4LC PCM. However, because the third storage level

Table 8: **Probability of Uncorrectable Errors by ECC and $SER_{combined}$ for 2GB per Bank 4LC PCM**

Scrubbing Period (Overheads)	$SER_{combined}$	Probability of Uncorrectable Errors for 512 bits = $P_{error}(512b)$		
		No ECC	(72, 64)	BCH-8 (512b+80b)
2^3 seconds (100+%)	0.030%	7.4%	0.05%	(too small)
2^4 seconds (60.29%)	0.070%	16.4%	0.24%	1.44E-10%
2^5 seconds (30.15%)	0.133%	28.9%	0.86%	3.80E-8%
2^6 seconds (15.07%)	0.218%	42.8%	2.26%	2.64E-6%
2^7 seconds (7.54%)	0.325%	56.5%	4.84%	7.45E-5%
2^8 seconds (3.77%)	0.475%	70.4%	9.76%	1.54E-3%
2^9 seconds (1.88%)	0.668%	82.0%	17.8%	0.02%
2^{10} seconds (0.94%)	0.91%	90.4%	29.4%	0.18%
2^{11} seconds (0.47%)	1.21%	95.6%	44.2%	1.08%
2^{12} seconds (0.24%)	1.57%	98.3%	60.6%	4.61%

Scrubbing Period (Overheads)	$SER_{combined}$	Probability of Uncorrectable Errors for 512 bits = $P_{error}(512b)$		
		BCH-16 (512b+160b)	BCH-24 (512b+240b)	BCH-32 (512b+320b)
2^3 seconds (100+%)	0.030%	(too small)	(too small)	(too small)
2^4 seconds (60.29%)	0.070%	(too small)	(too small)	(too small)
2^5 seconds (30.15%)	0.133%	(too small)	(too small)	(too small)
2^6 seconds (15.07%)	0.218%	(too small)	(too small)	(too small)
2^7 seconds (7.54%)	0.325%	(too small)	(too small)	(too small)
2^8 seconds (3.77%)	0.475%	1.27E-10%	(too small)	(too small)
2^9 seconds (1.88%)	0.668%	2.32E-8%	4.11E-13%	(too small)
2^{10} seconds (0.94%)	0.91%	2.15E-6%	2.81E-12%	(too small)
2^{11} seconds (0.47%)	1.21%	1.10E-4%	1.34E-9%	(too small)
2^{12} seconds (0.24%)	1.57%	3.14E-3%	2.66E-7%	8.69E-12%

shows significantly larger SER than the other levels, this combined SER is close to one fourth of the third storage level's SER. In addition, we show the maximum capacity by each given scrubbing overhead. When the overhead is 100%, the memory controller cannot service any request from the upper memory hierarchy. Since 100% scrubbing overhead is impractical, the third column of Table 9 can be viewed as an upper bound.

Table 9 also shows the maximum capacity when the scrubbing overhead are set to 12.5% and 1.0%, respectively. For example, if we design 4LC PCM with the scrubbing

Table 9: **Maximum Capacity Per Bank of Four-Level-Cell (4LC) PCM by Soft Error Rates and Scrubbing Overhead**

Scrubbing Period (s)	$SER_{combined}$	Scrubbing Overhead 100.0%	12.5%	1.0%
2	1.46E-06%	488MB	61.0MB	4.88MB
2^2	0.005%	977MB	122MB	9.77MB
2^3	0.030%	1.95GB	244MB	19.5MB
2^4	0.071%	3.91GB	488MB	39.1MB
2^5	0.132%	7.81GB	977MB	78.1MB

overhead of 1.0%, leaving 99% of the time for servicing memory requests, the maximum PCM capacity will be merely 4.88MB for achieving an average of $1.46 \times 10^{-6}\%$ SER. Note that when 4LC PCM comprises multiple ranks or banks, scrubbing can be performed in parallel. Thus, when one bank is being scrubbed, the other banks can respond to requests from the CPU. However, even with eight banks, the maximum capacity amounts to 39.1MB, which is still substantially below the main memory capacity required in modern computing systems. In sum, although a lower SER can be achieved by reducing the total capacity of 4LC PCM, the memory capacity becomes too small to be useful.

6.3.3 Using Error-Correcting Codes

Error-correcting codes (ECC) can be applied to compensate the SER of 4LC PCM. For example, the industry standard (72,64) Hamming code [35] can correct single bit errors by adding 8 redundant bits on top of 64 bits data.[3] This scheme is commonly found in main memory of server systems because of the simplicity in encoding and decoding [54]. Moreover, stronger ECC can also be used to protect data from multiple bit errors. For example, BCH codes [55, 56] correct 8, 16, 24, or 40 bits errors from 256, 512, 1024 bytes of data depending on the size of the redundant bits. Because decoding BCH codes require more computing power and time than (72,64) Hamming code, these codes are not frequently used for latency-sensitive devices such as main memory but commonly found in

[3]The capacity overhead is 12.5%.

slower devices such as NAND-based storage. With the combined SER for each cell of 4LC PCM developed in previous sections, we calculate the error rates after applying (72,64) Hamming code and various BCH codes. Note that for every ECC evaluated in this section, we fix the data size at 512 bits as in [46].

(72,64) Hamming code corrects one bit error, and thus, having more than two bit errors among 72 bits is uncorrectable. In addition, since storing 72 bits requires 36 4LC PCM cells, the probability of having more than two bit errors out of 36 cells can be calculated as follows. Note that by using Gray codes as described in Table 4, one step change in storage levels is limited to affect only one bit in two-bit data. Thus, two bit errors can happen only when two 4LC PCM cells are changed due to resistance drift.

$$\begin{aligned}
&\text{Probability of having at least two bit errors in 72 bits} \\
&= 1 - P(\text{no errors}) - P(\text{one bit error}) \\
&= 1 - (1 - SER_{combined})^{36} - \binom{36}{1}(1 - SER_{combined})^{35}(SER_{combined}) \qquad (5) \\
&= P_{error}(72b)
\end{aligned}$$

Now we calculate the probability of uncorrectable errors in 512 bits. 512 bits comprises eight of 64 bits data, therefore, to reconstruct the entire 512 bits, all eight blocks should not generate any uncorrectable error. If we define the result of Equation (5) as $P_{error}(72b)$, then the probability of uncorrectable error for 512 bits is defined as follows.

$$P_{error}(512b) = 1 - (1 - P_{error}(72b))^8$$

The fourth column in Table 8 shows the results. In Table 8, we calculate the probability of uncorrectable errors by scrubbing period, scrubbing overheads, and $SER_{combined}$. If we compare the error rates of 4LC PCM to that without ECC, (72,64) Hamming code reduces the error rates, but those rates are still too high for practical use. The results indicate that 4LC PCM must use stronger ECC that requires more redundant bits and higher computational overheads.

