ARCHITECTURES FOR LOW ENERGY, LOW LATENCY, HIGH PERFORMANCE, DURABLE MULTI-/TRIPLE-LEVEL CELL NON-VOLATILE MEMORIES

by

Poovaiah M. Palangappa

Bachelor of Engineering, R. V. College of Engineering, Visvesvaraya Technological University, 2008

Master of Technology, Indian Institute of Science, 2010

Submitted to the Graduate Faculty of the Swanson School of Engineering in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017
UNIVERSITY OF PITTSBURGH
SWANSON SCHOOL OF ENGINEERING

This dissertation was presented
by
Poovaiah M. Palangappa

It was defended on
February 24, 2017

and approved by

Kartik Mohanram, Ph.D., Associate Professor
Department of Electrical and Computer Engineering

William E. Stanchina, Ph.D., Professor
Department of Electrical and Computer Engineering

Jun Yang, Ph.D., Associate Professor
Department of Electrical and Computer Engineering

Zhi-Hong Mao, Ph.D., Associate Professor and William Kepler Whiteford Faculty Fellow
Department of Electrical and Computer Engineering

Ravi H. Motwani, Ph.D.
Intel Corporation, CA

Dissertation Director: Kartik Mohanram, Ph.D., Associate Professor
Department of Electrical and Computer Engineering
ARCHITECTURES FOR LOW ENERGY, LOW LATENCY, HIGH PERFORMANCE, DURABLE MULTI-/TRIPLE-LEVEL CELL NON-VOLATILE MEMORIES

Poovaiah M. Palangappa, PhD

University of Pittsburgh, 2017

Multi-level/triple-level cell non-volatile memories (MLC/TLC NVMs) such as phase-change memory and resistive RAM are the potential replacement candidates for DRAM, which is limited by its high refresh power and poor scaling potential. Besides the benefits of non-volatility (low refresh power) and improved scalability, MLC/TLC NVMs offer high data density and memory capacity over DRAM. However, the viability of MLC/TLC NVMs is limited due to (i) high programming energy/latency and low endurance, (ii) security vulnerability due to non-volatility, and (iii) high read latency. This dissertation presents three architectures for low energy, low latency, high performance, durable MLC/TLC NVMs.

First, it presents CompEx/CompEx++ coding, a low overhead scheme that synergistically integrates pattern-based compression with linear block expansion coding to realize simultaneous energy, latency, and lifetime improvements in MLC/TLC NVMs.

Second, it presents CASTLE, a Compression-based read-decrypt-free Architecture that provides a read-decrypt-free block-level Secure solution for low Latency, Low Energy durable NVMs. At its core, CASTLE adopts a block-level write-only sequence to eliminate the latency of the read-decrypt steps in state-of-the-art NVM security solutions. Whereas a write-only approach increases cell updates, and thereby energy and latency, CASTLE integrates pattern-based compression and expansion coding to realize energy reductions and lifetime improvements over state-of-the-art.
Third, it presents RAPID, a no-overhead critical-word-first read acceleration architecture for improved performance and durability in MLC/TLC NVMs. At its core, RAPID encodes the critical words in a cache line using only the most significant bits (MSbs) of the MLC/TLCs. Since the MSbs of an NVM cell can be decoded using a single read strobe, the data (i.e., critical words) encoded using the MSbs can be decoded with low latency.

This dissertation thus addresses the core challenges of write/read energy and latency, endurance, and security of MLC/TLC NVMs and proposes multiple solutions to these challenges.
# TABLE OF CONTENTS

**PREFACE** ................................................. xii

**1.0 INTRODUCTION** .................................... 1

1.1 Organization of this dissertation .................. 10

**2.0 BACKGROUND AND MOTIVATION** .................. 11

2.1 Write Energy and Latency of MLC/TLC NVM ........ 11

2.2 NVM Security ........................................ 13

2.3 Read Latency of MLC/TLC NVMs ..................... 16

2.3.1 MLC/TLC NVM Read ............................... 16


**3.0 COMPEX AND COMPEX++ CODING** ................. 20

3.1 Compression .......................................... 21

3.1.1 Frequent pattern compression (FPC) .......... 21

3.1.2 Base-Delta-Immediate (BΔI) .................... 22

3.2 Expansion coding .................................... 25

3.3 CompEx coding ....................................... 28

3.3.1 Theoretical analysis of CompEx coding ......... 29

3.3.2 CompEx codec architecture ..................... 31

3.3.3 CompEx codec logic overhead ................... 32

3.4 CompEx++ coding .................................... 34

3.4.1 Theoretical analysis of CompEx++ coding ..... 36

3.4.2 CompEx++ codec logic overhead ................. 38

3.5 CompEx/CompEx++ coding, ECC, and encryption .... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>CompEx++ECC (CompEx/CompEx++ coding with ECC)</td>
<td>39</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Improvement in NVM scrub interval due to CompEx++ECC</td>
<td>40</td>
</tr>
<tr>
<td>3.5.3</td>
<td>CompEx++ coding and encryption</td>
<td>43</td>
</tr>
<tr>
<td>3.6</td>
<td>Methodology</td>
<td>45</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Full-system simulation</td>
<td>45</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Trace-driven simulation</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>Evaluation and results</td>
<td>47</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Memory energy/latency</td>
<td>48</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Full-system evaluation</td>
<td>52</td>
</tr>
<tr>
<td>3.8</td>
<td>Related Work</td>
<td>55</td>
</tr>
<tr>
<td>3.9</td>
<td>Summary</td>
<td>57</td>
</tr>
<tr>
<td>4.0</td>
<td>CASTLE</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>‘Read-decrypt’-free write</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>CASTLE: Counter design</td>
<td>60</td>
</tr>
<tr>
<td>4.3</td>
<td>CASTLE: Low latency/energy encoding</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>CASTLE: Architecture</td>
<td>69</td>
</tr>
<tr>
<td>4.5</td>
<td>Methodology</td>
<td>71</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Full-system simulation</td>
<td>71</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Trace-driven simulation</td>
<td>73</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation and results</td>
<td>75</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Memory energy/latency</td>
<td>76</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Full-system evaluation</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Related Work and CASTLE</td>
<td>82</td>
</tr>
<tr>
<td>4.8</td>
<td>Summary</td>
<td>83</td>
</tr>
<tr>
<td>5.0</td>
<td>RAPID</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>RAPID: Write-Time CW Encoding</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>RAPID: Read Timing and Acceleration</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>RAPID: Data Organization</td>
<td>91</td>
</tr>
<tr>
<td>5.4</td>
<td>RAPID: NVM reliability</td>
<td>93</td>
</tr>
<tr>
<td>5.5</td>
<td>RAPID encryption</td>
<td>95</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64-bit FPC patterns</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>64-byte BΔI patterns</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>CompEx codec overhead</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>Chosen expansion codes for CompEx++ coding</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Scrub intervals and overheads for different ECC schemes</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Configuration of the evaluation architecture.</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>Composite workloads for evaluating CompEx/CompEx++ coding</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>Trace-based simulation results for CompEx/CompEx++ coding</td>
<td>51</td>
</tr>
<tr>
<td>9</td>
<td>Full-system simulation results for CompEx/CompEx++ coding</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>Global counter sizes for various system uptimes and availabilities</td>
<td>63</td>
</tr>
<tr>
<td>11</td>
<td>Configuration of the evaluation architecture.</td>
<td>72</td>
</tr>
<tr>
<td>12</td>
<td>Composite workloads for evaluating CASTLE coding</td>
<td>74</td>
</tr>
<tr>
<td>13</td>
<td>Trace-based simulation results for CASTLE</td>
<td>77</td>
</tr>
<tr>
<td>14</td>
<td>Full-system simulation results for CASTLE</td>
<td>81</td>
</tr>
<tr>
<td>15</td>
<td>Estimated gains using RAPID</td>
<td>90</td>
</tr>
<tr>
<td>16</td>
<td>Configuration of the evaluation architecture.</td>
<td>97</td>
</tr>
<tr>
<td>17</td>
<td>Composite workloads for evaluating RAPID coding</td>
<td>98</td>
</tr>
<tr>
<td>18</td>
<td>Trace-based simulation results for RAPID</td>
<td>103</td>
</tr>
<tr>
<td>19</td>
<td>Full-system simulation results for RAPID</td>
<td>104</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1 System level impact of MLC/TLC NVMs in comparison to SLC NVMs  . . .  3
2 Motivation for read acceleration . . . . . . . . . . . . . . . . . . . . . . . 5
3 Energy and latency issues using P&V . . . . . . . . . . . . . . . . . . . . . 12
4 AES encryption and decryption . . . . . . . . . . . . . . . . . . . . . . . . 15
5 Read-decrypt-modify-write sequence in state-of-the-art CME schemes . . . 16
6 Three-strobe iterative sensing for a TLC read . . . . . . . . . . . . . . . 17
7 \((k,m)_q\) expansion coding . . . . . . . . . . . . . . . . . . . . . . . . . 26
8 Expansion coding example . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
9 Illustration of CompEx coding . . . . . . . . . . . . . . . . . . . . . . . . . 29
10 CompEx codec . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
11 CompEx codec architecture . . . . . . . . . . . . . . . . . . . . . . . . . 31
12 Illustration of CompEx++ coding . . . . . . . . . . . . . . . . . . . . . . 35
13 Illustration of compression prefix decoding for CompEx++ coding. . . . 37
14 ECC in CompEx/CompEx++ coding . . . . . . . . . . . . . . . . . . . . . 40
15 NVM scrub interval . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
16 Energy results for CompEx/CompEx++ coding . . . . . . . . . . . . . . 50
17 Latency results for CompEx/CompEx++ coding . . . . . . . . . . . . . . 51
18 IPC results for CompEx/CompEx++ coding . . . . . . . . . . . . . . . . 52
19 bandwidth results for CompEx/CompEx++ coding . . . . . . . . . . . . 55
20 Write timing diagrams for CASTLE and DEUCE . . . . . . . . . . . . . . 60
21 Read timing diagrams for CASTLE and DEUCE . . . . . . . . . . . . . . 61
22 Illustrative example for CASTLE . . . . . . . . . . . . . . . . . . . . . . . . 68
I would like to take this opportunity to thank everyone who has helped me through this journey of my Ph.D. First and foremost, I would like to thank my advisor, Professor Kartik Mohanram, for supporting me during these past four years. I’m grateful for the extensive freedom he has given to me in pursuing my research. Kartik’s inspirational advice, insightful discussions, commitment to my all-round success, and mentoring me with my best interests in mind, are the primary reasons for my success in Ph.D. He has been an ideal Guru and a constant source of motivation for my research. I’m positive that his words of wisdom would guide me through the rest of my life.

I would also like to thank the members of my Ph.D. committee, Professors Jun Yang, Zhi-Hong Mao, and William Stanchina, and Dr. Ravi Motwani, for their encouragement, advice, and suggestions in general. I would like to thank Professor Jun Yang for the immensely helpful discussions, which helped me build my knowledge on computer architecture, during my early days in Ph.D. I would like to thank Professor Zhi-Hong Mao for his courses on control theory, which reignited my love towards mathematics. Furthermore, the combination of Prof. Mao’s vast knowledge and great humility has been both an enlightening and humbling experience for me. I would like to thank Ravi for his support and faith in me, and for the extensive discussions on coding theory during my time at Intel. I would also like to thank late Professor Steve Levitan for his kind advice and enthusiastic lectures.

I would like to thank my family for their support, motivation, and blessings through every step of my life. I acknowledge, but can never thank enough for, their sacrifices for my success and well-being.

Finally, I would like to thank my friends in Pittsburgh, San Jose, and back home in Bangalore, for sticking with me during the thick and thin of my life.
1.0 INTRODUCTION

The ITRS projects that resistance-class non-volatile memory (NVM) technologies such as phase-change memory (PCM) [3] and resistive RAM (RRAM) [4] are suitable replacement candidates for DRAM due to their low refresh power and better scaling potential [5]. Resistance-class NVMs store data by modulating the resistance of the storage material. Due to the wide range in the resistance of the NVM cells, multi-level/triple-level cell NVMs (MLC/TLC NVMs) that can be programmed to more than two distinct resistance ranges are the subject of active research and development [6]. MLC/TLC NVMs offer higher data density in comparison to single-level cell NVMs (SLC NVMs) since they can store more than 1 logical data-bit. However, MLC/TLC NVMs aggravate the challenges of working with SLC NVMs, primarily due to (i) the complex, iterative program-and-verify procedure necessary to program MLC/TLC cells [7], which increases access latencies as well as power/energy requirements [8] and reduces cell lifetime due to high currents and multiple iterations on writes [9], (ii) increased data security vulnerabilities due to non-volatility, which requires the stored data to be secured using data encryption, which incurs high bit-write, instructions-per-cycle (IPC), and bandwidth overheads, and (iii) increased read latency due to the iterative sensing procedure, which is employed to decode the bits stored in an NVM cell. These limitations are further aggravated by the move from single-level cell (SLC) to denser multi-/triple-level cell (MLC/TLC) NVM technologies.

Problems, related work, and motivation:

1. Whereas MLC/TLC NVMs offer higher capacity by packing multiple logical bits/cell, the energy and latency to program each cell is high in comparison to SLC NVMs. Fig. 1(a) shows the energy and latency v/s capacity trends for SLC, MLC, and TLC RRAM [10].
This increase in energy and latency for programming an MLC/TLC NVM cell translates to system level performance penalties in terms of IPC, bandwidth, and energy as shown in Fig. 1(b), which is measured for RRAM by using Stream benchmark [11] on MARSS [12] and DRAMSim2 [13]. These challenges of MLC/TLC NVMs pose a serious technological roadblock for their adoption in high performance computing systems.

MLC/TLC NVMs definitely benefit from the broad set of solutions developed to improve energy, latency, and lifetime of SLC NVMs in the literature. Example solutions include coding [14–16], write scheduling [17–20], data-migration using address translation [21–23], and architectural improvements [24,25]. However, these solutions do not address the energy/latency problems of MLC/TLC NVMs that is primarily due to the iterative nature of programming a cell. Solutions that explicitly address the challenges of working with MLC/TLC NVMs have mostly focused on data compression and data coding. Whereas data compression techniques such as [16,26–33] approach this problem by reducing the number of cell-writes per write access, data encoding solutions such as [9,10,34–37] reduce energy and latency by using only the desirable states and excluding undesirable states of an MLC/TLC NVM cell. However, these solutions are limited by computation, memory, and logic overheads. This motivates the development of solutions that can simultaneously realize the advantages of both data compression and data coding for no memory overhead.

2. NVM non-volatility makes MLC/TLC NVMs vulnerable to security attacks [38–40]. Although data encryption provides a robust solution to counter security vulnerabilities in NVMs, it incurs significant bit-write, instructions-per-cycle (IPC), and bandwidth overheads. A broad set of solutions [38–40] have been proposed to scramble data for resilience. However, data scrambling — achieved through encryption [38–40] — increases cell-update activity to 50%, 75%, and 87.5% of the total cells for SLC, MLC, and TLC NVMs, respectively. This increased cell activity results in an increase in memory energy and latency, and a decrease in IPC, bandwidth, and lifetime. Whereas standard bit-write reduction techniques like data-comparison write (DCW) [15] and flip-n-write (FNW) [41] can reduce bit-writes, their effectiveness is constrained by the increased entropy of the encrypted data. Block-level encryption (BLE) [42] and DEUCE [43] are state-of-the-
Figure 1: (a) Energy and latency trends for SLC, MLC, and TLC RRAM — packing multiple bits/cell increases the energy and latency for programming the cell. (b) Performance penalties for MLC/TLC RRAM in comparison to SLC RRAM for Stream [11] using MARSS [12] and DRAMSim2 [13]. The key take-away is that the move to denser technologies like MLC/TLC usually degrades IPC, bandwidth, and energy.

Art solutions for NVM security that achieve cell activity reduction by reducing memory encryption granularity from 512 bits to 128 bits and 16 bits, respectively. Both BLE and DEUCE use a read-decrypt-modify-write sequence, where the ciphertext (encrypted data) stored in the main memory is read to the processor-side memory controller, decrypted, and compared for modifications with the incoming plaintext data in chunks of 128 bits and 16 bits, respectively. Both BLE and DEUCE ensure that only the ciphertext corresponding to modified data chunks are updated in the memory. Whereas BLE and DEUCE have been successful in bit-write reduction over DCW and FNW, they are limited by the mandatory read-decrypt steps that are integral to every write; our full-system simulations show that the read-decrypt-modify-write sequence can result in an IPC loss of up to 50%, which is prohibitive in practice. This motivates the development of read-decrypt-free solutions for NVM security.