Now we calculate the probability of uncorrectable errors with stronger ECC. On top of 512 bits of data, BCH-8 corrects up to 8 bits errors by adding 80 redundant bits, and BCH-16 corrects up to 16 bits errors by adding 160 redundant bits.[4] We generalize Equation (5) for calculating the probability of having at least *n* bit errors out of *m* bits as follows.

$$\text{Probability of having at least } n \text{ bit errors out of } m \text{ bits} = 1 - \sum_{k=0}^{n-1} \binom{m}{k} (1 - SER_{combined})^{m-k} (SER_{combined})^{k} \tag{6}$$

Table 8 also shows the results from Equation (6). When the scrubbing period is 2^8 seconds, the scrubbing overhead is 3.77%, and $P_{error}(512b)$ is $1.54 \times 10^{-3}\%$ for BCH-8. The error rate is significantly smaller than that of 4LC PCM with (72,64) Hamming code; however, still $10^6 \sim 10^7$ times higher than the SER of DRAM without ECC support.

Table 8 shows that if we limit the maximum scrubbing overhead to 1%, 4LC PCM is only usable with BCH-32. However, such requirement prevents 4LC PCM from being used as main memory of commodity systems because of the following reasons. Firstly, a memory controller with a complex error-correcting mechanism requires extra chip area and design effort, which increase the chip cost. Since memory controllers are typically integrated in the same die with processor cores, vendors need to design and fabricate a customized CPU for supporting 4LC PCM. Secondly, the higher computational overhead in decoding increases the memory latency and degrades the performance. For these reasons, the majority of commodity systems typically implement no ECC schemes or at most the (72,64) Hamming code.

Moreover, the most critical downside of BCH-32 is a capacity overhead. To correct up to 32 errors from 512 bits of data, we must add 320 parity bits to make a total of 832 bits of data. In storing 832 bits, 416 4LC PCM cells are needed. On the other hand, 416 3LC PCM cells can store 659 bits ($= \lfloor log_2(3^{416}) \rfloor$). Note that because 3LC PCM has no soft errors in the time range of interest, all 659 bits can be used to store useful information without parity

[4]The capacity overheads are 15.6% and 31.3%, respectively.

bits. In summary, 3LC PCM theoretically achieves higher information density than 4LC PCM. The next sections explains practical implementation for 3LC PCM.

6.4 Tri-Level-Cell (3LC) PCM in Practice

So far, we have discussed that by using 3LC PCM, we can achieve a confident level of reliability that 4LC PCM cannot provide. In this section, we will address 3LC PCM implementation issues.

6.4.1 Binary-to-Ternary Conversion

Since tri-level cells do not match any conventional binary digital system, we need an efficient way to convert binary information into the ternary number system and vice versa. The efficiency of number conversion methods can be evaluated with cell utilization and implementation feasibility. In other words, it is desirable if a number conversion method can fully utilize the cell capacity with low-cost hardware.

First, we need a way to evaluate the cell utilization of a number conversion method considering extra overhead. For example, if a conversion method uses $\frac{n}{2}$ four-level cells to store n-bit data, then the four-level cells can be regarded as fully utilized. On the contrary, if a conversion method needs n four-level cells to store n-bit data, then its cell utilization will be halved. This concept can be generalized as follows. When a number conversion method uses m k-level cells to store n-bit data, its cell utilization is

$$\text{Cell Utilization} = \frac{\log_k 2^n}{m} = \frac{n}{m} \log_k 2. \tag{7}$$

In this equation, 2^n is the number of different states represented by n-bit data and $\log_k 2^n$ is the theoretically minimum number of k-level cells to store n-bit data. For instance, if a 4LC PCM requires a BCH-32 error correction scheme to prevent drift-induced soft errors, the cell utilization of the binary to quaternary conversion is calculated as $\frac{512}{416} \log_4 2 \simeq 0.615$ where the BCH-8 requires 320 more parity bits to recover 512-bit data from eight errors.

As mentioned in Section 6.1, the 3LC PCM does not require any complicated error

correction scheme. However, some capacity loss of 3LC PCM is unavoidable due to binary-to-ternary conversion. In other words, when using n bits as a basic store unit, the minimum number of 3LC PCM cells to store the n bits is $\lceil n \log_3 2 \rceil (= m)$. For example, storing three-bit data requires at least two 3LC PCM cells. The two 3LC PCM cells are used for differentiating 2^3 states even though they can represent maximum 3^2 states. Thus, one of the states represented by two 3LC PCM cells is remained unused. Figure 38 shows achievable cell utilization for $\langle n, m \rangle$ binary-to-ternary conversion methods when varying the size of a basic store unit, n, from one to 32. Among those conversion methods, the $\langle 19, 12 \rangle$ conversion storing 19 bits to 12 3LC PCM cells can achieve the highest cell utilization of 0.999, while the cell utilization of a $\langle 8, 6 \rangle$ conversion method is at most 0.841. For reference, in the $\langle 1, 1 \rangle$ and $\langle 2, 2 \rangle$ conversion methods, a 3LC PCM cell acts like a SLC-PCM cell and the cell utilization is 0.631.

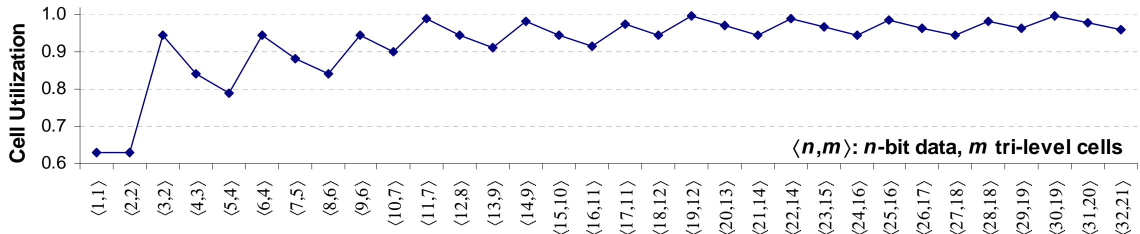


Figure 38: **Tri-Level-Cell Utilization**

Another factor that should be considered for a conversion method is how feasible the conversion method is to implement. Those $\langle n, m \rangle$ conversion methods for 3LC PCM can be implemented with several ways such as using a look-up table (LUT), calculating with arithmetic units, and implementing with basic logic gates. Those implementation methods have their respective pros and cons. Using a look-up table can reduce conversion latency, while the number of table entries is exponentially increased when increasing the size of a basic store unit, n. On the other hand, calculating ternary numbers with arithmetic units consumes less hardware costs than LUT but its latency is increased due to complicated

arithmetic operations. Thus, increasing a basic store unit, n, to achieve higher cell utilization results in high hardware cost or long access latency. Moreover, since a conversion method should be embedded inside a memory chip, its hardware cost and access latency are critical for its implementation feasibility.