3. In contrast to SLC NVMs, MLC/TLC NVMs use iterative sensing for reads and iterative program-and-verify for writes [2,6,44]. These iterative high latency read and write operations limit system performance in practice. Whereas the limitations of writes in MLC/TLC NVMs have been extensively addressed [15–25,41,45], less effort has been
devoted to improving read latency. Decoupled bit mapping (DBM) [1] and striped PCM (SPCM) [2] are state-of-the-art solutions that explicitly attempt to improve MLC/TLC NVM read latency using the observation that bits encoded in an NVM cell require different latencies for decoding, i.e., the most-significant bit (MSb) of an NVM cell can be decoded using just one read strobe in comparison to the least-significant bit (LSb), which requires 2 (3) read strobes for an MLC (TLC). DBM maps separate logical address blocks using MSbs or LSbs alone (for MLC PCM) and uses OS support to pair read-intensive pages to MSb-mapped address blocks for improving read performance. SPCM improves upon DBM by encoding odd/even address cache lines exclusively using MSbs/LSbs. However, both SPCM and DBM improve read performance at the expense of write latency, energy, and endurance, (by as much as 2× (3×) for MLC (TLC) NVM).

Figure 2 illustrates the impact of read/write latencies on module-level (energy and latency) as well as system-level (IPC and bandwidth) parameters, measured for TLC RRAM [10] using Stream benchmarks [11], MARSS [12], and DRAMSim2 [13]. Three scenarios: (i) baseline, with normal read and write latencies, (ii) state-of-the-art (DBM/SPCM), with fast reads (2×) but slow writes (0.5×), and (iii) motivation, with fast reads (2×) and normal writes are compared. It is clear from Fig. 2 that although improving read latency can improve system performance, doing so at the expense of write latency can be counter-productive on system performance. This motivates the development of solutions that improve NVM read latency without impacting write latency (this work).

Contributions:

1. First, this dissertation proposes compression-expansion (CompEx) coding, a low overhead scheme that synergistically integrates statistical compression with expansion coding to realize simultaneous energy and latency improvements in MLC/TLC NVMs. The core idea of CompEx coding is to selectively apply expansion codes, i.e., linear block codes that encode data using only the low energy states of an MLC/TLC (q-ary) cell to compressed data, thereby ensuring that the resulting data in expansion-coded form will not exceed the original data width. Thus, CompEx coding can translate to energy, latency, and lifetime improvements in MLC/TLC NVMs for negligible-to-no memory
overhead (although CompEx coding requires a tag bit, this is absorbed at no cost for data widths that are powers of 2 for TLC NVMs), which is a significant advancement over solutions in the literature. Although CompEx coding is agnostic to the choice of compression technique, in this dissertation, CompEx coding is evaluated using two compression techniques — frequent pattern compression (FPC) \cite{46} and base-delta-immediate (B∆I) compression \cite{47}. FPC is a low overhead, lossless word-level compression technique that uses a static pattern table to match a wide range of values without the need for off-line/online application profiling. B∆I is a compression technique that is similar to FPC, but operates at cache line level instead of word-level. B∆I takes advantage of regularity of data storage and the limited dynamic range of stored data. Following compression, the compressed data is selectively encoded using \((3,2)_8\) expansion coding. Expansions codes are linear block codes such that data stored in a \(q\)-ary NVM cell is encoded using only the \(p\) lowest energy/latency cell states \((p < q)\). Expansion codes can be studied theoretically in the context of MLC/TLC NVMs and theoretical estimates for the maximum achievable energy reductions can be derived. We show that for \((3,2)_8\) and \((6,5)_8\)
expansion coding, the expected energy reduction is 2.3× and 1.4×, respectively, over classical binary encoding; in close agreement with these estimates, our results on the SPEC CPU2006 \cite{48} benchmark suite show that it is possible to realize energy reductions of 2.2× and 1.25× in practice.

Furthermore, we present CompEx++ coding, a flexible CompEx coding scheme, which leverages the variable compressibility of pattern-based compression techniques to integrate custom expansion coding to each of the compression patterns to exploit the maximum energy/latency benefits of CompEx coding. Along the lines of CompEx coding, we evaluate CompEx++ coding for FPC and B∆I by integrating (3,2)\(_s\) and (3,1)\(_s\) expansion coding schemes. In addition, we also provide a mathematical model for the analysis of energy gains for compression, expansion coding, and CompEx++ coding, independently. Finally, we discuss the integration of error detection and correction (EDAC) with CompEx++ coding using both theory and simulations. CompEx++ coding extends the NVM EDAC capability by integrating with stronger EDAC schemes; we show that CompEx++ coding reduces the NVM scrub overhead by 40% over standard EDAC schemes for no additional overhead.

2. In order to address the high cell update activity of secure MLC/TLC NVMs, this dissertation presents CASTLE, a compression-based architecture that provides a read-decrypt-free, i.e., write-only block-level solution for NVM security to realize low latency, low energy durable NVMs. At its core, CASTLE adopts a block-level write-only sequence to completely eliminate the read latency of the read-decrypt steps in state-of-the-art methods such as BLE and DEUCE. Whereas a write-only approach potentially increases cell updates drastically, increasing energy and latency, CASTLE integrates data compression and partial data mapping (PDM) \cite{10} — an instance of expansion coding — to realize energy reductions and lifetime improvements over state-of-the-art methods for NVM security. Although CASTLE is agnostic to the choice of compression technique, CASTLE is evaluated using two compression techniques—frequent pattern compression (FPC) \cite{46} and base-delta-immediate (B∆I) \cite{47}. On all writes, data goes through a compression step prior to counter mode encryption (CME) and transmission to the NVM module. A PDM codec in the NVM module controller selectively applies PDM to compressed
encrypted data; however, the un compressed encrypted data is encoded using classical binary encoding. CASTLE integrates a novel counter design framework to enable the read-decrypt-free write-only sequence in CME solutions. Note that CASTLE is consistent with state-of-the-art CME schemes and is thus a drop-in replacement for such schemes in practice. CASTLE is also compatible with soft/hard error detection and correction support (ECC, ECP, etc.). Thus, CASTLE is a simple practical NVM memory security solution for low latency, low energy, and improved endurance.

3. In order to address the high read latency challenges of MLC/TLC NVM, this dissertation makes the following contributions. First, we propose RAPID, a no-overhead critical-word-first read acceleration architecture for improved performance and durability in MLC/TLC NVMs. At its core, RAPID improves read latency by encoding the critical word (CW) in a cache line using only MSbs; as opposed to read-time CW optimization, RAPID relies on static write-time CW optimization, which is a relatively new area of research [49]. Second, RAPID is a zero-cost solution; we show that unlike existing solutions such as DBM and SPCM, which incur up to 3× write overhead, RAPID does not incur any write overhead. Third, RAPID complements soft/hard error detection and correction techniques like ECC [50]/ECP [51] and state-of-the-art counter mode memory encryption schemes. RAPID is thus a simple holistic solution to improve the read latency of MLC/TLC NVMs with no impact to write latency, write energy, and endurance.

**Evaluation:** In this dissertation, the efficiency of CompEx/CompEx++ coding, CASTLE, and RAPID is evaluated on a 64-bit word, 64-byte cache line architecture, i.e., every word is stored using $22 \left(\lceil \frac{64}{3} \rceil \right)$ TLCs and every cache line is stored using $171 \left(\lceil \frac{512}{3} \rceil \right)$ TLCs. CompEx/CompEx++ coding relies on a codec embedded in the NVM module controller, which abstracts all data manipulations from the memory controller on the processor side, allowing the processor to communicate seamlessly with the NVM module. On memory writes, the CompEx/CompEx++ codec attempts to compress the data at the word-level using FPC or the cache line level using BΔI. If compression is successful, the data is encoded using $(3,2)_8/(3,1)_8$ expansion coding before it is written into the NVM array; else the data is written as-is into the NVM array. Note that although a single tag bit is necessary to
record the outcome of CompEx/CompEx++ coding, it is concatenated with the data (513 logical bits for BΔI and 65 logical bits for FPC) and absorbed into the last TLC at no cost (⌈513/3⌉ = ⌈512/3⌉ = 171 TLCs for BΔI and ⌈65/3⌉ = ⌈64/3⌉ = 22 TLCs for FPC). On memory reads, the tag bit is recovered from the last TLC and used to determine if the word must be decoded using the CompEx/CompEx++ codec, or if it can be forwarded directly to the NVM module controller. The logic overhead of the CompEx/CompEx++ codec\(^1\) is ≈ 10k/12.5k gates (< 0.1% per NVM module); although the codec has a latency of 2-3 cycles, we show that this can be hidden through memory access pipelining.

CASTLE relies on 2 codecs—a compression/decompression codec in the processor-side memory controller and a PDM codec in the NVM DIMM controller. The logic overhead of the 2 codecs is ≈ 10k gates (< 0.1% per NVM module); although the codecs have a latency of 2-3 cycles, we show that this can be hidden through memory access pipelining in practice.

RAPID and SPCM require equivalent modifications to I/O gating circuits and incur similar routing overheads. Furthermore, RAPID memory controller does not require any dedicated codec for CW optimization due to the static nature of the CW prediction.

**Results:** First, our full-system simulations of a system that integrates TLC RRAM using MARSS [12] and DRAMSim2 [13] for workloads from the SPEC CPU2006 [48] benchmark suite show that CompEx coding improves system performance by 5.7%, as measured using IPC, and memory-system performance by 11.8%, as measured using memory bandwidth, in comparison to binary encoding using data-comparison write (DCW) [15] (a read-modify-write process that only updates changed cells). Additionally, CompEx++ coding extends the benefits of CompEx coding to improve IPC and bandwidth by 10.6% and 19.9% over DCW and 5.2% and 6.4% over CompEx coding.

For a deep, multi-billion-instruction evaluation of CompEx/CompEx++ coding, we use NVMain [52] with the memory traces of SPEC CPU2006 benchmarks generated using the Intel Pin toolset [53]. Simulations of these traces show that CompEx/CompEx++ coding reduces total (active) write energy by 57%/61% (76%/78%), 16%/16% (15%/15%), and 25%/32% (23%/29%) on average over DCW, FPC, and BΔI, respectively. Simultane-\(^1\)In this dissertation, only the logic overhead of the FPC-based CompEx codec is evaluated. The logic overhead of the BΔI-based CompEx codec is based on the original proposal of classical BΔI [47].
ously, CompEx/CompEx++ coding reduces write latency by 23.5%/26%, 18.5%/19%, and 23.5%/28% on average in comparison to DCW, FPC, and BΔI, respectively, and improves TLC RRAM lifetime by 1.8x.

Second, our full-system simulations of a system that integrates TLC RRAM using MARSS [12] and DRAMSim2 [13] for workloads from the SPEC CPU2006 [48] benchmark suite show that CASTLE improves IPC by 1.6x and 1.2x and memory bandwidth by 2.1x and 1.3x in comparison to classical binary encoding and state-of-the-art DEUCE [43], respectively. Note that all the evaluated cases integrate data-comparison write (DCW) [15] to update only the modified cells in the NVM array.

For a deep, multi-billion-instruction evaluation of CASTLE, we use NVMain [52] with the memory traces of SPEC CPU2006 benchmarks generated using the Intel Pin toolset [53]. Simulations of these traces show that FPC-CASTLE/BΔI-CASTLE that integrates PDM(8,4) reduces total write energy by 33.5%/25.4 and 23.3%/14% over classical binary coding and state-of-the-art DEUCE, respectively. Simultaneously, FPC-CASTLE (BΔI-CASTLE) reduces write latency by 43% (48%) and 36% (42%) on average in comparison to classical binary coding and DEUCE, respectively. Furthermore, since some applications cannot tolerate the re-encryption overhead inherent to global counter architectures, we evaluate counter-cache-BΔI-CASTLE, which improves IPC by 52% and 14% in comparison to classical binary coding and DEUCE, respectively. Finally, we show that CASTLE improves TLC RRAM lifetime by 1.8x in comparison to classical binary coding.

Finally, our full-system simulations of RAPID using MARSS [12] and DRAMSim2 [13] for workloads from the SPEC CPU2006 [48] benchmark suite show that RAPID improves IPC by 7.5% and 15% and memory bandwidth by 7% and 23% in comparison to state-of-the-art SPCM [2] and classical binary encoding with data-comparison write (DCW, i.e., read-modify-write), respectively.

For a deep, multi-billion-instruction evaluation of RAPID, we use NVMain [52] with memory traces of SPEC CPU2006 benchmarks generated using the Intel Pin toolset [53]. Simulations of these traces show that RAPID reduces total read latency by 21%, and 11% on average over SPCM and classical binary encoding, respectively. RAPID also reduces memory energy by 24%, and 41% over SPCM and classical binary encoding, respectively.
Finally, our evaluation of RAPID integrated with ECP (RAPID-ECP) shows a TLC NVM lifetime improvement of $4.12 \times$ over state-of-the-art SPCM; RAPID-ECP also has lifetime equivalent to classical binary encoding.

1.1 ORGANIZATION OF THIS DISSERTATION

This dissertation is organized as follows. Chapter 2 provides the background for the write energy/latency, security, and read latency problems in MLC/TLC NVMs. Chapters 3, 4, 5 describes CompEx++ coding, CASTLE, and RAPID, respectively, with examples and presents the evaluation methodology, simulation setup, results, and the related work. Chapter 6 concludes the dissertation and Chapter 7 presents the future work.
2.0 BACKGROUND AND MOTIVATION

2.1 WRITE ENERGY AND LATENCY OF MLC/TLC NVM

MLC/TLC NVM technologies such as PCM and RRAM are the subject of active research and development as replacement candidates for DRAM, which is limited by its high refresh power and poor scaling potential [54–57]. However, most MLC/TLC NVMs suffer from higher write energy [8, 57] and latency [58] per cell, limited lifetime [3] and asymmetric read/write latencies [8, 25, 59–61]. In this section, we discuss the procedure used for programming an MLC/TLC NVM cell and its associated limitations.

**MLC/TLC NVM program-and-verify (P&V):** MLC/TLC NVM exhibits non-deterministic behavior due to process variation, variation in material composition, and resistance drift. This non-determinism increases the complexity of programming an MLC/TLC NVM to the target resistance range using a single precise pulse of current or voltage. Therefore, in practice, MLC/TLC NVMs are programmed using an iterative P&V procedure to bring the resistance of the cell to the required range. P&V uses a combination of *set-to-reset* (STR) and *reset-to-set* (RTS). In STR (RTS), the cell is first brought to a full SET (RESET) state by applying a long (short) pulse of small (high) magnitude current. Following this, multiple small duration RESET (SET) pulses are applied until the resistance of the cell is brought to the required range. In contrast, RTS starts by first bringing the cell to the full RESET state by applying a short pulse of high magnitude current, which is followed by multiple longer duration SET pulses until the resistance of the cell is in the required range. However, since a TLC NVM has 8 different resistance states, using only STR or RTS leads to high write latency. Hence, TLC NVMs are programmed using a combination of both STR and RTS.
Figure 3: Energy and latency for TLC RRAM [10] using P&V across 8 states. States 0-3 and 4-7 are programmed using RTS and STR, respectively. Programming the central states (2, 3, 4, and 5) requires more energy and latency in comparison to the terminal states (0, 1, 6, and 7). The key take-away is that the overall energy and latency can be reduced by encoding data using only the low energy states.

Impact of P&V on energy and latency: Iterative P&V has a negative impact on the energy and latency of MLC/TLC NVMs. First, writing to MLC/TLC NVMs require much higher current and latency than their SLC counterparts due to the iterative P&V method. Second, in order to prevent the modules from overloading the power supply, and to prevent overheating, NVM modules are restricted to writing a limited amount of data at once, which increases the write latency and impacts performance [25]. Third, since a single write access...
can potentially require MLC/TLCs to be brought to different target states in a word, the latency of the write operation is determined by the longest latency cell write, creating a bottleneck for individual write operations [8]. Finally, a single write for MLC/TLC NVMs involves multiple P&V cycles, which limits the lifetime of these memory cells to around $10^{5-8}$ writes [8] in comparison to SLC NVM, which has a lifetime of $10^{8-10}$ writes [3, 56, 57].

2.2 NVM SECURITY

Main memory data security is a pressing concern in modern high performance computing systems, which is further exacerbated by NVM non-volatility (data persistence). The secure-processor paradigm for data security assumes that the processor, on-chip cache system, and on-chip memory controller are secure, while the memory bus and the main memory that are off-chip are vulnerable to attacks [43]. Data encryption provides a robust solution for securing the memory bus and memory data against attacks. However, integrating data encryption with NVMs has to address the high energy, latency, and limited endurance (due to increased cell activity) of NVM technologies. These limitations are further aggravated in the case of MLC/TLC NVMs. We begin this section by discussing the attack models considered in this dissertation, followed by a discussion of counter mode encryption (CME) schemes. Second, we discuss the read-decrypt-modify-write sequence integral to the state-of-the-art CME schemes in detail, and motivate a write-only paradigm for NVM security.