Therefore, we propose to use a simple number mapping method such as $\langle 1, 1\rangle$, $\langle 2, 2\rangle$, and $\langle 3, 2\rangle$ conversion methods that are implementable with simple logic gates. With a combination of the three conversion methods, we can build any conversion method whose cell utilization is less than or equal to the cell utilization of $\langle 3, 2\rangle$, 0.946. For example, a $\langle 8, 6\rangle$ conversion can be composed of two $\langle 3, 2\rangle$ conversions and one $\langle 2, 2\rangle$ conversion, *i.e.,* $2\langle 3, 2\rangle+\langle 2, 2\rangle = \langle 8, 6\rangle$, and a $\langle 16, 11\rangle$ conversion can be composed of five $\langle 3, 2\rangle$ conversions and one $\langle 1, 1\rangle$ conversion, *i.e.,* $5\langle 3, 2\rangle + \langle 1, 1\rangle = \langle 16, 11\rangle$.

Table 10 shows an example of the $\langle 3, 2\rangle$ number mapping method. In this example, eight ternary states except the 11 state are used to represent three-bit binary data. This simple number mapping method can be implemented with several logic gates. Assume that three-bit data, $b_2b_1b_0$, is stored to two tri-level cells, t_1t_0 and each cell uses two control signals, p_{c1} and p_{c0}, to select a programming current corresponding to its state, where c indicates a corresponding cell number. If the relationship between the cell states and their control signals is shown in Table 10, the control signals can be represented with the three binary bits as follows.

$$\begin{aligned} p_{11} &= b_2 \cdot b_1 + b_2 \cdot b_0 \\ p_{10} &= b_2 + b_1 \cdot \overline{b_0} \\ p_{01} &= \overline{b_2} \cdot b_1 + b_1 \cdot \overline{b_0} \\ p_{00} &= b_1 + \overline{b_2} \cdot b_0 \end{aligned}$$

Similarly, when reading a 3LC PCM cell, its programmed resistance is represented with the outputs of two sense-amplifiers, r_{c1} and r_{c0} as described in Table 10. From the four outputs

of two cells, the three-bit data can be decoded with simple logic gates as follows.

$$b_2 = r_{11} + r_{10} \cdot \overline{r_{01}} \cdot \overline{r_{00}}$$

$$b_1 = r_{01} + r_{11} \cdot r_{00}$$

$$b_0 = r_{11} \cdot \overline{r_{01}} + \overline{r_{11}} \cdot \overline{r_{10}} \cdot r_{01} + \overline{r_{11}} \cdot \overline{r_{10}} \cdot r_{00}$$

Table 10: **An example of the $\langle 3, 2 \rangle$ number mapping method.**

3-digit binary	2-digit ternary	control signals			
		$cell_1$ for t_1		$cell_0$ for t_0	
$(b_2 b_1 b_0)$	$(t_1 t_0)$	s_{11}	s_{10}	s_{01}	s_{00}
000	00	0	0	0	0
001	01	0	0	0	1
010	12	0	1	1	x
011	02	0	0	1	x
100	10	0	1	0	0
101	20	1	x	0	0
110	22	1	x	1	x
111	21	1	x	0	1

Relationship between ternary levels and control signals:

⟨ Programming ⟩

t_c	p_{c1}	p_{c0}
2	1	x
1	0	1
0	0	0

⟨ Reading ⟩

t_c	r_{c1}	r_{c0}
2	1	x
1	0	1
0	0	0

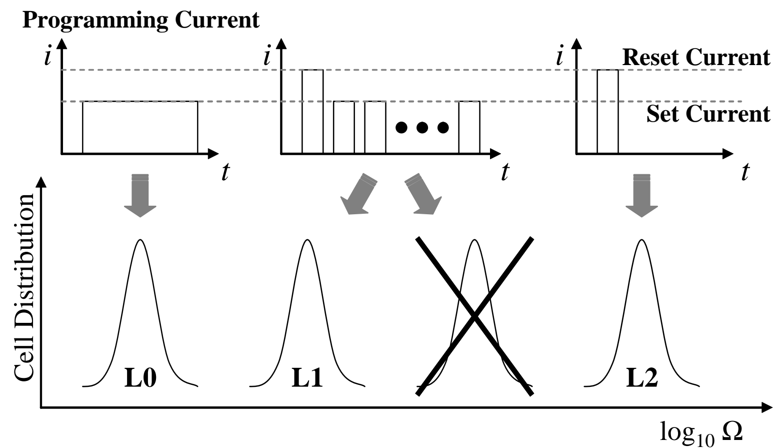
where “x” means redundant condition

In this section, we showed that by using a $\langle 3, 2 \rangle$ number mapping method we can achieve the cell utilization of up to 0.946 with low cost hardware. Thus, when using an $\langle 8, 6 \rangle$ conversion composed of two $\langle 3, 2 \rangle$ and one $\langle 2, 2 \rangle$ conversion methods, 512-bit data can be stored in 384 tri-level cells. Here, it is noteworthy that in the case of 4LC PCM, 416 cells are required to store 512-bit data when using a BCH 32-error correcting scheme to achieve a confident level of reliability.

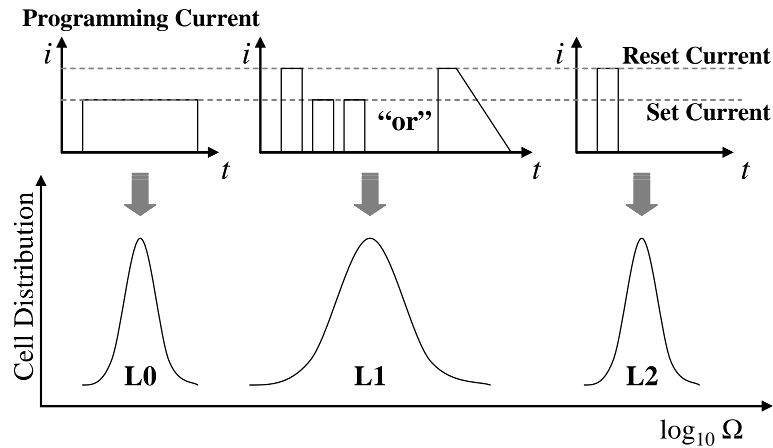
6.4.2 Bandwidth Enhancement

So far, we achieved the desired reliability with 3LC PCM by eliminating the most error-prone state from the four-level cell PCM as shown in Figure 39(a). To program a cell to the

intermediate state, “L1”, 3LC PCM use the same write-and-verify iterations of 4LC PCM. Since a prime concern in 4LC PCM is to maximize its reliability, it is desirable to precisely tune the resistance of intermediate levels to render drift margins between adjacent levels as large as possible. However, this precise programming leads to a long write latency, which is the root cause of low write bandwidth in MLC PCM. The question is whether the tight resistance ranges for the intermediate levels achieved by write-and-verify iterations are still necessary for 3LC PCM.