Attack models: This dissertation focuses on two important security attacks: the stolen-DIMM attack and the bus-snooping attack. In the stolen-DIMM attack, the adversary gains physical access to the NVM DIMM at some point in time during the operation of the system [43, 62]. The stolen-DIMM attack is particularly severe in NVM systems where the stored data persists in the memory for a long time. The bus-snooping attack is a conventional attack, where an adversary can obtain a memory trace—address, data, access times, etc.—by simply probing the bus between the main memory and the processor-side memory controller [39, 63].
Counter mode encryption (CME): Data encryption provides a robust solution for NVM data security. However, direct data encryption (512 bits) makes the encryption latency prohibitively high. Therefore, main memory encryption employs one-time pad (OTP) schemes that do not encrypt the data directly. OTP schemes use standard block ciphers like advanced encryption standard (AES) and a global key\textsuperscript{1} to first generate an OTP (a random vector), which is then XORed with the plaintext data to obtain ciphertext, i.e., encrypted data. This effectively decouples OTP generation from data access, allowing pipelining of memory encryption for latency reduction. However, encrypting data using an OTP that is generated using only the global key cannot successfully secure the data — by comparing the statistics of the plaintext and ciphertext data, one can employ dictionary-based attacks for breaking the security of such a scheme.

AES schemes thwart dictionary-based attacks using the cache line address along with the global key for OTP generation. However, such a scheme is still vulnerable to bus-snooping attacks, where consecutive accesses to the same location can be monitored to decrypt data using simple XOR operation \cite{43}. State-of-the-art counter mode encryption solutions thwart both types of attacks by using a cache-line level counter along with the global key and address for generating the OTP, as shown in Fig. 4. During a write in a CME scheme, the previous counter value and the previous ciphertext are first read from the memory location. The counter value is then incremented, ensuring that the data to a given location never gets encrypted using the same OTP (the OTP is XORed with the plaintext to obtain ciphertext) twice. Note that CME schemes require the storage of the counter value used for encryption in plaintext along with the ciphertext (incurring storage overhead), so that the decryption engine has access to global key, address, and the counter value during a read. Furthermore, if any of the line counters overflow, the whole memory needs to be re-encrypted using a new global key. Thus, the counters are designed to be large enough to keep the re-encryption overhead within acceptable limits.

\textsuperscript{1}Traditionally, the global key is stored in the memory controller, which is considered secure under the secure-processor paradigm.
Read-decrypt-modify-write: Although CME schemes are robust solutions for securing data, they incur high energy and latency overheads. First, encryption scrambles the data resulting in an increase in the cell-update activity to 50%, 75%, and 87.5% of the total number of bits on average for SLC, MLC, and TLC NVMs, respectively. This increase in cell-update activity directly translates to increased energy/latency. Second, the energy/latency overhead is further exacerbated with MLC/TLC NVMs due to their non-linear energy/latency profile (due to the iterative program-and-verify programming procedure), as shown in Fig. 3.

To reduce these energy and latency overheads, secure-processor-paradigm CME schemes such as BLE and DEUCE use a block-level read-decrypt-modify-write sequence. In this sequence, the entire cache line is first read to the memory controller (secure area), decrypted, and compared to the new incoming plaintext, so that only those parts of the ciphertext that correspond to modified data blocks are written to the memory. Additionally, existing CME schemes also need to read the counter value (for OTP generation) stored in the memory before every write. The mandatory read-decrypt step on every write operation, as shown in Fig. 5, is a performance bottleneck in practice. Our simulations using workloads from the SPEC CPU2006 [48] benchmark suite show that the mandatory read-decrypt steps result in
Figure 5: Illustration of the read-decrypt-modify-write sequence in state-of-the-art CME schemes in comparison to the write-only sequence.

up to 50% IPC overhead in write intensive applications (WD2 in Fig. 35). This motivates the design of NVM encryption architectures that eliminate the read-decrypt steps in favor of a simple write-only approach.

2.3 READ LATENCY OF MLC/TLC NVMS

This section describes the iterative sensing for MLC/TLC reads and introduces DBM [1] and SPCM [2], which are the state-of-the-art read latency improvement techniques for MLC/TLC NVMs.

2.3.1 MLC/TLC NVM Read

A $2^n$-level NVM cell that stores $n$ logical bits of data can be programmed to one of $2^n$ states (resistance ranges). During a read, the read circuit uses multiple strobes to identify the state of the NVM cell using binary search [2,6,44]. Therefore, a $2^n$-level cell that stores $n$ logical bits requires at least $n$ strobes to accurately determine the state of the NVM cell. Thus, $n = 2$ for MLC NVMs and $n = 3$ for TLC NVMs.

Example: Without loss of generality and for ease of understanding, we use a TLC NVM cell programmed to state ‘2’ to illustrate the MLC/TLC NVM read mechanism. Due to process variations and resistance-drift, the resistance of an NVM cell exhibits a resistance
Figure 6: Three-strobe iterative sensing for a TLC read. The key take-away is that whereas the MSb is decoded immediately after the first strobe, the LSb is decoded only after the last (third) strobe. Therefore, the MSb can be decoded faster in comparison to the mSb and the LSb in a TLC.

distribution as illustrated in Fig. 6. The read circuit attempts to find the range of resistance of the given NVM cell, illustrated in blue in Fig. 6. First, the read strobe is placed in the center of the resistance range of the cell, i.e., the resistance value that separates cell states 3 and 4; this strobe determines whether the resistance of the given cell ($R_{\text{cell}}$) is greater or lesser than the strobe resistance ($R_{\text{strobe-1}}$). As illustrated in Fig. 6(a), $R_{\text{cell}} > R_{\text{strobe-1}}$, which indicates that the data can be one of the states 0, 1, 2, or 3. This first logical bit of information decoded from the TLC is termed the most significant bit (MSb), and is ‘0’ in this case. Second, now that $R_{\text{cell}}$ is determined to be in the range of cell states 0-3, a read strobe is placed between the resistance ranges of cell states 1 and 2 to determine the next significant bit, termed the middle significant bit (mSb), as illustrated in Fig. 6(b)\(^2\). Since $R_{\text{cell}} < R_{\text{strobe-2}}$, the state of the given cell is determined to be either 2 or 3. Finally, a read strobe is placed between cell states 2 and 3, as illustrated in Fig. 6(c). Since $R_{\text{cell}}$

\(^2\)Note that for an MLC NVM cell, there are only two logical bits, the MSb and the LSb
> R_{\text{strobe-3}}, the given cell is determined to be programmed to state 2, i.e., the LSb is ‘0’. The read terminates once the LSb is determined. In summary, three strobes are necessary to determine the three stored logical bits of a TLC NVM cell; however, it is important to note that the MSb was obtained using just the first strobe. This motivates NVM cell read solutions that can leverage the graded read latency between the MSb and the LSb to improve memory performance.


The iterative read sensing in MLC/TLC NVMs effectively creates a read-latency-based classification of the memory. In other words, MLC/TLC NVM, which stores 2/3 bits in each cell, has 2/3 speed grades for reads with MSbs being the fastest and LSbs being the slowest. Classical binary encoding encodes a linear stream of logical bits into MLC/TLCs by packing 2/3 logical bits at a time; this does not leverage the different read speed grades that are available and is always limited by the slow read speed of LSbs. In contrast, DBM [1] and SPCM [2] are two state-of-the-art schemes that leverage the read speed disparity by encoding different regions of memory address space using only MSbs/LSbs.

DBM proposes that by mapping MSbs/LSbs of MLC PCM to separate logical address blocks, contiguous blocks of fast-/slow-read memory can be created: DBM requires OS support for online profiling and prediction to map read-intensive pages to fast-read regions to improve read performance. However, these page mapping and migration mechanisms require OS support and incur additional read and write overheads, which decreases system performance and memory endurance. Also, as a consequence of DBM, the two sets of addresses in the row buffer are completely independent of each other (reduced spatial locality), which reduces memory performance due to increased row-buffer misses.
SPCM addresses some of the limitations of DBM by encoding odd and even address cache lines exclusively using MSbs and LSbs, respectively, for MLC PCM. Therefore, read requests to odd-address cache lines can be served faster in comparison to even-address cache lines. Furthermore, unlike DBM, the data stored in the row buffer corresponds to contiguous address chunks, which increases row-buffer hits due to spatial locality. When SPCM is extended to TLC NVM, a given cell encodes data from 3 consecutive cache lines (partner cache lines)—enabling read accesses of 3 separate speed grades. Therefore, each third of the memory addresses can be read using one of the three speed grades; this potentially increases the performance of two-thirds of the entire memory. However, the SPCM read latency improvements come at a steep cost of up to $3\times$ increase in the number of writes\textsuperscript{3}. The $3\times$ write overhead translates to $3\times$ increase in write energy and $3\times$ reduction in endurance. Additionally, in a power-limited NVM module the writes may need to be serialized, which translates the $3\times$ increase in writes to $3\times$ increase in write latency\textsuperscript{4}. Hence, the write latency, energy, and endurance overheads are potential show-stoppers for both DBM and SPCM. This motivates read-speed-grade solutions that result in improvements in read latency without any impact to write performance.

\textsuperscript{3}SPCM proposes that the $2\times$ write overhead for MLC PCM can be reduced using a buffer, which delays the write to a cache line until a write is issued to its partner line, such that the two partner lines can be updated together. The reasoning is that when a cache line is evicted from the last-level cache (to be written to the main memory), the chances of eviction of its partner line increase due to spatial locality. However, the chances of pairing all the partner lines reduce for denser technologies (TLCs and beyond). In this work, we evaluate the performance of SPCM using TLC RRAM for 2 cases—the best case, where the write overhead is zero, and the worst case, where the write overhead is $3\times$.

\textsuperscript{4} To prevent the modules from overloading the power supply, and to prevent overheating, NVM modules are restricted to writing a limited amount of data at a time, which in turn increases the write latency and lowers performance [25].
3.0 COMPEX AND COMPEX++ CODING: COMPRESSION-EXPANSION CODING FOR ENERGY, LATENCY, AND LIFETIME IMPROVEMENTS IN MLC/TLC NVM

Moving to MLC/TLC technology from SLC technology (and similarly to a TLC technology from an MLC technology) enables static tradeoffs between memory density/capacity and memory energy/latency. We motivated the technology considerations and the system level implications of these tradeoffs in Fig. 1. However, to the best of our knowledge, there has been no systematic effort to dynamically leverage these tradeoffs to realize the density/capacity benefits of an MLC NVM (TLC NVM) while also reaping the low energy/latency benefits of an SLC NVM (SLC/MLC NVM). In other words, we believe that the ability to integrate a dense high capacity MLC/TLC technology for the memory while dynamically operating that memory in the SLC/MLC mode (fully or partially, as developed in the theory of expansion codes in this section) has far-reaching implications for simultaneously improving NVM energy, latency, density/capacity, and lifetime (the lifetime improvement is indirect, due to a reduction in the P&V effort needed for memory operation). CompEx/CompEx++ coding, as instances of such a dynamic tradeoff approach, relies on the holistic integration of two independent/orthogonal but complementary areas of research (data compression and data coding). The core idea of CompEx/CompEx++ coding, developed over the rest of this section, is to apply expansion coding to selectively compressed data such that the resulting data in expansion coded form does not exceed the original data width.

Section 3.1 introduces two pattern-based compression techniques that can be used in CompEx coding — frequent pattern compression [46] and base-delta-immediate compression [47]. Section 3.2 provides a formal treatment of expansion coding. Section 3.3 describes CompEx coding from first principles with examples and discusses a practical architecture for
Table 1: 64-bit FPC patterns (inclusive of 3-bit prefix, indicated in red). Illustrative examples are given in col. 3 and 4. The last column shows the percentage of compressed words using a given pattern. On the whole, FPC can compress about 60% of write accesses for the benchmarks from SPEC CPU2006 [48] benchmark suite.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Pattern encoded</th>
<th>Example</th>
<th>Compressed example</th>
<th>Encoded size</th>
<th>Value space</th>
<th>Compression contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Zero run</td>
<td>0x0000000000000000</td>
<td>0x0</td>
<td>3 bits</td>
<td>1 value</td>
<td>29.7%</td>
</tr>
<tr>
<td>001</td>
<td>8-bit sign-extended</td>
<td>0x000000000000007F</td>
<td>0x8</td>
<td>11 bits</td>
<td>127 values</td>
<td>1%</td>
</tr>
<tr>
<td>010</td>
<td>16-bit sign-extended</td>
<td>0xFFFFFFFFFFFFF6B6</td>
<td>0x8B6</td>
<td>19 bits</td>
<td>65536 values</td>
<td>1%</td>
</tr>
<tr>
<td>011</td>
<td>Half-word sign-extended</td>
<td>0x0000000076543210</td>
<td>0xB6B6</td>
<td>35 bits</td>
<td>≈2^{32} values</td>
<td>1%</td>
</tr>
<tr>
<td>100</td>
<td>Half-word, padded with a zero half-word</td>
<td>0x7654321000000000</td>
<td>0x76543210</td>
<td>35 bits</td>
<td>≈2^{32} values</td>
<td>16.4%</td>
</tr>
<tr>
<td>101</td>
<td>Two half-words, each a byte sign-extended</td>
<td>0x86BEEF0000003CAB</td>
<td>0x8BEEF3CAB</td>
<td>35 bits</td>
<td>≈2^{32} values</td>
<td>8.3%</td>
</tr>
<tr>
<td>110</td>
<td>Word consisting of four repeated double bytes</td>
<td>0xCAFECAFECAFECAFE</td>
<td>0xCAFECAFE</td>
<td>19 bits</td>
<td>65534 values</td>
<td>1.1%</td>
</tr>
</tbody>
</table>

CompEx coding. Section 3.4 presents CompEx++ coding, which extends the energy/latency benefits of CompEx coding. Finally, Section 3.5 demonstrates the integration of CompEx++ coding with ECC and encryption.

### 3.1 COMPRESSION

Although CompEx/CompEx++ coding is agnostic to the choice of compression technique, good candidates for CompEx/CompEx++ coding should have features such as low latency, low overhead, high compressibility, and low complexity. Based on our survey of compression techniques, two pattern-based compression techniques — frequent pattern compression and base-delta-immediate compression — possess these desirable traits for integration into CompEx/CompEx++ coding.

#### 3.1.1 Frequent pattern compression (FPC)

FPC is a pattern-based compression scheme that leverages program data statistics to successfully compress a wide range of data. FPC was originally proposed for 32-bit data word compression in L2 caches to increase their memory capacity [46]. More recently, FPC was
extended to reduce bit-writes in NVMs [16]. In this work, we extend FPC to a 64-bit word using the patterns tabulated in Table 1. FPC maintains a table of seven most frequent data patterns, against which the incoming data is compared. When the incoming data matches one of the frequent patterns, it is compressed and stored along with a 3-bit prefix corresponding to the pattern which is represented by the column 1 of Table 1. A 1-bit tag is used to indicate whether the written data is its compressed/un-compressed form.

During a read access, the tag bit and prefix are used to uncompress the data to a full word. Columns 3, 4, and 5 of Table 1 shows examples for each of the data patterns along with their compressed form and compressed data size. Column 6 of Table 1 represents the range of data values that each pattern can compress and column 7 represents the percentage of data words compressed by each pattern (FPC cumulatively compresses about 60% of write accesses) in trace-based simulations of the SPEC CPU2006 [48] benchmark suite.

3.1.2 Base-Delta-Immediate (BΔI)

BΔI was originally proposed for on-chip cache data compression [47] by storing the compressed data using a ‘base’ (B) and ‘delta’ (∆), a series of offsets with respect to B. BΔI proposes that a cache-line \( C = \{V_0, V_1, ..., V_{n-1}\} \) can be compressed and written as \( X = \{B, \Delta_0, \Delta_1, ..., \Delta_{n-1}\} \) along with a 4-bit tag, where \( B = V_0 \) (definition), \( \Delta_i = V_i - B \), \( n = \text{sizeof(cache-line)}/k \), and \( k = \text{sizeof(\Delta)} \). It is reported that data stored in a cache line is often regular, with limited dynamic range [47]. Whereas the regularity in the data is due to the common use of array data structures to store program data, the limited dynamic range in the stored data is due to the nature of computation.

Similar to FPC, BΔI maintains a pattern table for compression. BΔI leverages regularity in data to compress cache lines using 64-byte patterns; table 2 shows the 8 different cache line patterns that BΔI is capable of compressing, along with illustrative examples for each of the patterns. The last column represents the percentage of data words compressed by

---

\(^1\)This work integrates BΔI with a single ‘base’, without the implicit second base, for simplicity. This is similar to the base+delta technique proposed in the same paper [47]. Furthermore, instead of using a 4-bit tag as proposed, we use a single tag bit to indicate whether the stored data is in compressed (tag=1) or uncompressed (tag=0) form. However, a 3-bit prefix (only for compressed data) is stored along with the compressed line to indicate the BΔI pattern that was used for compression.
each pattern (B∆I cumulatively compresses about 46% of write accesses) in trace-based simulations of the SPEC CPU2006 [48] benchmark suite. Given a cache line, it is matched against each row (pattern) of Table 2. If the cache-line data matches a pattern, then the compressed data along with the prefix is stored in the data memory; the tag bit is set to indicate that the data stored is in compressed format. In contrast, if the data is not compressed, then it is stored as-is in the uncompressed format; the tag bit is reset to record this information.