(a) 3LC PCM Programming



(b) Bandwidth-Enhanced (BE) 3LC PCM Programming

Figure 39: **Cell Distribution vs. Programming Sequence**

According to our analytical model, the 3LC PCM using the same resistance ranges with 4LC PCM is virtually free from drift-induced soft errors. As described in Table 7, its

SER is extremely small even comparing with that of DRAM, indicating that the resistance range for the intermediate level is unnecessarily tight. Since the tight resistance range is obtained by sacrificing the write latency, we can reduce the write latency by relaxing the resistance range and eventually improve the overall write bandwidth. Therefore, we propose bandwidth-enhanced 3LC (BE-3LC) PCM using a relaxed resistance range for the intermediate level.

Figure 39(b) shows two examples to program a relaxed intermediate level in 3LC PCM. The relaxed intermediate level can be programmed with less number of write iterations because of its widened resistance range. Another choice to program the relaxed intermediate level is to use the moderate-quenched (MQ) programming which controls the falling slope of a reset current pulse [57]. By using the MQ programming method, the write latency of an intermediate level in 3LC PCM cell can be reduced below the set latency, *i.e.,* the write latency of SLC PCM.

As mentioned, relaxing the acceptable resistance range for the intermediate level helps to reduce the write latency and enhance write bandwidth. However, it reduces the drift margin between resistance levels and the narrow margin causes the drift-induced SER to be somewhat increased. In Section 6.4.3, we will introduce how to use conventional ECC schemes for the slightly increased SER of BE-3LC PCM. Also, we will evaluate the SERs of both 3LC PCM and BE-3LC PCM in Section 6.5.1.

6.4.3 Efficient $\langle 3, 2 \rangle$ Conversion for Error Correction

Using error correcting codes can improve the 3LC PCM reliability as in other memory systems. The problem is how efficiently 3LC PCM uses the conventional ECC schemes. In the case of 4LC PCM, four states of a cell is encoded with two-bit Gray code. By doing so, one state transition in a four-level cell affects only one bit in binary data, which enables to use a binary error-correcting code for correcting the state transition of four-level cells. Similarly, if one drift-induced error in a tri-level cell affects only one bit in the corresponding binary code, a binary error correcting code can be used for recovering the

data from the drift-induced error.

In this section, we propose a state mapping method of $\langle 3,2\rangle$ conversion for using binary error correcting codes. Figure 40(a) shows possible state transitions caused by drift-induced errors in two 3LC PCM cells. Because the resistance drift increases the resistance level of a PCM cell, *i.e.,* from level 0 to level 1 or from level 1 to level 2, the state transitions are uni-directional. The main idea is to map the state transition graph of two 3LC PCM cells into the transition graph of the three-bit Gray code depicted in Figure 40(b).

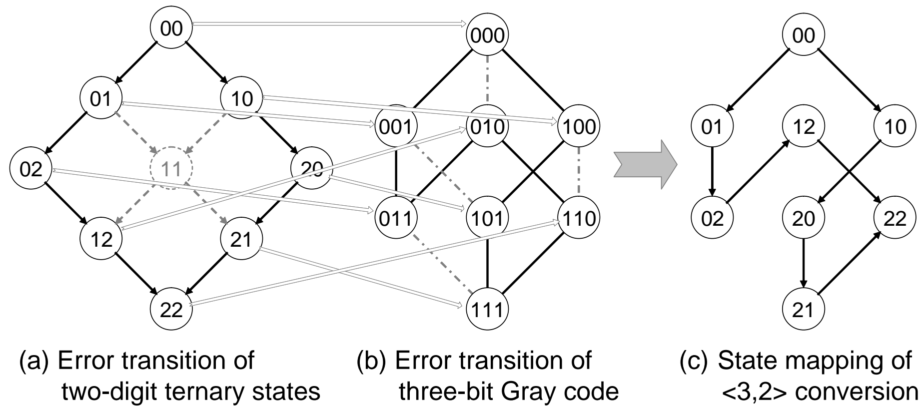


Figure 40: **State mapping of $\langle 3,2\rangle$ conversion for efficient error correction in 3LC PCM**

First, we exclude “11” state from the state mapping of our $\langle 3,2\rangle$ conversion. Note that the state “11” was excluded because it has four transition edges, which cannot be mapped into the Gray code, and also two tri-level cells can represent one more state than a three-bit binary code. Then, the rest of states and edges are mapped into the Gray code graph as shown in Figure 40(c),[5] which means that all one-hop error transitions of the two-ternary-cell states except ones from/to the “11” state are represented with one-hop error transitions of the three-bit binary code. Note that we need a special process for the “11” state because removing the “11” state from the state mapping cannot prevent error transitions to the “11” state. When the “11” state is read from two 3LC-PCM cells, it indicates that the state results from one or more drift-induced errors. Also, considering the monotonic increase property of resistance drift, only “00”, “01”, and “10” states can be shifted to the “11” state.

[5]This state mapping is the same as one in Table 10.

Thus, when the "11" state is read from the two tri-level cells, we can limit the maximum number of transition-error hops to one by substituting it with a "00" state. By doing so, one state transition error caused by resistance drift affects only one data bit. For example, let's assume that (72,64) Hamming code is used for single error correction and double error detection (SECDED). The 72-bit code can be stored in 48 3LC PCM cells when using $\langle 3, 2\rangle$ conversion. With the state mapping of $\langle 3, 2\rangle$ conversion, the (72,64) Hamming code can detect two drift-induced errors in the 48 tri-level cells and can correct one drift-induced error.

Furthermore, considering a 72-bit PCM DIMM composed of 8 PCM chips, each PCM chip has a 9-bit datapath which are matched to three $\langle 3, 2\rangle$ conversion units. Note that if eight 8-bit PCM chips are used to compose a 64-bit PCM DIMM for symmetry, each chip becomes to use $\langle 8, 6\rangle$ conversion. As a result, the 64-bit data is stored to 48 3LC PCM cells which is the same amount of 3LC PCM cells to store a 72-bit code. Therefore, given a real PCM DIMM organization, our $\langle 3, 2\rangle$ state mapping method allows to use the (72,64) Hamming code without additional storage overhead.

6.5 Evaluation

6.5.1 Soft Error Rate of BE-3LC PCM

As discussed in Section 6.4.2, 3LC PCM can reduce writing latency by using fewer writing iterations. As such, the distribution of the resistance is compromised, and which will increase the SER of the PCM cell. In this section, we formulate the relationship between writing latency and the SER of 3LC PCM and argue that 3LC PCM can achieve the writing latency close to SLC-PCM without compromising the SER.

Kang *et al.* [57] shows the distribution of the resistance of a PCM cell by two different writing strategies; (i) iterative writing (write and verify) and (ii) writing without iterations. As 3LC PCM does not use the third storage level, we focus on the distribution of the second storage level. More specifically, we read the distribution of the resistance of the second storage level from Figure 1 based on [57] and calculate the mean and the variance of the

resistance when a cell is written without iterations. As we show in Table 11, σ_R and μ_α are worsened from 4.0 to 4.255 and from 0.167 to 0.188, respectively. In addition, we assume linear increment in μ_α and σ_α by σ_R for estimating the distribution of α. For example, we use the physical parameters in Table 4 and apply 0.04 increment in μ_α for every 10x in R.

Table 11: **Physical Parameters for the Second Storage Level of 3LC PCM When t_0 = 1 s.**

Writing Strategy	$\log_{10}(R)$		α	
	μ_R	σ_R	μ_α	σ_α
Iterative	4.0	0.167	0.02	$0.4 \times \mu_\alpha$
Non-iterative	4.255	0.188	0.02157	

After obtaining the physical parameters of the second storage level of 3LC PCM, we calculate the SER of 3LC PCM by using analytical models discussed in Section 6.2. The summary of results is as follows. Firstly, the majority of the errors happen in between the set state and the second storage level. Such errors are not due to the resistance drift, but because of the initial writing failure. For example, the memory controller writes 01 to a 3LC PCM cell, and the cell reads 00 immediately after the writing. The error rate for this case is $3.04 \times 10^{-3}\%$. Secondly, if we exclude such initial writing failures, the SER of 3LC PCM caused by resistance drift is negligible until $t = 2^{20}$ seconds. Table 12 shows the error rate for this case.

When a PCM chip reads PCM cells immediately after writing them, the chip can detect and rewrite the cells to fix the initial writing failures. More specifically, we assume that the PCM chips rewrite the cells by sensing the written values immediately after writing. Even though such strategy is similar to the iterative writing commonly used in 4LC-PCM, this strategy is different in terms of the expected numbers of iterations. For the $(100 - 3.04 \times 10^{-3})\%$ of the time, writing to our proposed BE-3LC-PCM finishes at the first attempt. The second attempt is required for only $3.04 \times 10^{-3}\%$ of the time, and the expected numbers of writing iterations in this case is close to one. Moreover, we also show the SER of 3LC-PCM with industry standard (72,64) ECC support in the rightmost column of Table 12. This

column is calculated based on the fact that 48 3LC-PCM cells store 72 bits, and (72,64) ECC corrects one bit error. Again, the proposed PCM with (72,64) ECC shows a negligible SER until $t = 2^{25}$ seconds. In summary, writing to 3LC PCM cells must be followed by reading and verifying. Such small overhead will remove majority of the errors, and 3LC PCM experiences no errors in the time range of our interest.

Table 12: **Soft Error Rates of Intermediate Storage Level of BE-Three-Level-Cell (BE-3LC) PCM**

Scrubbing Period (s)	Iterative Writing	BE Writing	BE Writing +(72, 64) ECC
2^5	(too small)	(too small)	(too small)
2^{10}		(too small)	(too small)
2^{15}		(too small)	(too small)
2^{20}		3.60E-16%	(too small)
2^{25}		1.28E-10%	2.66E-15%

6.5.2 Performance

4LC PCM requires a scrubbing mechanism and a multiple-error correction scheme for a confident level of reliability. However, to use the 4LC PCM, we have to consider other aspects such as performance and hardware overhead. If the gain from its high density needs other considerable cost, the 4LC PCM will be regarded as infeasible. First, to evaluate the performance impact of using MLC PCM, we simulated 26 applications from SPEC2006 benchmark using SESC [28]. The read and write latencies of SLC PCM are assumed to be 150ns including a row activation latency (tRC) of 120ns, and 200ns considering an internal write verification delay, respectively [53]. For 3LC and 4LC PCMs, its read latency is the same with the SLC's, while its write latency is assumed to be 1000ns because of its iterative write-and-verify iterations [46]. Similar to other studies [2, 9], an 8MB L3 DRAM cache composed of 256B cache-lines is employed to hide the PCM access latency. Also, we assumed a PCM main memory composed of eight 2GB banks and we modeled a memory controller that can efficiently schedule memory requests by exploiting bank-level parallelism and PCM row buffer hits. Note that in the request scheduling, read requests

have higher priority than write requests because write accesses are typically not on the critical path in terms of performance.

Figure 41 shows the relative instruction-per-cycle (IPC) values normalized to the IPC of SLC PCM. As in the recent paper [46], the two 4LC PCM configurations are assumed to use a BCH code capable of eight error corrections for a 512-bit data block.[6] According to our analysis, the 4LC PCM with the BCH code has to scrub the entire memory space every eight seconds to achieve a DRAM-level soft error rate. However, the eight-second scrubbing is impossible because the minimum latency to scrub a 2GB PCM bank is about 9.6 seconds. Thus, we chose a 16 second scrubbing for 4LC PCM. Awasthi *et al.* proposed a scrubbing overhead reduction scheme called *Light Array Read for Drift Detection* (LARDD) [46]. However, since LARDD reduces scrubbing overhead by sacrificing reliability, we assume an 8-second period for the LARDD scheme. As shown in Figure 41, the 4LC PCM scrubbed every 16 seconds experienced 72.2% performance degradation on average. Especially, 429.mcf that shows the highest write frequency (2.81 per 1000 instructions) incurred 95.2% performance degradataion. This is because of the five times longer write latency of 4LC PCM and its scrubbing overhead occupying 60.0% of total execution time. This tremendous performance degradation can be reduced by employing the LARDD scheme. However, the LARDD still experienced 26.7% performance degradation. This means that although LARDD reduces the write frequency to PCM, there are still too many read-and-check operations performed inside a chip, leading to substantial performance degradation. On the other hand, the 3LC PCM experienced only 10.4% performance degradation on average, although its write latency is also 1000 ns as in 4LC PCM.

Furthermore, the performance of 3LC PCM can be improved by using the bandwidth-enhanced (BE) 3LC. Figure 42 shows the relative IPC of BE-3LC PCM which is normalized to the IPC value of SLC PCM. The write latency of BE-3LC PCM is obviously

[6]We assumed the encoding and decoding latencies of the eight-error-correction BCH code take one memory clock cycle because the encoding and decoding logic can be fully parallelized by accepting its exponentially increased area overhead.