Due to the low overhead and high compression ratio of FPC and B∆I, this work is motivated by the potential of integrating one of these compression techniques with expansion coding for a holistic solution that reduces energy and latency without any additional memory overhead.
Table 2: 64-byte \( B \Delta I \) patterns (with 3-bit prefix, indicated in red) along with illustrative examples and encoded size post-compression. On the whole, \( B \Delta I \) can compress about 46% of write accesses for the benchmarks from SPEC CPU2006 \[48\] benchmark suite. Note that although the compression contributions of FPC and \( B \Delta I \) are comparable, the compression ratios are very different. For example, the bulk of the compression from ‘Zero values’ in both FPC and \( B \Delta I \) have comparable compression contributions. Whereas FPC compresses data from 64 bits to 3 bits, \( B \Delta I \) compresses 64 bytes (512 bits) to 3 bits. Hence we expect to see higher reduction in energy using \( B \Delta I \) in comparison to FPC. On the other hand, since latency is limited by the worst case cell-write delay, we expect latency improvement using \( B \Delta I \) to be comparable to that of FPC.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Pattern</th>
<th>Example</th>
<th>Compressed example</th>
<th>Encoded size</th>
<th>Compression contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Zeros</td>
<td>0x000000000000 000000000000 ... 000000000000</td>
<td>0x0</td>
<td>0 bytes</td>
<td>24.7%</td>
</tr>
<tr>
<td>001</td>
<td>Rep. values</td>
<td>0xFEEDC0FFEEFEEDC0FFEE ... FEEDC0FFEEFEEDC</td>
<td>0x1:FEEDC0FFEEFEEDC</td>
<td>8 bytes</td>
<td>6.8%</td>
</tr>
<tr>
<td>010</td>
<td>Base8-( \Delta 1 )</td>
<td>0xABCDABCDABCDAB00 ABCDABCDABCDAB01 ... ABCDABCDABCDAB08</td>
<td>0x2ABCDABCDABCD00,01,...,08</td>
<td>15 bytes</td>
<td>0.2%</td>
</tr>
<tr>
<td>011</td>
<td>Base8-( \Delta 2 )</td>
<td>0xABCDABCDABCD0000 ABCDABCABCD00FA ... ABCDABCABCD5A5A</td>
<td>0x3ABCDABCDABCD0000,00FA,...,5A5A</td>
<td>22 bytes</td>
<td>0.2%</td>
</tr>
<tr>
<td>100</td>
<td>Base8-( \Delta 4 )</td>
<td>0xABCDABCD00000000 ABCDABCDBEEFFEED ... ABCDABCDBEEFFEED</td>
<td>0x4ABCDABCD00000000, BEEFFEED,...,COFFEED</td>
<td>36 bytes</td>
<td>1.3%</td>
</tr>
<tr>
<td>101</td>
<td>Base4-( \Delta 1 )</td>
<td>0xABCDAB00 ABCDAB01 ABCDAB02 ... ABCDABFF</td>
<td>0x6ABCDAB00,01,02,...,FF</td>
<td>19 bytes</td>
<td>3.2%</td>
</tr>
<tr>
<td>110</td>
<td>Base4-( \Delta 2 )</td>
<td>0xABCD0000 ABACFF FEEDF00F ... ABACFF</td>
<td>0x6ABCD0000,FEED,F00F,...,EDDC</td>
<td>34 bytes</td>
<td>6.6%</td>
</tr>
<tr>
<td>111</td>
<td>Base2-( \Delta 1 )</td>
<td>0xAB00 ABFF ABED ABED ... ABDC</td>
<td>0x7AB00,FF,ED,EF,...,DC</td>
<td>35 bytes</td>
<td>2.7%</td>
</tr>
</tbody>
</table>

* First, the encoded size is 3 bits in addition to this column’s contents (for prefix). Second, since \( B=V_0 \) by definition, storing \( \Delta_0 \) is redundant and hence omitted.
3.2 EXPANSION CODING

As described in Section 2, the iterative P&V procedure used for programming MLC/T-LC NVMs results in the central MLC/TLC states requiring more energy and latency in comparison to the terminal states. This motivates data encoding using only the low energy states, avoiding the high energy states to reduce the overall energy and latency. This work uses expansion coding, which encodes data using only these lower energy states. The rest of this sub-section provides formal definitions for expansion coding and illustrates expansion coding with examples.

**Definition:** A \((k,m)_q\) expansion code, \(m < k\), is a linear block code with \(q\)-ary code words of length \(k\) encoding \(q^m\) \(q\)-ary message words. Expansion code encodes data using \(p\) lowest energy states of total \(q\) states \((p < q)\), where \(p = \lceil q^{m/k} \rceil\), incurring memory overhead of at least \((k/m) - 1\) [64, Ch. 3].

We elaborate on the \((k,m)_q\) expansion code with the help of Fig. 7. Given a technology where each cell can represent \(q\) states, we can encode \(\lceil \log_2 q \rceil\) bits of data in each cell. The total number of cells in each message is \(m\). Hence each message word is one of the \(q^m\) possible words, where each word is \(q\)-ary. This is represented using the left-hand-side of the Fig. 7. All these messages are mapped to \(k\) cells, where each cell can represent only \(p\) states logically (though physically they are still capable of storing \(q\) states). This is represented using the right-hand-side of the Fig. 7. The code expands a \(m\)-cell word to a \(k\)-cell word after encoding, and hence the name ‘expansion code’. Expansion coding lowers both NVM energy and latency in practice, since the lowest energy TLC states also require fewer P&V cycles (Fig. 3).

**Example:** For ease of understanding, we illustrate \((3,2)_8\) expansion coding and incremental data mapping (IDM) [10], which is an instance of \((6,5)_8\) expansion coding. First, consider an example of \((3,2)_8\) expansion code, which encodes 16-bit data using TLC RRAM as shown in Fig. 8. In the regular case, data is encoded using all 8 TLC states as shown in 8(a). Every slice of 3 logical data bits is stored in one TLC, requiring 6 \((= \lceil 16/3 \rceil)\) TLCs and 120.1pJ. In contrast, the \((3,2)_8\) expansion code uses only 4 \((\lceil 8^{2/3} \rceil)\) out of the 8 TLC states.
to encode the data (Fig. 8(b)). Every slice of 2 logical data bits is stored in one TLC, requiring \(8 (=\lceil16/2\rceil)\) TLCs and 39.2pJ. In this example, although expansion coding results in energy and latency reduction, it incurs 50% memory overhead. IDM [10] seeks to lower the memory overhead to 20% by using the \((6,5)_8\) expansion code utilizing the 6 lowest energy TLC states. Whereas binary coding encodes 3 information bits into each TLC, IDM encodes 5 data bits into 2 TLCs. Since 2 TLCs together can store a maximum \(6 \times 6 = 36\) states, they can easily encode the \(2^5 = 32\) states required for storing information from 5-bits. For a 16-bit data encoded using IDM (Fig. 8(c)), binary encoding uses 6 TLCs to represent 16 bits and requires 120.1pJ. However, using IDM, 16 data bits can be encoded using 7 TLCs (using 6 states) and 94.3pJ.

Although IDM in combination with dynamic data remapping has been shown to improve the lifetime of MLC/TLC NVMs [10], it provides marginal reduction in energy (up to 15%) for 20% memory overhead. In this work, we look to data statistics as a potential source of flexibility to realize the full energy and latency benefits of expansion coding for negligible memory and logic overhead.
Figure 8: Comparison of binary, $\text{(3,2)}_8$ expansion coding, and $\text{(6,5)}_8$ expansion coding (IDM [10]). Energy for $\text{(3,2)}_8$ expansion coding < $\text{(6,5)}_8$ expansion coding < binary coding.
3.3 COMPEX CODING

The core idea of this chapter is to apply expansion coding selectively to compressed data such that the resulting data in expanded form does not exceed the original data width. For example, if $n$-bit raw data is compressed to $\leq 2n/3$ bits, the $(3,2)_8$ expansion code will incur no memory overhead and also yield the full energy and latency benefits of expansion coding. However, in order to be successful in practice, the compression scheme must compress a large fraction of the memory traffic while also being lossless and simple to design with low performance overhead. FPC and B\(\Delta\)I are excellent candidates for this purpose due to their ability to compress a wide range of data for low overhead (refer Tables 1 and 2). Furthermore, since the largest size of an FPC-compressed 64-bit word (or B\(\Delta\)I-compressed 64-byte cache line) is only 35 bits (292 bits), we can easily layer $(3,2)_8$ expansion coding atop FPC without incurring additional memory overhead. In this work, all examples and evaluation of CompEx coding use the $(3,2)_8$ expansion code (and the $(6,5)_8$ code as necessary/appropriate).

**Example:** Without loss of generality, we illustrate CompEx coding using 16-bit word as an example, which is compressed to 8 bits using FPC as shown in Fig. 9. In the first case, we encode the data using all the 8 TLC states from Fig. 2(a). This allows us to encode data using just 3 of the 6 TLCs in the memory location for 97.0pJ. However, using CompEx coding, as shown in Fig. 9(b), the 8-bit compressed data can be encoded using the $(3,2)_8$ expansion code. This uses 4 of the 6 TLCs for 24.9pJ. This example illustrates the integration of FPC with expansion coding without any additional memory overhead. Therefore, for a 64-bit word, CompEx coding encodes an FPC-compressed 64-bit word to a maximum of 18 ($\lceil 35/2 \rceil$) TLCs. Similarly CompEx coding encodes a B\(\Delta\)I-compressed 64-byte cache-line to a maximum of 146 ($\lceil (36\times8+4)/2 \rceil$) TLCs.

**No tag overhead:** Although a single tag bit is necessary to record the outcome of CompEx coding, it is concatenated with the data (65 bits for FPC-based and 513 bits for B\(\Delta\)I-based Compex codec) and absorbed into the last TLC at no cost ($\lceil 65/3 \rceil = 22$ TLCs for FPC and $\lceil 513/3 \rceil = 171$ TLCs for B\(\Delta\)I). Consider the 16-bit example shown in Fig. 9(b), where the tag bit is appended to the end of the data word and encoded into the last TLC. Since $2^n$
Figure 9: Illustration of CompEx coding. The key take-away is that CompEx coding reduces energy for no overhead since the tag bit is encoded within the last TLC.

\( (n \geq 1) \) can never be a multiple of 3, the tag bit can always be encoded within the existing TLCs without any overhead for word sizes that are integer powers of 2. Furthermore, to ensure that the tag bit does not impact latency, we reserve state ‘7’ of the TLC to indicate CompEx-coded data.

### 3.3.1 Theoretical analysis of CompEx coding

As we have established, CompEx coding can be applied without additional memory overhead, we now theoretically estimate the energy reduction for TLC RRAM whose energy and latency numbers are given by Fig. 3. We start by calculating the average energy for TLCs with 8 states, followed by the average energy for only 4 lowest energy TLC states. First, we assume that during writes, all permitted states of a TLC are equally likely to be programmed to [41]. Thus, the average/expected energy for a TLC with 8 equally likely states, i.e., binary coding, is given by the sum of the entries for energy for all the states in Fig. 3 divided by 8 (= 16pJ for this case), i.e., \( E[\text{Energy}_{\text{binary}}] = \sum_{i=0}^{7} e_i = 16\text{pJ} \). Here, \( e_i \) is the energy required for programming the TLC RRAM to \( i^{\text{th}} \) state. Second, the average energy for a TLC that utilizes only 0, 1, 6, and 7 states, i.e., the \((3,2)_8\) expansion coding, is given by the sum of all energy for these subset of states divided by 4 (= 4.7pJ for this case), i.e., \( E[\text{Energy}_{(3,2)_8}] = \sum_{i=0,1,6,7} e_i = 4.7\text{pJ} \). Since \((3,2)_8\) expansion code uses \((3/2)\times\) more TLCs over binary encoding, we also need to factor this number into our energy computation. Therefore, the overall reduction in
Figure 10: Schematic diagram for FPC-based CompEx codec. (a) Encoder: The incoming 64-bit write-data is first encoded using the FPC encoder (latency = 2 cycles) followed by the expansion coding encoder (latency = 1 cycle) to obtain encoded data and tag bit. (b) Decoder: The tag bit and the encoded data are first decoded for expansion coding (latency = 1 cycle) and then decoded for FPC (latency = 1 cycle). Our implementation hides these latencies using memory access pipelining.

energy by using \((3,2)_8\) expansion code for this example is \(\frac{E[\text{Energy}_{\text{binary}}]}{E[\text{Energy}_{(3,2)_8}]} = \frac{16}{4.7} \cdot \frac{2}{3} \approx 2.3\). Additionally, since a fraction of the data is left uncompressed by FPC, the overall reduction in energy by using CompEx coding in comparison to binary encoding is \(\frac{E[\text{Energy}_{\text{binary}}]}{E[\text{Energy}_{\text{CompEx}}]} = 2.3\ p\), where \(p\) is the compression ratio of the compression scheme. Similarly, for \((6,5)_8\) expansion code, the average energy reduction can be derived to be \(1.4\times\) using the same procedure (not discussed here for brevity). Our evaluation of expansion coding using real-world workloads (discussed in Sec 5.7.1) closely agrees with the theoretical results, i.e, \(2.1\times/1.3\times\) practical energy reduction in comparison to \(2.3\times/1.4\times\) derived theoretically for \((3,2)_8/\(6,5)_8\) expansion coding.
Figure 11: CompEx codec architecture. The codec lies in the read/write data path adding 2/3 cycles overhead, respectively. However the latency reduction obtained due to CompEx coding far out weigh this overhead.

3.3.2 CompEx codec architecture

This section describes the architecture that integrates CompEx coding within the NVM module controller. The architecture \(^2\) for CompEx coding consists of the CompEx code encoder on the write-path and the CompEx code decoder on the read-path, as shown in Fig. 11. The CompEx codec (encoder-decoder) logic is embedded in the NVM module controller between the NVM array and the data bus to seamlessly encode and decode the accessed data as shown in Fig. 11. The CompEx code encoder is made up of compression logic followed by an expansion code encoder. Similarly, the CompEx code decoder is made up of an expansion code decoder followed by decompression logic. FPC-based CompEx coding uses word-size write/read accesses, while BΔI-based CompEx coding uses cache-line size accesses.

\(^2\)For a multi-channel memory system, each channel uses a dedicated codec, which results in a small overhead (Section 4.4) for the realized performance gains.
**Write:** Whenever the MLC/TLC NVM module receives a write access from the memory controller on the processor side, the CompEx codec handles them as follows. First, the incoming data is passed through the compression logic, which compares the data with all the compression patterns to attempt data compression. If the data is compressible, then the compressed data is passed through the expansion code encoder; uncompressible data are directly sent to the write circuit. The expansion code encoder encodes every 2-bit slice of the compressed data to 3-bit codewords. However, since the width of the encoded data is always less than 64 bits (for FPC) and 64 bytes (for BΔI), CompEx coding has zero memory overhead. Note that the tag bit may need to be updated to record the outcome of CompEx coding, regardless of whether it is successful or unsuccessful. But, as discussed earlier, this tag bit can be concatenated and absorbed into the data using the last TLC without requiring an extra cell for the tag bit.

**Read:** When the MLC/TLC NVM module receives a read access from the memory controller, the CompEx codec inside the NVM module controller decodes the data before forwarding it to the memory controller. The read circuit reads the cells of the whole word and forwards the data to the CompEx codec. The tag bit is recovered from the last TLC of the read word and used to determine whether the data has to be CompEx decoded or forwarded directly to the NVM module controller.

### 3.3.3 CompEx codec logic overhead

To estimate the logic overhead, we designed and synthesized 64-bit FPC-based CompEx codec (Fig. 10). The design of BΔI-based compressor is assumed from the original BΔI proposal [47]. The estimated logic overhead for the CompEx codec tabulated in Table 3 is ≈10k 2-input nand gates (< 0.1% per NVM module). Furthermore, although the CompEx codec has an estimated latency of 3/2 cycles for encoding/decoding, respectively, our implementation uses memory access pipelining to hide this in practice.
Table 3: Logic overhead for synthesized CompEx codec.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Logic</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-path</td>
<td>64-bit 2-to-1 multiplexer</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>3-bit subtractor</td>
<td>9</td>
</tr>
<tr>
<td>Write-path</td>
<td>16-bit comparator (equality check only)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2-bit comparator</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>1-bit 2-to-1 multiplexer</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>3-bit adder</td>
<td>9</td>
</tr>
</tbody>
</table>
This section presents CompEx++ coding, which extends the energy/latency benefits of CompEx coding by leveraging the variable compressibility of pattern-based compression schemes. Pattern-based compression schemes like FPC and BΔI use multiple compression patterns, which compress data to different sizes. This variability in compressed data size motivates us to explore a scheme with custom expansion code for each compression pattern as opposed to a single expansion code for the entire scheme. In other words, we design CompEx coding for each of the compression patterns separately, such that the custom expansion code along with the compression pattern maximizes the energy/latency advantages.

CompEx coding using FPC (BΔI), which compresses 64-bit (512-bit) data using 22 (171) TLCs to 3–35 bits (3–292 bits), as illustrated in Table 1 (Table 2). Although the (3,2)\textsubscript{8} expansion code is a good choice when we consider FPC or BΔI as a whole, it might not be the optimum choice for each pattern individually. In order to find the best choice for each of the patterns, we need to integrate a custom expansion code based on the compressed data size for each compression pattern.