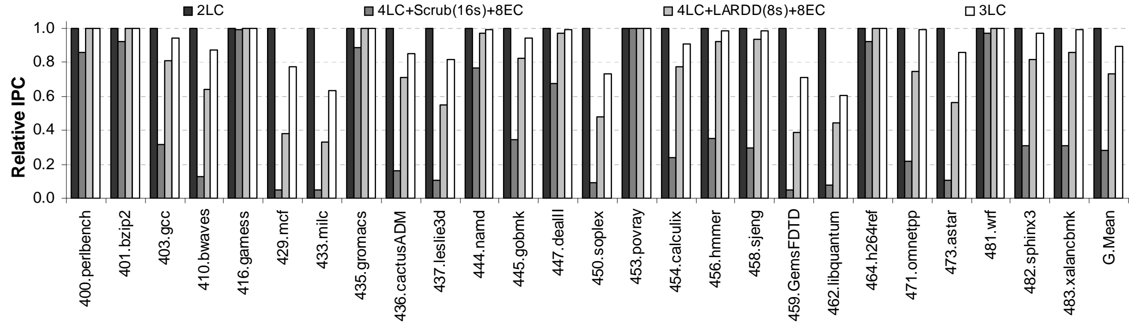


Figure 41: **Performance comparison with 4LC and 3LC**

decreased close to SLC PCM's latency, however, estimating the accurate write latency is beyond this research scope. Thus, we performed a sensitivity study varying its write latency from 350ns down to 200ns monotonically decremented by a 50ns interval. In addition, the SER of BE-3LC is confined to less than 3.6×10^{-18} when the BE-3LC is scrubbed every 2^{20} seconds, as mentioned in Section 6.5.1. Thus, all the BE-3LC PCM configurations are assumed to use a 2^{20} second scrubbing scheme. The relative IPC of the four configurations are 0.982, 0.988, 0.994, and 1.000, respectively. As a result, BE-3LC PCM makes it feasible to achieve the increment of memory capacity with negligible performance degradation, compared with SLC PCM.

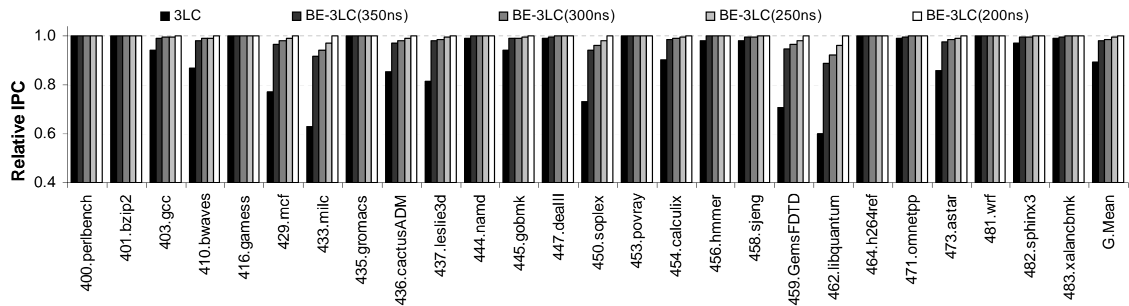


Figure 42: **Sensitivity study of bandwidth-enhanced 3LC**

6.5.3 Information Density

Another way to reinforce 4LC PCM reliability is to increase the number of correctable errors in a data block. However, if the number of additional cells required for a multiple error

correction code is equal to or larger than the number of data cells, then it is meaningless to use 4LC PCM. For example, since the BCH code correcting nine errors in a 64-bit data block requires additional 64 bits, a total of 64 4LC PCM cells should be used to store the 128 bits. Then, the 4LC PCM using a nine-bit correction (128,64) BCH code is no better than using SLC PCM.

Here, we define *information density* as the number of data bits stored in one cell to measure the cell efficiency. For instance, information density of SLC PCM is 1.00 because every SLC PCM cell stores one data bit, and SLC PCM does not require capacity overheads from ECC. In the case of 3LC PCM, it uses an $\langle 8, 6 \rangle$ conversion scheme and thus its information density is $\frac{8}{6} \simeq 1.33$. In the proposed BE-3LC PCM, a (72,64) hamming code is stored to 48 cells. Thus, its information density is still $\frac{64}{48} \simeq 1.33$.

In Figure 43, we compare the information density of 4LC PCM with SLC PCM and our proposed 3LC PCM. For example, the eight-bit correction (592,512) BCH code uses $\frac{592}{2}$ cells to store a 512-bit data block and its information density is 1.73. However, the 4LC PCM using a (592,512) BCH code requires an eight-second scrubbing scheme to achieve confident reliability, which seriously degrades performance as discussed in Section 6.5.2. If we use a strong error correction code recovering a data block from more errors, we can reduce the scrubbing frequency and diminish its performance degradation caused by scrubbing operations. Thus, we evaluate the SER of each configuration when a scrubbing period is 2^{10} seconds. Because it spends 9.65 seconds to scrub all 256B memory lines in a 2GB PCM bank, the maximum performance degradation caused by scrubbing operations can be limited to less than 1.00% ($> \frac{9.65}{2^{10}}$). According to our analytical model, when the size of a data block is 512 bits, a 26 or more error correction scheme is required to achieve the same level of SER with the proposed BE-3LC in Table 12. When using a 256-bit data block, an error correcting scheme has to be able to correct 20 or more errors. As shown in Figure 43, those configurations marked in rectangles have lower information density than that of 3LC PCM, 1.33. In other words, 3LC PCM is more efficient than 4LC PCM to

store data bits at the same level of reliability.

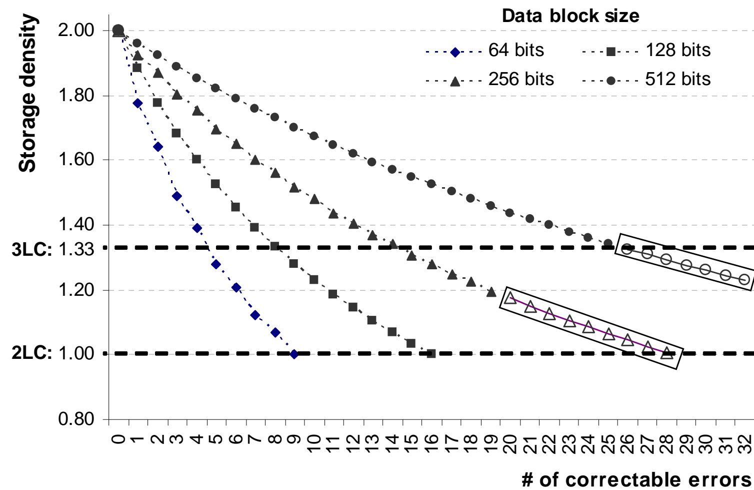


Figure 43: **Information density of 4LC PCM**

6.6 Summary

In this chapter, we asked the question about how reliable the widely studied four-level-cell (4LC) PCM can be by exploiting the schemes aimed at overcoming resistance drift problems. We modeled the resistance drift in MLC PCM and showed that conventional ECC schemes and scrubbing mechanisms are not usable in 4LC PCM for minimizing drift-induced soft errors to a tolerable level for reliability due to their unduly overheads and certain physical limit. We then evaluated architectural approaches addressing drift issues in 4LC PCM including efficient scrubbing mechanisms and multiple error correction schemes. To achieve a confident level of reliability, however, the latest scrubbing mechanism still incurs significant performance degradation of 26.7% compared to 2LC PCM. On the other hand, when using a stronger error correction code for correcting multiple errors, the performance impact of the scrubbing mechanism could be alleviated but the increase of codeword length compromises *information density*, *i.e.,* the number of data bits stored in each cell, to lower than 1.33.

Considering these shortcomings, it is premature to use the 4LC PCM for an efficient

and reliable memory system. Therefore, we propose tri-level-cell (3LC) PCM by removing the most drift-error-prone level from 4LC PCM. This new technology can eliminate the reliability concerns due to drift-induced errors. Furthermore, by relaxing an acceptable resistance range of the intermediate level, the programming latency of 3LC PCM can be reduced close to that of 2LC PCM, making the performance impact negligible. Also, we propose a state-mapping $\langle 3,2 \rangle$ conversion to efficiently store binary data to tri-level (ternary) cells. The state-mapping $\langle 3,2 \rangle$ conversion scheme can be implemented with simple logic gates. Another merit of the state-mapping scheme is that it enables a conventional binary ECC scheme such as a $(72,64)$ Hamming code to be used for correcting a ternary cell error while maintaining its information density to at least 1.33. In sum, by using 3LC PCM, we can obtain benefits from the increasing memory capacity without any concerns about memory reliability as well as without performance degradation.

CHAPTER 7

CONCLUSIONS

In this dissertation, we addressed reliability issues of phase-change memory, such as the limited write endurance and the resistance drift. Those issues are the most critical to use PCM as a main memory. To overcome those issues, we propose various architectural solutions and compare them with prior schemes in various aspects such as reliability, security, performance, feasibility, etc. This dissertation includes the following contributions.

- A secure wear-leveling scheme dynamically changing address mapping.
- A multiple stuck-at-fault error correction scheme.
- A hybrid memory architecture using multi-dimensional classification to detect and isolate malicious writes.
- Tri-level-cell phase-change memory as a practical use of multi-level cell PCM.

The first approach used to overcome the limited write endurance is a wear-leveling scheme which evenly wears out the entire memory space to extend PCM lifetime. However, we found that if the memory mapping used by a wear-leveling scheme is leaked to an adversary, malicious code can be easily designed to accelerate PCM cell aging and fail the PCM main memory. Ironically, PCM's relatively fast access time can be used to reduce the attack time to fail the memory. Thus, we propose a secure, low-cost wear-leveling scheme, *security refresh*, which can dynamically change memory address mapping. For address mapping, security refresh uses an algebraic function, which uses much less hardware overhead than other table-based wear-leveling schemes. Since the algebraic function periodically changes random keys, security refresh can effectively obfuscate address mapping information. Another finding is that the recursive use of security refresh can extend PCM lifetime further under malicious attacks even with less remapping overhead. The evaluation shows that two-level security refresh endures more than five years under malicious attacks with less than 2.0% remapping overhead.

Secondly, we focused on the fact that PCM cell wear-outs eventually incur stuck-at faults. Thus, without any error recovery schemes, the weakest endurance cell dictates the PCM lifetime. Moreover, multiple error recovery schemes are desirable considering PCM cell endurance variation that may increase as technology scales. To provide a stable PCM lifetime, we proposed a multiple error correction scheme, SAFER, specialized for treating stuck-at-fault errors. Different from other ECC schemes originally devised for correcting transient errors, SAFER efficiently recovers data from multiple stuck-at faults by using the properties of stuck-at faults such as permanency and readability. SAFER dynamically partitions a data block into multiple groups ensuring that each group has at most one stuck-at fault, and then applies a data-inversion scheme to each group as a single-error-correction scheme. By doing so, SAFER32 which can correct at least 6 errors shows better lifetime improvement than an eight-error correction Hamming code even with less storage overhead.

The third approach to address the limited write endurance is to efficiently detect malicious writes and isolate them. To do so, we proposed a hybrid memory architecture that integrates a small SRAM called isolation cache with a detection mechanism. For the detection mechanism, we also proposed a multi-dimensional classification. In the mechanism, the overall operation of each dimension is similar to a counting Bloom filter but its counters indicate not write frequency itself but the degree of deviation of the write frequency. Thus, temporarily concentrated write addresses are detected and isolated to the isolation cache. Another merit of this scheme is to make wear-leveling more efficient by detecting abnormal write behavior and forcing malicious code to use more attack targets than the number of isolation cache entries.

The last contribution of this dissertation is to evaluate the reliability of multi-level-cell (MLC) PCM exploiting prior schemes to overcome resistance drift issues. According to the evaluation, the bit error rate of four-level-cell (4LC) PCM achieved by the prior schemes are still much higher than that of DRAM. To achieve a DRAM-level bit error rate in 4LC PCM,

the benefit from the capacity increase of multi-level cells will be offset due to its storage and performance overhead. Thus, we proposed to use tri-level-cell (3LC) PCM and showed that 3LC PCM is a more feasible option than 4LC PCM when considering additional overhead and information density as well as the reliability. Also, for the practical use of 3LC PCM, we proposed a state-mapping conversion scheme to efficiently store binary data to tri-level cells. The $\langle 3, 2 \rangle$ state-mapping conversion using two tri-level cells to store three bits can achieve at least 1.33 of information density. In addition, we showed that when using the $\langle 3, 2 \rangle$ state-mapping, a (72,64) Hamming code can further increase the reliability of 3LC PCM without any storage overhead.

REFERENCES

[1] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *Proceedings of the International Symposium on Computer Architecture*, 2009.

[2] M. Qureshi, V. Srinivasan, and J. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[3] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[4] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the International Symposium on Computer Architecture*, 2009.

[5] 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," in *Proceeding of IEEE International Symposium on Circuit and Systems*, 2007.

[6] S. Cho and H. Lee, "Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[7] H. Chung, B. Jeong, B. Min, Y. Choi, B. Cho, J. Shin, J. Kim, J. Sunwoo, J. Park, Q. Wang, *et al.*, "A 58nm 1.8 V 1Gb PRAM with 6.4 MB/s program BW," in *Digest of Technical Papers of the 2011 IEEE International Conference on Solid-State Circuits Conference (ISSCC)*, pp. 500–502, 2011.

[8] M. Qureshi, A. Seznec, L. Lastras, and M. Franceschini, "Practical and secure pcm systems by online detection of malicious write streams," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2011.

[9] M. K. Qureshi, J. Karidis, M. Fraceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[10] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for Hard Failures in Resistive Memories," in *Proceedings of the International Symposium on Computer Architecture*, 2010.

[11] E. Ipek, J. Condit, E. Nightingale, D. Burger, and T. Moscibroda, "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[12] D. 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 *Proceedings of the International Symposim on High Performance Computer Architecture*, pp. 466–477, 2011.

[13] D. Ielmini, D. Sharma, S. Lavizzari, and A. Lacaita, "Reliability impact of chalcogenide-structure relaxation in phase-change memory (pcm) cells part i: Experimental study," *Electron Devices, IEEE Transactions on*, vol. 56, no. 5, pp. 1070–1077, 2009.

[14] F. Bedeschi, R. Fackenthal, C. Resta, E. Donze, M. Jagasivamani, E. Buda, F. Pellizzer, D. Chow, A. Cabrini, G. Calvi, *et al.*, "A multi-level-cell bipolar-selected phase-change memory," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 428–625, IEEE, 2008.