**Example:** Without loss of generality, and for ease of understanding, consider a 16-bit word FPC example, which requires 6 TLCs for storage, as illustrated in Fig. 12. The incoming write data, as illustrated in Fig. 12(a), requires 6 TLCs and 77.2pJ. However, the write data is recognized to be a sign-extended half-byte that can be written using only 4 bits after compression. Note that although FPC requires the storage of a 3-bit prefix along with the data, we ignore the 3-bit prefix in this example for brevity. First, the compressed data is encoded using conventional binary encoding — encoding 3 bits per TLC — for 77.9pJ, as illustrated in Fig. 12(b). Second, the regular CompEx coding that uses (3,2)\textsubscript{8} expansion coding encodes the compressed data by packing 2 logical bits into each TLC; each 2-bit group is encoded using the 4 low energy states of the TLC (0, 1, 6, and 7) resulting in 16.7pJ, as illustrated in Fig. 12(c). Finally, since the compressed data size is about a quarter of the original size, it is possible to encode data using (3,1)\textsubscript{8} expansion coding. Figure 12(d) illustrates data encoding using (3,1)\textsubscript{8} expansion coding; this requires 5 TLCs for data encoding, which is less than the originally available 6 TLCs, for only 8.5pJ. Thus,
Figure 12: Illustration of CompEx++ coding using (3,1)\textsubscript{8} expansion coding, as opposed to (3,2)\textsubscript{8} expansion coding used in CompEx coding. The key take-away is that although CompEx coding reduces energy for no overhead, the optimum energy reduction can be achieved by custom expansion code for each compression pattern.

by making a smart choice of expansion codes (shown in Tab. 4) that is customized for each compression pattern, CompEx++ coding reduces the overall energy/latency expense over CompEx coding.

**Encoding the compression prefix:** Since CompEx coding encodes using only one type of expansion coding, encoding/decoding of the prefix is trivial—it depends only on the compression tag-bit. However, CompEx++ coding uses multiple types of expansion coding and hence it becomes important to carefully encode the prefix bits to eliminate ambiguity in decoding and latency overheads during encoding.

Figure 13 illustrates the encoding and decoding of compression prefixes. Since both CompEx coding and CompEx++ coding do not require any read circuit modifications, the encoded data is always decoded for conventional binary encoding, i.e., each TLC is decoded to be in one of the 8 possible states. Following the read circuit, the data is expansion-code decoded using the compression tag bit. However, in the case of CompEx++ coding it is not
Table 4: Chosen expansion codes for the compression patterns of FPC and BΔI.

<table>
<thead>
<tr>
<th>FPC pattern</th>
<th>Compressed size</th>
<th>Expansion code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero run</td>
<td>3 bits</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>8-bit sign-extended</td>
<td>11 bits</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>16-bit sign-extended</td>
<td>19 bits</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>Half-word sign-extended</td>
<td>35 bits</td>
<td>(3,2)$_8$</td>
</tr>
<tr>
<td>Half-word, padded with a zero half-word</td>
<td>35 bits</td>
<td>(3,2)$_8$</td>
</tr>
<tr>
<td>Two half-words, each a byte sign-extended</td>
<td>35 bits</td>
<td>(3,2)$_8$</td>
</tr>
<tr>
<td>Word consisting of four repeated double bytes</td>
<td>19 bits</td>
<td>(3,1)$_8$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BΔI pattern</th>
<th>Compressed size</th>
<th>Expansion code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zeros</td>
<td>0 bytes</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>Rep. values</td>
<td>8 bytes</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>Base8-Δ1</td>
<td>15 bytes</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>Base8-Δ2</td>
<td>22 bytes</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>Base4-Δ4</td>
<td>36 bytes</td>
<td>(3,2)$_8$</td>
</tr>
<tr>
<td>Base4-Δ1</td>
<td>19 bytes</td>
<td>(3,1)$_8$</td>
</tr>
<tr>
<td>Base4-Δ2</td>
<td>34 bytes</td>
<td>(3,2)$_8$</td>
</tr>
<tr>
<td>Base2-Δ1</td>
<td>35 bytes</td>
<td>(3,2)$_8$</td>
</tr>
</tbody>
</table>

It is clear if the data should be decoded for (3,2)$_8$ expansion coding or (3,1)$_8$ expansion coding. This ambiguity is resolved as follows: As we know, (3,2)$_8$ expansion coding encodes the 3-bit compression prefix using 2 TLCs and (3,1)$_8$ expansion coding encodes the same using 3 TLCs. If the first TLC among the 2/3 prefix-TLCs encode states 1 or 6, the data is decoded using (3,2)$_8$ expansion coding. In contrast, if the first of the 2/3 TLCs encodes states 0 or 7, the data is decoded using (3,1)$_8$ expansion coding. Such an encoding inherently prevents the compression prefix from becoming a latency bottleneck during encoding. Note that such a compression prefix encoding enables CompEx++ coding to encode a maximum of 4 compression patterns using each of (3,2)$_8$ and (3,1)$_8$ expansion coding. However, encoding more than 4 compression patterns using one type of expansion coding (for example, BΔI-based CompEx++ coding, as shown in Tab. 4) requires additional prefix bits.

3.4.1 Theoretical analysis of CompEx++ coding

We now evaluate the theoretical energy gains of CompEx++ coding using TLC RRAM as an example. We start by mathematically describing a pattern-based compression scheme and establish the bit write (cell-update) reduction for a given compression scheme. This is followed by the mathematical description of custom expansion codes for each of the compression patterns. Finally, we integrate the closed-form expressions for compression and expansion coding to obtain a closed-form expression for the energy reduction due to CompEx++ coding.
Compression: Consider a pattern-based compression scheme with \( n \) different compression patterns, which are represented by \( P_i \), where \( i = 0, 1 \ldots (n-1) \). The \( i^{th} \) pattern \( P_i \) has a compression ratio of \( r_i \) (\( 0 \leq r_i \leq 1 \)) and matches \( w_i \) (\( 0 \leq w_i \leq 1 \)) fraction of the whole write traffic. Such a compression scheme would reduce bit writes to \( \sum_{i=0}^{n-1} r_i w_i \) (\( \leq 1 \)) times the original bit writes. This implies that the compression scheme with low \( r \) (high compressibility) and high \( w \) (matching a large fraction of the traffic) can significantly reduce the bit writes.

Expansion coding: The optimum expansion code for the \( i^{th} \) compression pattern leverages the entire space recovered using compression, but does not spill over the original data size. This implies that for a data-word (cache-line) size of \( L \), for a word-level (cache-line-level) compression scheme, the ideal expansion code would be \( (L, r_i L)_8 \) expansion code. Furthermore, the \( (L, r_i L)_8 \) expansion code encodes data using only \( \lceil 8 r_i \rceil \) (refer Sec. 3.2 for details) low energy states (the set of these low energy states is defined as \( S_i \)) out of the 8 states for TLC RRAM. Therefore, the energy reduction from encoding a TLC using \( (L, r_i L)_8 \) expansion code is \( \sum_{j \in S_i} e_j / \sum_{j=0}^{8} e_j \), where \( e_j \) is the energy required to program a TLC to its \( j^{th} \) state.
Finally, since expansion coding increases the number of cell-updates by a factor of \((1/r_i)\), the closed-form expression for the energy reduction of a given expansion code is \((1/r_i) \times \sum_{j \in S_i} e_j / \sum_{j=0}^{8} e_j\).

**CompEx++ coding:** Integrating the mathematical forms of both pattern-based compression scheme and expansion coding schemes, the total energy reduction for CompEx++ coding is given by \(\sum_{i=0}^{n-1} w_i \times \{ \sum_{j \in S_i} e_j / \sum_{j=0}^{8} e_j \}\). Although in practice it might not always be possible to obtain expansion codes for all values of \(r_i\), the above expression serves as a theoretical upper bound for the energy reduction using CompEx++ coding.

### 3.4.2 CompEx++ codec logic overhead

Since CompEx++ coding requires multiple expansion code codecs, the logic overhead is higher in comparison to CompEx coding. However, it is important to note that since these expansion codecs are multiplexed (chosen) depending on the compression pattern, and do not appear in series; therefore, there is no additional latency penalty for CompEx++ coding over CompEx coding. Our evaluation of the 64-bit FPC CompEx++ codec shows that each additional expansion code codec requires \(\approx 2.5k\) 2-input nand gates. Therefore, along with the original 10k gates for the rest of the logic (CompEx coding), the total logic overhead for CompEx++ coding is \(\approx 12.5k\) 2-input nand gates. However, this might not be a steep price to pay if the energy gain from CompEx++ coding is significantly higher than CompEx coding.
This section discusses the integration of error correction support and encryption with CompEx/CompEx++ coding, while preserving the low energy and latency benefits of CompEx/CompEx++ coding. First, we describe the integration of error correction support with CompEx/CompEx++ coding. Second, we establish the improvement in NVM scrub overhead due to CompEx/CompEx++ coding using theory and evaluate the same using simulations. Finally, we discuss the compatibility of CompEx/CompEx++ coding with encryption.

### 3.5.1 CompEx++ECC (CompEx/CompEx++ coding with ECC)

MLC/TLC NVMs are susceptible to both soft and hard errors, which necessitates the use of error detection and correction (EDAC) techniques such as ECC [50], ECP [51], etc. Since a write operation may alter both the data and the EDAC fields, the benefits of CompEx/CompEx++ coding the data field may be nullified by high latency writes in the EDAC field. However, by smartly organizing data and EDAC bits, we can preserve the energy/latency benefits of CompEx/CompEx++ coding for no additional overhead.

NVMs with EDAC usually use separable coding techniques, i.e., the data field is stored separate from the EDAC field. Therefore, we propose that whenever the data field is compressible, the EDAC field be written in expansion-coded form — the additional cells required for expansion coding the EDAC field can be obtained from the residual cells after compression of the data field. From Tables 1 & 2, we observe that the worst-case compressed data size for FPC-based (BΔI-based) CompEx/CompEx++ coding is 35 (292) bits, which corresponds to 18 (146) TLCs out of the available 22 (171) TLCs. Therefore, FPC-based (BΔI-based) CompEx/CompEx++ coding leaves 4 (25) TLCs unused per word (cache line) in the worst case, which can be purposed to expansion-code the EDAC field to preserve/extend the latency/energy benefits of CompEx/CompEx++ coding.

Without loss of generality and for ease of understanding, consider a 64-bit word, 8-bit ECC example, as illustrated in Fig. 14. The 64-bit data and the 8-bit ECC require
Figure 14: Illustration of ECC in CompEx/CompEx++ coding. The key take-away is that CompEx/CompEx++ coding can preserve its energy and latency benefits of in the presence of ECC for no additional overhead.

\[ \left\lceil \frac{64}{3} \right\rceil + \left\lceil \frac{8}{3} \right\rceil = 25 \text{ TLCs using conventional binary encoding, as illustrated in Fig. 14.} \]

If the data is compressible, then the compressed data field requires at-most \( \left\lceil \frac{35}{2} \right\rceil = 18 \) TLCs, while the 8-bit ECC requires \( \left\lceil \frac{8}{2} \right\rceil = 4 \) TLCs using \( (3,2)_8 \) expansion coding, i.e., one additional TLC in comparison to the conventional binary encoding. However, the additional TLC required to encode the 8-bit ECC can be repurposed from extra TLCs remaining unused in the data field after encoding using CompEx/CompEx++ coding, as illustrated in Fig. 14.

In contrast, if the data field is incompressible, then both the data and EDAC fields are written using conventional binary coding. Thus, CompEx/CompEx++ coding can be used even in the presence of ECC, while preserving its energy and latency benefits.

### 3.5.2 Improvement in NVM scrub interval due to CompEx++ECC

As we have already established, CompEx/CompEx++ coding can easily support ECC without compromising its energy/latency benefits. In addition to supporting the existing error correction capabilities, CompEx/CompEx++ coding can further increase the error correction capability by leveraging the unused TLCs (after accommodating both data and EDAC fields), shown in gray in Fig. 14. Therefore, FPC-based (BΔI-based) CompEx/CompEx++ coding can use a 14-bit (94-bit) EDAC field, as opposed to the 8-bit (64-bit) EDAC field resulting in stronger error correction capability. This increase in the error-correction strength, not only improves NVM lifetime but also system performance as follows.
Figure 15: NVM scrub to mitigate errors due to resistance drift in binary encoding and CompEx/CompEx++ coding. The key take-away is that CompEx/CompEx++ coding increases useful time and reduces downtime for no additional cost.

MLC/TLC NVMs suffer from resistance drift, where a programmed cell-resistance tends to continuously increase (drift) due to the ambient temperature. If the resistance drift is left unchecked, it moves the cell resistance to a different range altogether, which leads to data corruption. In order to prevent such data corruption, it is proposed that the NVM be periodically refreshed (popularly known as scrubbing in literature) — the data is read, corrected for errors, and re-written to memory [50]. Although periodic scrubbing of NVMs can mitigate errors, it incurs performance penalties due to the mandatory reads and writes. Whereas stronger ECC can reduce the scrub intervals, this is at the expense of memory overhead. CompEx++ coding on the other hand has the potential to reduce the performance overhead due to periodic scrub for no overhead. CompEx/CompEx++ coding encodes about 46%–60% of the data residing in the memory using (3,2) \textsuperscript{s} expansion coding. The fraction of the memory encoded using CompEx/CompEx++ coding enjoys superior error correction strength \(^3\) due to stronger ECC, which improves its resilience towards resistance drift. Therefore, the scrub period for the data encoded using CompEx/CompEx++ coding can be much lower in comparison to binary encoded data, as shown in Fig. 15. Note that strengthening

\(^3\)FPC-based CompEx/CompEx++ coding extends existing error correction capability — single error correction double error detection (SECDED) — to double error correction per word. Similarly, B\(\Delta\)I-based CompEx/CompEx++ coding enables correction of 3 additional errors per cache line over the existing error correction capability.
error correction capability using the memory cells recovered after compression is not a contribution of our work — several earlier schemes have leveraged compression for improved reliability \([65, 66]\). We integrate the principles from such schemes to evaluate not just the improvement in error correction capabilities of CompEx/CompEx++ coding, but also its effect on the scrub rate, which directly impacts system performance and availability.

**Theory:** This section builds the required theoretical framework for understanding and evaluating CompEx++ECC. Consider a \(k\)-bit data line and an \(m\)-bit EDAC field, which together constitute an \(n\)-bit \((n = k + m)\) cache line stored using TLC NVM. Let \(p(t)\) be the soft error rate, which is defined as the probability that an NVM cell is not in its originally programmed state; where \(t\) is the time elapsed since the cell was last was programmed. Let us consider an ECC scheme, which can correct up to \(n_e\) soft errors — a stronger ECC will correspond to a higher numerical value of \(n_e\), and vice versa. Assuming that the soft errors are independent and uncorrelated \([50]\), we use the principles of binomial distribution to find the uncorrectable block error rate (UBER), which is the probability that a cache line (block) has more than \(n_e\) errors, as follows:

\[
UBER(t) = 1 - \sum_{i=0}^{n_e} \binom{n}{i} p(t)^i (1-p(t))^{(n-i)}
\]  

(3.1)

Whereas a programmed NVM cache line exhibits almost no errors immediately after the time of programming, the probability of error continuously increases as time progresses. Thus, \(UBER(t)\) is a monotonically increasing function of \(p(t)\), which is in turn a monotonically increasing function of time due to resistance drift. When a cache line accrues more errors than the ECC capability, the data in the cache line is of no use for all practical purposes; such an event triggers expensive system-level memory error exceptions resulting in program rollbacks, program termination, etc. Thus, the upper limit on \(UBER(t)\), \(UBER_{\text{max}}\), is an important system-level design parameter (\(UBER_{\text{max}} = 10^{-10}\) in this work), which constrains the maximum time between successive scrubs, i.e., \(t_{\text{scrub}} = UBER^{-1}(UBER_{\text{max}})\).

**Evaluation:** In the rest of this section, we compare and contrast the scrub intervals across various error correction schemes for TLC PCM \([67, 68]\) using equation 1; this is tabulated in Tab. 5. Although the rest of the chapter uses TLC RRAM as the example technology, we
use TLC PCM in this section due to the lack of available data for RRAM reliability. First, we establish our baseline, binary encoded Hamming code, which is the standard (72,64) SECDED with 12.5% memory overhead for storing ECC bits. SECDED can correct one error in every 72-bit (24-TLC) codeword, which translates to a scrub interval of 2s. Second, we evaluate binary encoded BCH code – a cache line level (576,512) ECC – which can correct up to 6 soft errors in a cache line; this corresponds to a scrub interval of 5.92s, which is an improvement of $2.96 \times$ over the baseline. Third, we evaluate CompEx++ECC using Hamming code; since a CompEx/CompEx++ encoded data word can spare a total of 14 bits for ECC, which is 6 bits more than the baseline, a stronger Hamming code is possible – double error correction and double error detection (DECDED). Such a stronger code results in a higher scrub interval of 2.36s for the CompEx/CompEx++ encoded data. Assuming 50% of the incoming data patterns to be compressible, CompEx++ECC Hamming code can improve the scrub interval by $1.09 \times (2 + 2.36)/2$. Fourth, we similarly evaluate CompEx++ECC BCH code, which can correct up to 9 soft errors in a cache line as opposed to 6 errors in binary encoded BCH code; this results in an extended scrub interval of 8.33s, which is a $4.16 \times$ improvement over the baseline. Finally, we evaluate Free ECC [65] and Compress and Protect (COP) [66], which are the state-of-the-art compression based ECC schemes. Both Free ECC and COP integrates the standard (72,64) SECDED Hamming code, which results in a scrub interval equal to that of the baseline; however, it is important to note that they do so for no overhead. Thus, we conclude that CompEx++ECC can improve the scrub interval of the memory system, which improves the system performance and availability.

### 3.5.3 CompEx++ coding and encryption

NVM non-volatility has emerged as a serious data security concern [42,62,69] — the stored data continues to persist in the memory even after the system is powered down, exposing sensitive data to a malicious intruder. The dominant proposal to thwart such attacks advocates processor-side data encryption before the data is written to the NVM main memory. However, encryption scrambles the data and potentially reduces the data regularity that is
Table 5: **Scrub intervals for different ECC schemes and their corresponding memory area overheads.**

<table>
<thead>
<tr>
<th>ECC scheme</th>
<th>Scrub interval (s)</th>
<th>Scrub overhead improvement</th>
<th>ECC overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary encoded Hamming code</td>
<td>2</td>
<td>1×</td>
<td>12.5%</td>
</tr>
<tr>
<td>Binary encoded BCH</td>
<td>5.92</td>
<td>2.96×</td>
<td>12.5%</td>
</tr>
<tr>
<td>CompEx++ECC Hamming code</td>
<td>2.18</td>
<td>1.09×</td>
<td>12.5%</td>
</tr>
<tr>
<td>CompEx++ECC BCH</td>
<td>8.33</td>
<td>4.16×</td>
<td>12.5%</td>
</tr>
<tr>
<td>Free ECC [65]</td>
<td>2</td>
<td>1×</td>
<td>0%</td>
</tr>
<tr>
<td>COP [66]</td>
<td>2</td>
<td>1×</td>
<td>0%</td>
</tr>
</tbody>
</table>

necessary for CompEx/CompEx++ coding. Thus, if CompEx/CompEx++ coding follows encryption, there would be no energy/latency benefits. Therefore, we propose a key modification to the CompEx-encryption data path, which allows CompEx/CompEx++ coding to retain its energy/latency benefits even in the presence of data encryption. We propose to incorporate the following sequences for write and read data paths: On a write, the write data is first compressed, and then the compressed data is encrypted in the processor-side memory controller. The resulting compressed encrypted data, which is smaller than the original word/cache line size due to compression, is stored in the memory using expansion coding (inside the NVM DIMM). On a read, the data stored in the memory is first decoded for expansion coding (inside the NVM DIMM), which is followed by decryption and decompression in the processor-side memory controller. Thus, data encryption can be seamlessly integrated with CompEx/CompEx++ coding to achieve data security without compromising the energy/latency benefits of CompEx/CompEx++ coding. Although data encryption is a robust data security solution, the data after encryption results in a high cell-update overhead due to its high randomness. Furthermore, the move from SLC to MLC to TLC increases the average cell-updates from 50% to 75% to 87.5%, respectively, which translates to prohibitively high energy and latency overheads. Therefore, although CompEx/CompEx++ coding is compatible with data encryption, we believe that translating data security solutions from SLC to MLC to TLC is best addressed as a subject of future research.
3.6 METHODOLOGY

Our evaluation of CompEx++ coding is based on (i) full-system simulations to evaluate the system-level performance and (ii) trace-based simulations for deep, multi-billion instruction evaluation of memory-level energy and latency of CompEx++ coding.

3.6.1 Full-system simulation

CompEx++ coding is evaluated using full-system simulations of a system that integrates TLC RRAM memory using MARSS [12], a full-system multi-core simulator, and DRAM-Sim2 [13], a cycle-accurate main memory simulator.

MARSS uses x86 core models from PTLSim [70], a cycle-accurate x86 micro-architecture simulator and plugs it into QEMU [71], a binary-translation system for emulating full systems. QEMU provides the capability of emulating various I/O devices (HDD, ethernet, HID, etc.) that can be used to boot up entire operating systems without any modification (Linux in this work). Note that in this work, we modify MARSS to propagate write data along with the address throughout its memory hierarchy.

We use DRAMSim2, a cycle-accurate main memory simulator for simulating the DDR3 NVM main memory system. MARSS and DRAMSim2 are integrated to provide a monolithic, seamless, cycle-accurate simulation of the entire system. Since each access in a TLC NVM memory can potentially have different access latencies, we modify DRAMSim2 to account for this.

MARSS setup: MARSS was configured to simulate a standard 4-core out-of-order system running at 3GHz. Each core has its own L1 cache with 2 separate instances of 32kB SRAM for data and instructions; the L2 cache is private, with each core having its own instance of 256kB SRAM; finally, the L3 cache is a single, shared, write-back cache of size 8MB. The latencies of each level of cache is tabulated in Table 6.

DRAMSim2 setup: For accurate timing simulation, we modified the DDR timing parameters along the lines of [3] for substituting DRAM with NVM. We use TLC RRAM with latency parameters extracted and summarized in [10].

45
Table 6: **Configuration of the evaluation architecture.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Attributes</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>4 cores, dual issue out-of-order</td>
<td>—</td>
</tr>
<tr>
<td>L1 (instruction)</td>
<td>32kB, 2 way</td>
<td>2ns</td>
</tr>
<tr>
<td>L1 (data) cache</td>
<td>32kB, 2 way</td>
<td>2ns</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256kB, private, 8 way</td>
<td>5ns</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8MB, shared, 16 way</td>
<td>20ns</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
<td>—</td>
</tr>
<tr>
<td>Main memory</td>
<td>16GB, 8 banks, 1 channel</td>
<td>160ns</td>
</tr>
<tr>
<td></td>
<td>(DRAMSim2), 1GHz</td>
<td>(average)</td>
</tr>
</tbody>
</table>

**Workloads:** We evaluate CompEx++ coding using SPEC CPU2006 [48] benchmark suite, which reflect a variety of integer and floating-point workloads used by modern computing systems. To evaluate real-world usage, we use 9 composite workloads with each workload containing 4 SPEC CPU2006 benchmarks. These composite workloads are derived from [72], where benchmarks are selectively picked due to their memory intensive nature. Table 7 lists the constituent benchmarks for each composite workload and their corresponding writes per kilo-instruction (WPKI) and misses per kilo-instruction (MPKI) as reported by MARSS.

### 3.6.2 Trace-driven simulation

For running deep, multi-billion-instruction simulations, CompEx++ coding is evaluated using both an in-house trace-driven simulator for evaluating the memory array level dynamic energy and NVMain [52] — an architectural-level main memory simulator for emerging NVMs — for evaluating the total energy at the memory module level. We modified NVMain to reflect the variable-write-latency behavior of CompEx++ coding and also configured NVMain to simulate an architecture equivalent to that in Table 6. The traces are generated from the SPEC CPU2006 [48] benchmark suite using Intel Pin binary instrumentation tool [53] on a machine running a 3.3 GHz Intel Core i7 CPU. Note that we also use Gem5 system simulator [73] to validate these results using an equivalent architecture; the results are consistent with what is reported here and not discussed for brevity. Our simulation
Table 7: Composite workloads comprising of 4 benchmarks from SPEC CPU2006 benchmark suite (derived from [72]) for full-system evaluation of CompEx++ coding. The WPKI and MPKI numbers are sensitive to the CPU and memory system architecture; the numbers presented in this table are compiled for the architecture defined in Table 6 using MARSS.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Constituent benchmarks</th>
<th>WPKI</th>
<th>MPKI</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD1</td>
<td>lesie3d, lesie3d, mcf, mcf</td>
<td>4.61</td>
<td>9.66</td>
</tr>
<tr>
<td>WD2</td>
<td>lbm, lesie3d, libquantum, mcf</td>
<td>6.01</td>
<td>12.55</td>
</tr>
<tr>
<td>WD3</td>
<td>lbm, lbm, libquantum, libquantum</td>
<td>8.60</td>
<td>17.57</td>
</tr>
<tr>
<td>WD4</td>
<td>bwaves, lesie3d, omentpp, sphinx3</td>
<td>3.68</td>
<td>7.95</td>
</tr>
<tr>
<td>WD5</td>
<td>GemsFDTD, libquantum, milc, zeusmp</td>
<td>3.72</td>
<td>7.93</td>
</tr>
<tr>
<td>WD6</td>
<td>GemsFDTD, libquantum, milc, milc</td>
<td>5.00</td>
<td>10.44</td>
</tr>
<tr>
<td>WD7</td>
<td>bzip, libquantum, milc, omentpp</td>
<td>7.41</td>
<td>15.32</td>
</tr>
<tr>
<td>WD8</td>
<td>cactusAMD, gcc, gobmk, zeusmp</td>
<td>4.48</td>
<td>10.07</td>
</tr>
<tr>
<td>WD9</td>
<td>astar, gobmk, hmmer, soplex</td>
<td>5.01</td>
<td>10.80</td>
</tr>
</tbody>
</table>

framework captures memory accesses from the processor, recording only those accesses sent to main memory. During trace generation, the benchmarks are first run through $5 \times 10^5$ memory writes, to ignore the write accesses from program initialization; they are then run until $4 \times 10^6$ memory write operations (equivalent to about 4 billion instructions on average) have been recorded or until the program terminates.

3.7 EVALUATION AND RESULTS

This section presents the evaluation of CompEx/CompEx++ coding at the memory and system levels. First, we present the energy and latency results at the memory level for different encodings — baseline (binary encoding with DCW), compression techniques (FPC and BΔI), expansion coding ((3,2)$_8$ and (6,5)$_8$), CompEx coding (FPC-based and BΔI-based CompEx coding, using (3,2)$_8$ expansion coding), and CompEx++ coding (FPC-based and BΔI-based). Second, we present the results for system-level evaluation of CompEx/CompEx++ coding; primarily to evaluate the impact of latency improvements of CompEx/CompEx++ coding on system performance.
3.7.1 Memory energy/latency

**Summary:** Table 8 summarizes and compares the total module energy, memory array dynamic energy, latency, and overhead for the 9 encoding techniques considered in this chapter. In summary, FPC-based and $B\Delta I$-based CompEx coding reduces the memory array write energy by 33% and 76%, write latency by 22% and 25%, respectively, in comparison to binary encoding for no overhead. The energy and latency benefits of CompEx coding are further extended by CompEx++ coding; FPC-based and $B\Delta I$-based CompEx++ coding reduces the memory write energy by 1.2% and 9.3% and write latency by 0% (no improvement) and 6.7% in comparison to FPC-based and $B\Delta I$-based CompEx coding, respectively.

**Energy:** Our simulation framework tracks all the cell writes that occur from the beginning of program execution to compute the cumulative energy required. It is important to note that although CompEx/CompEx++ coding does not require any additional memory overhead in comparison to classical binary encoding, the static energy from peripheral circuits and the memory array are indirectly influenced by the reduction in the latency of each write operation. A lower write latency translates to a lower energy expense to keep the peripheral circuits active, which is evaluated using NVMain [52]. Figures 16(a) and 16(b) show the memory array dynamic energy and the total memory module energy, respectively, for FPC, $B\Delta I$, $(3,2)_8$ and $(6,5)_8$ expansion coding, and FPC-based and $B\Delta I$-based CompEx coding, normalized to binary coding. The last entries of Fig. 16 represent the geometric mean of the energy reduction across all the benchmarks, which is equivalent to simulating all these benchmarks for the same execution time. Note that all the cases use classical read-modify-write (DCW) [15] for writing only the modified cells to the NVM array.

Our simulations show that FPC-based CompEx coding reduces the memory array dynamic energy (total energy) by 33% (18%) and 15% (16%) in comparison to binary coding and FPC, respectively, while $B\Delta I$-based CompEx coding reduces energy by 76% (57%) and 16% (25%) in comparison to binary coding and $B\Delta I$, respectively. Additionally, $(3,2)_8$ and $(6,5)_8$ expansion coding in isolation show a reduction of $2.1\times$ and $1.3\times$ in memory array write energy in comparison to binary coding, which is in close agreement with the theoretical estimate of $2.3\times$ and $1.4\times$, respectively, as derived in Section 3.3.1. Furthermore, our
trace-based evaluations show that FPC-based and BΔI-based CompEx++ coding extends the energy benefits of CompEx coding by 0% (1.2%) and 16% (7%), respectively, in terms of dynamic (total) energy. CompEx++ coding definitely shows improvements in energy over CompEx coding, however, the difference in improvements is quite small for FPC-based CompEx++ coding. The reason for this small difference energy numbers is because of the low percentage of pattern-match for FPC patterns that can leverage (3,1)$_8$ expansion coding. However CompEx++ coding can potentially have larger influence in comparison to CompEx coding for compression schemes that offer higher (3,1)$_8$ expansion-coding-possibilities, as shown by the BΔI-based CompEx++ coding.

**Latency:** We evaluate the impact on memory array write latency due to CompEx++ coding. As detailed in Section 2, the states with lower energy also require lower write latencies due to the iterative P&V procedure. Whereas the energy required for writing a word is given by the sum of the energy required for all the cells, the latency for writing a word depends on the cell that requires the longest latency. Therefore, since CompEx++ coding encodes data using only lower energy states, the program latency for writing compressed data is also reduced. Our simulation determines the latency for each write access individually by tracking the maximum latency cell-write for the word access. The write latency for each access is then cumulatively computed to obtain the overall latency for program execution. Fig. 17 shows that FPC-based CompEx coding is able to reduce the overall write latency by 22% and 19% in comparison to binary coding and FPC, respectively, and BΔI-based CompEx coding reduces overall write latency by about 25% and 26% in comparison to binary coding and BΔI, respectively. Furthermore, CompEx++ coding extends the latency benefits of CompEx coding by 0.6% and 6.7% over FPC-based and BΔI-based CompEx coding, respectively. Since the latency improvements, as observed using trace-based evaluations, of FPC-based CompEx++ coding over FPC-based CompEx coding is marginal, we do not evaluate FPC-based CompEx++ coding separately using full-system simulations.
Figure 16: (a) Dynamic energy at the memory array level and (b) total memory module energy results (using NVMain [52]) for FPC, (3,2)_8 expansion coding, (6,5)_8 expansion coding, FPC-based CompEx coding, and B∆I-based CompEx coding, normalized to classical binary coding. FPC-based and B∆I-based CompEx coding reduce dynamic energy (total energy) by 33% (18%) and 76% (57%) in comparison to binary encoding. As expected (in Table 2), the dynamic energy performance of B∆I-based CompEx coding is better than FPC-based CompEx coding. Furthermore, FPC-based CompEx++ coding and B∆I-based CompEx++ coding reduces the dynamic (total) energy by 33% (18%) and 88% (61%) in comparison to binary encoding, respectively, which is 0% (1.2%) and 16% (7%) over CompEx coding, respectively.
Figure 17: Latency results for FPC, \((3,2)_8\) expansion coding, \((6,5)_8\) expansion coding FPC-based CompEx coding, and BΔI-based CompEx coding, normalized to classical binary coding. FPC-based and BΔI-based CompEx coding reduce latency by 22% and 25% in comparison to binary encoding. As expected (in Table 2), the latency performance of FPC-based CompEx coding and BΔI-based CompEx coding are comparable to each other. Furthermore, FPC-based and BΔI-based CompEx++ coding extends the latency improvements by 0.6% and 6.7%, respectively, over CompEx coding.

Table 8: Comparison of memory array write energy, latency, and overhead normalized to classical binary coding for the 7 schemes considered in this chapter. Note that all the cases use classical read-modify-write (DCW) [15] to update only the modified cells in the NVM array.
Figure 18: IPC results from full-system simulation (MARSS+DRAMSim2). The last bar in each workload represents the arithmetic mean of the IPCs of individual benchmarks. The last set of bars represent the harmonic mean of IPCs for each workload. FPC-based CompEx coding, BΔI-based CompEx coding, and BΔI-based CompEx++ coding show an IPC improvement of 6.3%, 5.1%, and 10.6% over binary encoding with DCW.

3.7.2 Full-system evaluation

In this section, we discuss the impact of CompEx++ coding on system performance. We use two metrics: (i) instructions-per-cycle (IPC), which is a metric for full-system performance and (ii) main memory bandwidth, which is a metric for main memory performance.

Summary: Table 9 summarizes and compares IPC and bandwidth for binary encoding, (3,2)₈ expansion coding, FPC-based CompEx coding, BΔI-based CompEx coding, and BΔI-based CompEx++ coding. We do not evaluate FPC-based CompEx++ coding separately using full-system simulations since the trace-based simulations of the same resulted in negligible latency improvement over FPC-based CompEx coding. (3,2)₈ expansion coding, FPC-based, BΔI-based CompEx coding, and BΔI-based CompEx++ coding improves IPC by 34%, 6.3%, 5.1%, and 10.6%, and memory bandwidth by 76%, 11.1%, 12.6%, and 19.9% in comparison to binary encoding, respectively. In addition, (3,2)₈ expansion coding, which is an upper bound for CompEx coding, improves IPC by 34% over binary encoding.
Table 9: IPC and bandwidth for FPC-based CompEx coding, BΔI-based CompEx coding, and BΔI-based CompEx++ coding, normalized to classical binary encoding. Note that all the cases use DCW [15] to update only the modified cells in the NVM array.