[15] G. Close, U. Frey, J. Morrish, R. Jordan, S. Lewis, T. Maffitt, M. Breitwisch, C. Hagleitner, C. Lam, and E. Eleftheriou, "A 512mb phase-change memory (pcm) in 90nm cmos achieving 2b/cell,"

[16] H. Park, S. Yoo, and S. Lee, "Power Management of Hybrid DRAM/PRAM-based Main Memory," in *Proceedings of the 48th Design Automation Conference*, 2011.

[17] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proceedings of the Internatioanl Conference on Parallel Architectures and Compilation Techniques*, pp. 101–112, 2009.

[18] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security Refresh: Prevent Malicious Wear-out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping," in *Proceedings of the International Symposium on Computer Architecture*, 2010.

[19] N. Papandreou, H. Pozidis, T. Mittelholzer, G. Close, M. Breitwisch, C. Lam, and E. Eleftheriou, "Drift-tolerant multilevel phase-change memory," in *2011 3rd IEEE International Memory Workshop (IMW)*, pp. 1–4, IEEE.

[20] Y. Hwang, C. Um, J. Lee, C. Wei, H. Oh, G. Jeong, H. Jeong, C. Kim, and C. Chung, "Mlc pram with slc write-speed and robust read scheme," in *Proceedings of the 2010 Symposium on VLSI Technology (VLSIT)*, pp. 201–202, 2010.

[21] W. Xu and T. Zhang, "A time-aware fault tolerance scheme to improve reliability of multilevel phase-change memory in the presence of significant resistance drift," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1357–1367, 2011.

[22] B. Jun and P. Kocher, "The Intel Random Number Generator," tech. rep., Cryptography Research, Inc., April 1999.

[23] M. Klamkin and D. Newman, "Extensions of the birthday surprise," *Journal of Combinatorial Theory*, vol. 3, no. 3, pp. 279–282, 1967.

[24] A. Seznec, "A Phase Change Memory as a Secure Main Memory," *Computer Architecture Letters*, vol. 9, no. 1, pp. 5–8, 2010.

[25] L. Price and G. McKittrick, "Setting the Stage: The "New Economy" Endures Despite Reduced IT Investment," in *Digital Economy*, 2002.

[26] S. Madara, "The future of cooling high density equipment." 2007 IBM Power and Cooling Technology Symposium.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 190–200, 2005.

[28] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005. http://sesc.sourceforge.net.

[29] M. Stan and W. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, vol. 3, no. 1, pp. 49–58, 1995.

[30] T. Nirschl, J. Phipp, T. Happ, G. Burr, B. Rajendran, M. Lee, A. Schrott, M. Yang, M. Breitwisch, C. Chen, *et al.*, "Write strategies for 2 and 4-bit multi-level phase-change memory," in *IEEE International Electron Devices Meeting (IEDM)*, pp. 461–464, 2007.

[31] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *isca*, p. 0222, Published by the IEEE Computer Society, 1999.

[32] Y. Moon, Y.-H. Cho, H.-B. Lee, B.-H. Jeong, S.-H. Hyun, B.-C. Kim, I.-C. Jeong, S.-Y. Seo, J.-H. Shin, S.-W. Choi, H.-S. Song, J.-H. Choi, K.-H. Kyung, Y.-H. Jun, and K. Kim, "1.2V 1.6Gb/s 56nm 6F2 4Gb DDR3 SDRAM with Hybrid-I/O Sense Amplifier and Segmented Sub-Array Architecture," in *Proceedings of the 2009 IEEE International Solid-State Circuits Conference*, 2009.

[33] R. Baker, *CMOS: Circuit Design, Layout, and Simulation*. Wiley-IEEE Press, 2007.

[34] "International Technology Roadmap for Semiconductors, Emerging Research Devices," 2009.

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

[36] N. Wax, "On upper bounds for error detecting and error correcting codes of finite length," *IRE Transactions on Information Theory*, vol. 5, no. 4, pp. 168–174, 1959.

[37] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[38] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 4, 1970.

[39] M. Ghosh, E. Özer, S. Ford, S. Biles, and H.-H. S. Lee, "Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches," in *Proceedings of the 14th ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 165–170, 2009.

[40] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Scavenger: A new last level cache architecture with global block priority," in *Proceedings of the International Symposium on Microarchitecture*, 2007.

[41] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[42] B. Xiao and Y. Hua, "Using parallel bloom filters for multiattribute representation on network services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 1, pp. 20–32, 2009.

[43] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, "Hybrid cache architecture with disparate memory technologies," in *Proceedings of the International Symposium on Computer Architecture*, 2009.

[44] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2010.

[45] M. Qureshi, M. Franceschini, and L. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2010.

[46] M. Awasthi, M. Shevgoor, K. Sudan, B. Rajendran, R. Balasubramonian, and V. Srinivasan, "Efficient scrub mechanisms for error-prone emerging memories," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2012.

[47] B. Schroeder, E. Pinheiro, and W. Weber, "Dram errors in the wild: a large-scale field study," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pp. 193–204, ACM, 2009.

[48] D. Ielmini, A. Lacaita, and D. Mantegazza, "Recovery and drift dynamics of resistance and threshold voltages in phase-change memories," *IEEE Transactions on Electron Devices*, vol. 54, no. 2, pp. 308–315, 2007.

[49] D. Ielmini, S. Lavizzari, D. Sharma, and A. Lacaita, "Physical interpretation, modeling and impact on phase change memory (pcm) reliability of resistance drift due to chalcogenide structural relaxation," in *Proceedings of the IEEE International on Electron Devices Meeting (IEDM)*, pp. 939–942, 2007.

[50] W. Zhang and T. Li, "Helmet: A resistance drift resilient architecture for multi-level cell phase change memory system," in *Proceedings of 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pp. 197–208, 2011.

[51] N. H. Seong, D. H. Woo, V. Srinivasan, J. A. Rivers, and H.-H. S. Lee, "SAFER: Stuck-at-fault error recovery for memories," in *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, 2010.

[52] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security Refresh: Protecting Phase-Change Memory against Malicious Wear Out," *IEEE Micro*, vol. 31, no. 1, pp. 119–127, 2011.

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

[54] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[55] R. Bose and D. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and control*, vol. 3, no. 1, pp. 68–79, 1960.

[56] A. Hocquenghem, "Codes correcteurs d'erreurs," *Chiffres*, vol. 2, no. 2, pp. 147–156, 1959.

[57] D. Kang, J. Lee, J. Kong, D. Ha, J. Yu, C. Um, J. Park, F. Yeung, J. Kim, W. Park, *et al.*, "Two-bit cell operation in diode-switch phase change memory cells with 90nm technology," in *Proceedings of 2008 Symposium on VLSI Technology*, pp. 98–99, 2008.