<table>
<thead>
<tr>
<th>Code</th>
<th>IPC</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary coding</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>(3,2)_h expansion coding</td>
<td>134%</td>
<td>176%</td>
</tr>
<tr>
<td>FPC-based CompEx coding</td>
<td>106.3%</td>
<td>111.1%</td>
</tr>
<tr>
<td>BΔI-based CompEx coding</td>
<td>105.1%</td>
<td>112.6%</td>
</tr>
<tr>
<td>BΔI-based CompEx++ coding</td>
<td>110.6%</td>
<td>119.9%</td>
</tr>
</tbody>
</table>

**Instructions per cycle (IPC):** To evaluate the impact of CompEx/CompEx++ coding on IPC, we use a full-system simulator based on MARSS [12] and DRAMSim2 [13]. The simulation setup is described in detail in Section 5.6. We simulate nine composite workloads from Table 7. Figure 18 shows the IPC for each benchmark in each workload. The last set of bars in each workload in Fig. 18 represents the mean IPC for that workload (since each benchmark is run on a separate and exclusive core, the mean IPC is given by the arithmetic mean of the IPCs of the constituent benchmarks of the workload [74, Ch. 1]). The last set of bars in Fig. 18 represents the harmonic mean of the IPCs of all the workloads (computing the harmonic mean is equivalent to running all the workloads in the same system for a fixed number of instructions).

First, our simulations show a good correlation between IPC and both WPKI and MPKI. For example, consider the workloads WD_3 (highest MPKI) and WD_5 (lowest MPKI) — as expected, the high cache miss-rate of WD_3 lowers the IPC in comparison to WD_5 that has a lower cache miss-rate. To understand the dependence of IPC on WPKI, let us consider the workloads WD_7 (highest WPKI) and WD_4 (lowest WPKI). CompEx coding reduces latency and improves IPC only during write accesses. Thus, the workload with higher WPKI should show a higher improvement in IPC in comparison to the workload with lower WPKI; our simulations, in agreement with the above argument, show that FPC (BΔI)-based CompEx coding improves the IPC by 17.5% (14.8%) for WD_7 (high WPKI) in comparison to 5.5% (1%) for WD_4 (low WPKI). Second, our simulations also show similar correlations at the individual benchmark level. Intuitively, memory-intensive benchmarks like milc (WD_6) or
Ibm (WD₃) should have lower IPC in comparison to less memory-intensive benchmarks like gemsFDTD (WD₆) or omnetpp (WD₄); our simulations are in excellent agreement with the above argument. Finally, our simulations show that the overall IPC improvement using FPC-based, BΔI-based CompEx coding, and BΔI-based CompEx++ coding is about 6.3%, 5.1%, and 10.6%, respectively, in comparison to classical binary encoding. In addition, (3,2)₈ expansion coding, an upper bound for CompEx coding, improves IPC by 34% over classical binary encoding.

**Memory bandwidth:** Fig. 19 shows the main memory bandwidth across nine composite workloads for the baseline and CompEx/CompEx++ coding in comparison to classical binary encoding. In Fig. 19, the bars represent FPC-based CompEx coding and BΔI-based CompEx coding normalized to the baseline (binary encoding with DCW). The last set of bars in Fig. 19 represents the geometric mean (GM) of the normalized improvements (computing the GM is equivalent to running each workload for the same execution time). Intuitively, similar to IPC, workloads that have high WPKI should contribute to higher bandwidth improvements in comparison to those with low WPKI. This can be seen using the example of workloads WD₆/WD₇ (high WPKI), which show an improvement of 30% in bandwidth using BΔI-based CompEx coding in comparison to WD₅ (low WPKI), which shows an improvement of only 6%. On the whole, our simulations show that FPC-based, BΔI-based CompEx coding, and BΔI-based CompEx++ coding improve the average memory bandwidth by 11.1%, 12.59%, and 19.9% respectively, in comparison to binary encoding. Furthermore, the bandwidth improvement of (3,2)₈ expansion coding, which is an upper bound for CompEx coding, over classical binary encoding is 76%.

**Lifetime:** In this dissertation, we theoretically evaluate the lifetime gains of CompEx/CompEx++ coding using TLC RRAM as an example. [75] discusses the three primary mechanisms for cell failure in RRAM in detail and each mechanism contributes to limit the lifetime of a cell; at the memory array level, the lifetime of an RRAM cell is specified as a limit on the

---

⁴We see a good correlation between the trace-based latency results and the full-system IPC results. Benchmarks like milc and libquantum that have better improvement in latency using FPC-based CompEx coding in comparison to BΔI-based CompEx coding (from trace-based simulations) have also shown better IPC numbers for FPC-based CompEx coding in comparison to BΔI-based CompEx coding. On the other hand, benchmarks like sphinx3 and gobmk show that BΔI-based CompEx coding provides better improvement in latency and IPC in comparison to FPC-based CompEx coding.
number of programming cycles (SET/RESET) it can endure before the cell becomes dysfunctional [75]. Therefore, the lifetime of MLC/TLC RRAM is lower in comparison to SLC RRAM since programming an MLC/TLC RRAM requires higher number of P&V cycles in comparison to SLC RRAM [8]. Since CompEx/CompEx++ coding effectively reduces the number of P&V cycles for programming an NVM cell (by limiting cell states to only low energy/latency states), it also improves the lifetime of the MLC/TLC NVM. Trace-based lifetime evaluations using TLC RRAM — along the lines of [8, 9, 44] — show that CompEx/CompEx++ coding increases the lifetime by $1.8\times$ over binary coding.

### 3.8 RELATED WORK

MLC/TLC NVMs definitely benefit from the broad set of solutions developed to improve energy, latency, and lifetime of SLC NVMs [76]. However, SLC-based solutions do not address the energy/latency problems of MLC/TLC NVMs that is primarily due to the iterative cell-write operation, i.e., iterative P&V. Solutions that explicitly address the challenges of working with MLC/TLC NVMs have focused on data compression and data coding by exploiting the physical properties of the NVM cell and the locality in data traffic [29, 36, 46].
Memory compression: Increasing cache capacity by various data compression techniques that leverage different localities have been studied [26–30]. On similar lines, there are solutions that compress main memory traffic for the benefits of bandwidth, power, and capacity [16,31–33,77]. In the context of MLC/TLC NVMs, compressing main memory traffic yields fewer cell-writes per write access, thus lowering energy. Whereas there are solutions that reduce MLC/TLC energy and latency by excluding undesirable states for additional memory area (summarized below), CompEx coding is to the best of our knowledge the first work to explicitly investigate the tradeoffs between the area recovered from compression and solutions that exclude undesirable MLC/TLC states.

Excluding undesirable states: [78,79] propose circuit and architectural changes to dynamically configure MLC PCM cells as either MLC or SLC for latency benefits. On the other hand, [34] excludes programming the hard-to-reset cells that require high programming current and recovers the lost data using ECC, thereby reducing the power consumption and improving lifetime. Extending the observations of [34], Elastic RESET (ER) [9] proposes data coding that eliminates the undesirable terminal RESET state (high programming current) and uses only 2 or 3 of the 4 states of a cell to realize lifetime/power/latency improvements. Similar to ER, hybrid MLC/SLC [35] proposes to opportunistically use PCM cells as either MLC/SLC for energy and latency benefits. Independently, for MLC PCM, [36] proposes energy-efficient data coding to reduce the usage of intermediate high-energy states by mapping the most frequent data patterns to the low energy states. However, [36] requires online computation and storage of the most frequent patterns for every memory line at runtime, incurring compute, memory, and logic overhead. For MLC PCM, [37] proposes data coding that eliminates one of the intermediate resistance states; this improves cell retention but incurs memory overhead. For TLC RRAM, [10] proposes data coding that uses 6 out of 8 TLC RRAM states to improve latency and energy by eliminating the use of intermediate resistance states [10,36]. By combining IDM with dynamic data remapping and error-correcting pointers, lifetime improvements for 20% memory overhead and negligible impact on energy/latency are reported. Recently, [80] observed a super-linear relationship between RESET latency and the number of 1s written to the array, and advocated writing smaller chunks of data using compression to reduce the RESET latency. For MLC PCM,
form-switch (FS) [24] first introduced the notion of writing data in SLC/MLC depending on the result of compression. However, FS may not always result in energy/latency reduction since it depends upon the compression technique used and the energy/latency profile of the NVM cell. In practice, it is necessary to balance dynamic tradeoffs between data compression, the NVM energy/latency profile, and data encoding: CompEx/CompEx++ coding is a step in this direction to realize simultaneous improvements in energy, latency, and lifetime of MLC/TLC NVMs.

3.9 SUMMARY

Our full-system simulations of a system that integrates TLC RRAM show that CompEx/CompEx++ coding reduces total memory energy by 57%/61% and cell latency by 23.5%/26%; these improvements translate to a 5.7%/10.6% improvement in IPC, a 11.8%/19.9% improvement in main memory bandwidth, and 1.8×/1.8× improvement in lifetime over classical binary coding using data-comparison write. CompEx/CompEx++ coding thus addresses the programming energy/latency as well as the lifetime challenges of MLC/TLC NVMs that pose a serious technological roadblock to their adoption in high performance computing systems. In summary, this chapter presented a low overhead solution to realize simultaneous energy, latency, and lifetime improvements in MLC/TLC NVMs.
4.0 CASTLE: COMPRESSION ARCHITECTURE FOR SECURE LOW LATENCY, LOW ENERGY, DURABLE NVM

The core contribution of this chapter is CASTLE, a compression-based architecture that provides a read-decrypt-free, i.e., write-only block-level solution for NVM security for low latency, low energy durable NVMs. CASTLE uses a novel counter design framework and data compression to simultaneously reduce energy and latency and improve IPC, bandwidth, and lifetime. CASTLE extends its energy and latency gains by integrating PDM for no overhead. CASTLE is capable of seamless integration with soft/hard error detection and correction in MLC/TLC NVMs. Thus, CASTLE is a simple practical NVM memory security solution for low latency, low energy, and improved endurance. To the best of our knowledge, this is the first work to systematically explore the integration of data compression and PDM with memory encryption of MLC/TLC NVMs.
4.1 ‘READ-DECRYPT’-FREE WRITE

Figures 20 and 21 illustrate write and read sequences for the state-of-the-art CME scheme DEUCE in comparison to CASTLE. During writes, DEUCE needs to first initiate a read and wait for the read data to be received from the main memory to begin AES OTP generation. In contrast, CASTLE does not require this read operation. As soon as the memory controller issues the write command, the AES OTP generation begins and the write to main memory immediately follows. Figure 20 illustrates this latency gain due to CASTLE’s non-dependence on the existing data in memory.

During reads, CASTLE reads the data from the main memory and proceeds to decrypt the data. However, an important distinction between CASTLE and DEUCE is that CASTLE decrypts the data using only a single OTP. In contrast, DEUCE is a dual counter scheme that uses 2 OTPs: the first OTP is for encryption of the first write in an epoch (32 writes) whereas the second OTP is for encryption of the data chunks on subsequent writes, which differ from the first write. The latency reduction by CASTLE due to single OTP architecture is illustrated in Fig 21. Note that DEUCE [43] argues that it is possible to hide the AES-128 OTP generation latency by reading the counter value first and by using 2 decryption engines—one for each OTP— as a trade-off between logic overhead and latency reduction. In summary, it is clear that ‘read-decrypt’-free nature of CASTLE offers latency/logic-overhead advantage during both write and read operations in comparison to state-of-the-art CME schemes.

CASTLE needs to address two important architectural challenges to be successful in practice: the need for the previous counter value during a write and the increase in energy/latency due to the write-only sequence.

First, in a typical CME architecture, the OTP generation procedure requires the previous counter value, which is stored in main memory along with the data. Therefore, the expensive read-decrypt-modify-write sequence is inevitable with the state-of-the-art CME schemes. CASTLE addresses this challenge by proposing a novel counter design framework, which allows OTP generation without the previous counter value (discussed in Sec. 4.2). Second, state-of-the-art CME schemes achieve energy and latency gains primarily due to the ‘modify-
Figure 20: Illustration of write timing diagrams for CASTLE and DEUCE. CASTLE has lower write latency due to the read-decrypt-free write.

write’ steps in the read-decrypt-modify-write sequence; eliminating the read-decrypt steps, may increase cell activity. CASTLE addresses this energy and latency challenge by leveraging data compression and PDM (discussed in Sec. 4.3).

4.2 CASTLE: COUNTER DESIGN

CME schemes using a one-time-pad (OTP) are the most widely used main memory encryption schemes due to their simplicity in design, low encryption latency, and robustness to statistical attacks. CME schemes use three inputs—a unique counter value for each cache line, the cache line address, and a global secret key—to generate a unique OTP. The OTP is then XORed with the incoming write data (plaintext) to generate the encrypted data (ciphertext). Finally, the memory controller stores the unencrypted counter value (that was used to generate the OTP) and the encrypted data in the NVM main memory.
Figure 21: Illustration of read timing diagrams for CASTLE and DEUCE. CASTLE needs to encrypt/decrypt only 1 OTP at a time as opposed to DEUCE, which requires 2 OTPs.

It is important for a robust encryption scheme to not use the same OTP for two data writes to the same location. Therefore, for every write to a given cache line, state-of-the-art CME schemes first read the previous counter value stored with that cache line, increment the previous counter value to obtain the new counter value, and finally generate the new OTP. Furthermore, whenever a counter overflows, the entire memory needs to be re-encrypted with all the counters reset to an initial value—the counter size is thus chosen to keep the re-encryption overhead within acceptable limits.

However, in order to realize the bandwidth and IPC gains from the read-decrypt-free sequence, CASTLE has no access to the stored counter value for write data encryption. Hence, we propose a counter design framework featuring a global counter (in the memory controller on the processor-side), which is incremented on every write and used to generate the OTP. Consistent with state-of-the-art CME schemes, CASTLE stores the counter value (snapshot of the global counter) that was used for encryption in plaintext along with the ciphertext, i.e., every data line stores a snapshot of the global counter along with it. Note that CASTLE is consistent with state-of-the-art CME schemes and is thus a drop-in replacement for such schemes in practice. However, since the counter is now global, it needs to be bigger in size lest the re-encryption overhead becomes high.
Re-encryption overhead: Consider the re-encryption overhead for a standard DDR3-1333 4-channel memory with 32 GB capacity, 64-bit (8-byte) memory channel, and 512-bit cache line. The time required to rewrite entire 32 GB = 32 GB/(8B×1333 MHz×4 channel) ≈ 0.75s. However, this computation accounts only for the burst-write time and not for the write protocol overhead (efficiency). The DDR3 protocol efficiency can be computed using the Request Access Distance methodology to be about 67% [81]. Furthermore, since re-encryption involves both reading and writing, the computed write overhead should be doubled. Therefore, whenever the global counter overflows, a 2.2s (= 0.75×2/0.67) penalty is incurred by the memory system for re-encryption—this overhead may be higher in a power limited system that limits the number of concurrent writes. The memory system is unusable during such re-encryption times, which may not be tolerable for some applications. Hence, downtime (for a given uptime) is an important factor in determining the size of the global counter.

Computing global counter size: Now, assuming that the system can tolerate a re-encryption overhead of 2.2s every \( T \) hours, the size of the global counter is given by \( \lceil \log_2 (T\times3600s\times 1.333 \text{ GHz} / 8) \rceil \). Table 10 summarizes the global counter size for different uptimes. Note that the above counter design is for the worst case, which assumes that data is written continuously to memory with 100% bus utilization. In reality, the memory bus utilization is not 100% and writes are interspersed with reads, which makes re-encryption less frequent in practice. Furthermore, by using a 48-bit counter, the re-encryption period can be extended to 2 weeks in the full-load worst case. Note that the re-encryption problem is similar to the refresh problem in DRAM (however re-encryption frequency is lower in comparison to refresh frequency). Therefore, techniques like preemptive re-encryption during idle cycles, staggering re-encryption for multiple banks, etc., can mitigate the re-encryption overhead; however, such an evaluation is beyond the scope of this dissertation. Note that the 40-bit counter overhead per cache line is 8 bits more than the 32-bit overhead of DEUCE; hence, CASTLE incurs 1.5% memory overhead over DEUCE for implementing the global counter approach to eliminate the read-decrypt steps of current CME schemes [43].
Table 10: **Global counter sizes for various system uptimes and availabilities.** The key take-away is that the global counter size depends on the system design parameters.

<table>
<thead>
<tr>
<th>Uptime</th>
<th>Availability</th>
<th>Global counter size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 hour</td>
<td>0.9997</td>
<td>40 bits</td>
</tr>
<tr>
<td>1 day</td>
<td>0.99999</td>
<td>44 bits</td>
</tr>
<tr>
<td>1 week</td>
<td>0.999998</td>
<td>47 bits</td>
</tr>
<tr>
<td>2 weeks</td>
<td>0.9999992</td>
<td>48 bits</td>
</tr>
</tbody>
</table>

**Counter cache to eliminate global counter:** Although the global counter architecture enables a completely read-decrypt-free write protocol for CASTLE, it incurs memory overhead to store wider counters along with the data. Furthermore, re-encryption overhead—system downtime—may be unacceptable for some applications. In such cases, a counter cache architecture can be used for storing the counters to eliminate the counter-read bottleneck. On a counter cache hit, the counter can simply be incremented and stored in the counter cache; on a counter cache miss, such a system would need to first read the counter from the memory and then update the new counter value in the counter cache. The use of such a counter cache has been extensively studied in the literature [39, 63]; in our evaluation, the global-counter-based CASTLE can be interpreted as the upper bound (100% counter cache hit scenario) for counter-cache-based CASTLE. Our full-system evaluation of counter-cache-based CASTLE using BΔI on SPEC CPU2006 benchmarks show that counter-cache-BΔI-CASTLE can improve IPC by 14% in comparison to the state-of-the-art DEUCE [43], which is about 90% in comparison to the IPC improvement of BΔI-CASTLE using global counter architecture (refer Sec. 4.6.2).
4.3 CASTLE: LOW LATENCY/ENERGY ENCODING

Although moving away from the read-decrypt-modify-write sequence enables CASTLE to improve IPC and bandwidth, the high energy/latency challenges due to the increased cell activity need to be addressed. This section describes the use of data compression and partial data mapping (PDM) [10], which is an instance of expansion coding, in CASTLE to reduce energy and latency.

CASTLE leverages the compressibility of the program data to realize its energy/latency wins. Program data exhibits regularities (redundancies) due to which we can compress cache lines to a smaller size. However, this regularity is lost when data is encrypted, since encryption scrambles the data to remove all regularity. In the context of MLC/TLC NVMs, compression of the main memory traffic is shown to yield fewer cell updates, thus lowering the energy and latency of writes [16, 35, 82]. Therefore, CASTLE first attempts to compress the incoming data using a cache line compression scheme (frequent-pattern compression (FPC) [46] and base-delta-immediate (B\(\Delta\)I) [47] in this work); if compression is successful, then the compressed data is encrypted by XORing it with the OTP, which is truncated to the size of the compressed data. However, if the data is not compressible, then the entire cache line is encrypted as-is. CASTLE requires 1 tag bit to indicate whether compression was successful/unsuccesful. We show later in this section that this tag bit can be absorbed into the last TLC without incurring any overhead. The success of CASTLE depends on the choice of the compression technique – if the compression technique can compress a large fraction of input data, then CASTLE needs to update fewer cells and needs fewer bus cycles to transmit data to the NVM DIMM in comparison to the original data. Note that CASTLE’s effectiveness depends only on the properties of the incoming write data, i.e., it is independent of the previous data that is existing in the memory location, enabling it to be a read-decrypt-free scheme. Further, once the data is in the NVM DIMM, it is selectively encoded using PDM depending on its compression status (PDM of compressed data is discussed following the illustrative example below).
Example: Without loss of generality and for ease of understanding, consider a 64-bit example shown in Fig. 22 illustrating the 3 data encoding schemes considered in this chapter—DCW [15], DEUCE [43], and CASTLE for TLC RRAM, whose energy and latency numbers are given by Fig. 3.

The data set shown in Fig. 22(a) corresponds to the existing data in memory—plaintext data, OTP-1 (the OTP used to encrypt the existing data in the memory), the ciphertext (the data that is existing in the memory), and the TLC map (conventional binary encoding of 3 bits per TLC). The data set shown in Fig. 22(b) corresponds to the new data—plaintext, the new OTP (OTP-2 $\neq$ OTP-1), and the ciphertext.

Figure 22(c) illustrates the encoding using DCW [15]. Due to data scrambling, which results in high cell activity, all 22 TLCs require cell updates using DCW (updated TLCs are in red). The high cell activity translates to an energy of 337.6 pJ and latency of 150 ns.

In contrast, Fig. 22(d) illustrates DEUCE [43] operating on 16-bit boundaries. DEUCE identifies that only two 16-bit blocks have been modified in the new plaintext data. Therefore, only the TLCs corresponding to the differing blocks are updated (shown in red). Hence, the modify-write step of DEUCE reduces the energy to only 147.1 pJ, while having no impact on latency.

Finally, Fig. 22(e) illustrates data encoding with CASTLE. CASTLE uses FPC to compress the data to half of its original size (in reality, FPC results in a 35-bit compressed word and not 32 bits due to the 3 bit prefix, which is ignored in this example for brevity), which requires updates to only 11 of the 22 TLCs resulting in an energy expense of 160.6 pJ and has no impact on latency. Although this example shows that the energy and latency reduction of CASTLE is comparable to state-of-the-art DEUCE, CASTLE leverages the observation that a large fraction of the TLCs remain unused after compression (see Fig. 22(f)). CASTLE uses PDM (described below) to realize simultaneous energy and latency improvements over DEUCE.

Partial data mapping (PDM) [10]: The iterative program-&-verify (P&V) procedure used for programming MLC/TLC NVMs results in the central MLC/TLC states requiring more energy and latency in comparison to the terminal states. The energy/latency numbers for TLC RRAM shown in Fig. 3 show that the energy and latency to program the central
states can be about 24× and 12× in comparison to the terminal states, respectively. Such large disparity in the energy and latency of states motivates data encoding using only the low energy states, while avoiding the high energy states altogether to reduce energy and latency. PDM allows the use of only the desired MLC/TLC states with no modifications to the write circuitry. A PDM($m,n$) code, which is an instance of expansion coding, encodes data using only $n$ low-energy states in an $m$-ary NVM cell. For example, PDM(8,4), which is equivalent to $(3,2)_8$ expansion coding, operates on an 8-level cell (TLC) NVM by encoding data using only 4 low-energy states out of the total 8 states. Since PDM(8,4) excludes 4 states out of the total 8, we need more TLCs to encode the given data. Furthermore, since $8^2 = 4^3$, 6 logical bits that originally required 2 physical TLCs require 3 TLCs using PDM(8,4). Thus, it is important to note that the integration of PDM with state-of-the-art CME schemes incurs high memory area overheads. For example, integrating BLE [42] and DEUCE [43] with PDM(8,4) incurs 50% memory area overhead. In contrast, CASTLE can easily encode data using PDM(8,4) for no overhead when the data is compressed; when the data is incompressible, CASTLE takes the hit of high cell update activity. However, since a large fraction (60% for FPC and 46% for BΔI) of write traffic is compressible, the overall energy and latency of CASTLE is lower than the state-of-the-art CME schemes like DEUCE.

For the example in Fig. 22(e), the integration of PDM(8,4) to the compressed data leverages the unused TLCs for energy and latency gains. Figure 22(f) illustrates the integration of PDM(8,4), which results in an energy expense of only 66.5 pJ and latency of only 55.7 ns. This translates to a $5 \times (2.7 \times)$ and $2.2 \times (2.7 \times)$ reduction in energy (latency) over DCW and DEUCE, respectively.

**No tag overhead:** Although CASTLE requires a single tag bit in the data field (512 bits) to record the outcome of compression, it is concatenated with the data (513 bits) and absorbed into the last TLC at no cost $\lceil 513/3 \rceil = \lceil 512/3 \rceil = 171$ TLCs. Since $2^n$ ($n > 1$) can never be a multiple of 3, the tag bit can always be encoded within the existing TLCs without any overhead for word sizes that are integer powers of 2. Further, to prevent the tag bit from becoming a latency bottleneck, it is encoded using the lowest latency states of the TLC—‘0’ and ‘7’—when the data is in PDM(8,4) form.
CASTLE and soft/hard error detection and correction: MLC/TLC NVMs are susceptible to both soft and hard errors, which necessitates the use of error detection and correction (EDAC) techniques such as ECC [50], ECP [51], etc. NVMs with EDAC usually use separable coding, i.e., codes such that the data field is stored separate from the EDAC field. Since a write may alter both the data and the EDAC fields, the benefits of encoding the data field using PDM(8,4) may be nullified by high latency writes in the EDAC field. To address this potential shortcoming, we propose that whenever the data field is compressible, the EDAC field be written in PDM(8,4) form—the additional cells required for encoding the EDAC field using PDM(8,4) can be obtained from the residual cells after compression of the data field. From Table 2, we can infer that BΔI-CASTLE leaves 24 TLCs (there are 30 free TLCs, but 6 are used to store the counter value in PDM(8,4) form) unused in the worst case, which can be purposed to provide EDAC in PDM(8,4) to preserve/extend the latency/energy benefits of CASTLE. However, if the data field is incompressible, both the data and EDAC fields are written using classical binary coding.
### Figure 22: Illustration of cell updates, energy, and latency for DCW, DEUCE, and CASTLE for 64-bit input data. Note that the energy and latency numbers are from Fig. 3.
4.4 CASTLE: ARCHITECTURE

The architecture for CASTLE can be logically and physically partitioned into two parts: (i) the compression-decompression engine (CDE), which is a codec in the processor-side memory controller, and (ii) a separate PDM(8,4) codec in the NVM DIMM. The CASTLE codec is embedded in the memory data path as shown in Fig. 23.

Write: During writes, the data is passed through the compression logic, which compares the data with all of the compression patterns to attempt compression. If the data is compressible, then the compressed data is encrypted using the generated OTP (using AES-128 encryption similar to other CME schemes), and sent to the NVM DIMM with the compression tag-bit set (indicating compressed data). The compressed data is further encoded using PDM(8,4) to write data using only the low energy/latency states of the cell.

In contrast, if the data is not compressible, the data is encrypted using the OTP and sent to the NVM DIMM with the compression tag-bit reset (indicating uncompressed data). The PDM(8,4) encoder simply encodes the data using conventional binary encoding and updates the memory array. However, since the width of the encoded data is always less than 64 bytes (shown earlier in this section), CASTLE has zero memory overhead in the data field. Note that the tag bit may need to be updated to record the outcome of CASTLE, regardless of the whether compression is successful or unsuccessful. However, as discussed in Sec. 4.3, this tag bit can be concatenated and absorbed into the data using the last TLC without requiring an extra cell for the tag bit.

Read: On reads, the ciphertext and the stored counter value are read from the NVM DIMM. The PDM(8,4) decoder inspects the tag bit to determine if the ciphertext is in binary-encoded form or in PDM(8,4) form; depending on the tag bit, the data is either decoded or directly passed on to the data bus. To pipeline the read operations efficiently, the counter value is read first, so that the OTP generation is parallel to the cache line read. This is followed by decryption and decompression on the processor-side memory controller—if the data is in compressed form, then the CDE decompresses the data; else, the data is passed on as-is.
Figure 23: CASTLE architecture. The codec lies in the read/write data path adding 2/3 cycles overhead. However, the latency gains due to CASTLE far outweigh this overhead.

Figure 24: Schematic diagram for FPC-CASTLE codec for 64-bit data path. The same codec is replicated depending on the size of the cache line (for example, the codec is replicated 8× for a 512-bit cache line) (a) Encoder: The incoming 64-bit write-data is first encoded using the FPC encoder (latency = 2 cycles) followed by the PDM(8,4) encoder (latency = 1 cycle) to obtain encoded data and the tag bit. (b) Decoder: The tag bit and the encoded data are first decoded for PDM(8,4) (latency = 1 cycle) and then decoded for FPC (latency = 1 cycle). Our implementation hides these latencies using memory access pipelining.
CASTLE codec overhead: We designed and synthesized the 64-bit FPC-CASTLE codec (Fig. 24). The design of B∆I-compressor is assumed from the original B∆I proposal [47]. Note that the FPC-CASTLE codec needs to be replicated depending on the number of 64-bit words in a cache line. For example, the codec is replicated 8× for a 512-bit cache line. The estimated logic hardware overhead for the CASTLE codec tabulated in Fig. 24 is ≈10k gates (< 0.1% per NVM module). Furthermore, although the CASTLE codec has an estimated latency of 3 (2) cycles for encoding (decoding), our implementation uses memory access pipelining to hide this in practice.

4.5 METHODOLOGY

Our evaluation of CASTLE is based on (i) full-system simulations to evaluate the system-level performance and (ii) trace-based simulations for deep, multi-billion instruction evaluation of memory-level energy and latency.

4.5.1 Full-system simulation

CASTLE is evaluated using full-system simulations of a system that integrates TLC RRAM memory using MARSS [12], a full-system multi-core simulator, and DRAMSim2 [13], a cycle-accurate main memory simulator.

MARSS+DRAMSim2: MARSS uses x86 core models from PTLSim [70], a cycle-accurate x86 micro-architecture simulator and plugs it into QEMU [71], a binary-translation system for emulating full systems. QEMU provides the capability of emulating various I/O devices that can be used to boot up entire operating systems without any modification (Linux in this work). MARSS operates in two modes: a fast and simple emulation mode and a detailed simulation mode. In this work, we operate in emulation mode until the region of interest is reached before switching to the detailed simulation mode. Additionally, MARSS tracks only the address component of a memory access, whereas the data component of a memory access is maintained only by QEMU. However, since we the data component of each memory
access, we modify MARSS to propagate data along with the address throughout its memory hierarchy. We use DRAMSim2, a cycle-accurate main memory simulator for simulating the DDR3 NVM main memory system. MARSS and DRAMSim2 are integrated to provide a monolithic, seamless, cycle-accurate simulation of the entire system. Since each access in a TLC NVM memory can potentially have different access latencies, we modify DRAMSim2 to account for memory access latencies.

**MARSS setup:** MARSS was configured to simulate a standard 4-core out-of-order system running at 3GHz. Each core has its own L1 cache with 2 separate instances of 32kB SRAM for data and instructions; the L2 cache is private, with each core having its own instance of 256kB SRAM; finally, the L3 cache is a single, shared, write-back cache of size 8MB. The latencies of each level of cache is tabulated in Table 11.

**DRAMSim2 setup:** For accurate timing simulation, we modified DDR timing parameters along the lines of [3] to substitute DRAM by TLC RRAM NVM with latency parameters extracted and summarized in [10].

**Workloads:** We evaluate CASTLE using the SPEC CPU2006 [48] benchmark suite. These benchmarks reflect a variety of integer and floating-point based workloads used by modern computing systems. To evaluate real-world usage, we use 9 composite workloads with each workload containing 4 benchmarks from the SPEC CPU2006 benchmark suite. These composite workloads are derived from [72], where benchmarks are selectively picked due to

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Attributes</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>4 cores, dual issue out-of-order</td>
<td>—</td>
</tr>
<tr>
<td>L1 (instruction)</td>
<td>32kB, 2 way</td>
<td>2ns</td>
</tr>
<tr>
<td>L1 (data) cache</td>
<td>32kB, 2 way</td>
<td>2ns</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256kB, private, 8 way</td>
<td>5ns</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8MB, shared, 16 way</td>
<td>20ns</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
<td>—</td>
</tr>
<tr>
<td>Main memory</td>
<td>16GB, 8 banks, 1 channel (DRAMSim2), 1GHz</td>
<td>160ns (average)</td>
</tr>
</tbody>
</table>
their memory intensive nature. Table 12 lists the constituent benchmarks for each composite workload and their corresponding writes per kilo-instruction (WPKI) and misses per kilo-instruction (MPKI) as reported by MARSS. Furthermore, in order to study the IPC contribution of each benchmark separately, we tie each benchmark to a core in our simulated system.

4.5.2 Trace-driven simulation

For running deep, multi-billion-instruction simulations, CASTLE is evaluated using NVMain [52]—an architectural-level main memory simulator for emerging NVMs—that estimates the total energy at the memory module level. We modified NVMain to reflect the variable-write-latency behavior of CASTLE and also configured NVMain to simulate an architecture equivalent to that in Table 11. The traces are generated from the SPEC CPU2006 [48] benchmark suite using the Intel Pin binary instrumentation tool [53] on a machine running a 3.3 GHz Intel Core i7 CPU. Our simulation framework captures memory accesses from the processor, recording only those accesses sent to main memory. During trace generation, the benchmarks are first run through $5 \times 10^5$ memory writes, to ignore the write accesses from program initialization; they are then run until $4 \times 10^6$ memory write operations (equivalent to about 4 billion instructions on average) have been recorded or until the program terminates.
Table 12: Composite workloads comprising of 4 benchmarks from SPEC CPU2006 benchmark suite (derived from [72]) for full-system evaluation of CASTLE. The WPKI and MPKI numbers are sensitive to the CPU and memory system architecture; the numbers presented in this table are compiled for the architecture defined in Table 11 using MARSS.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Constituent benchmarks</th>
<th>WPKI</th>
<th>MPKI</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD1</td>
<td>leslie3d, leslie3d, mcf, mcf</td>
<td>4.86</td>
<td>11.96</td>
</tr>
<tr>
<td>WD2</td>
<td>lbm, leslie3d, libquantum, mcf</td>
<td>3.07</td>
<td>8.36</td>
</tr>
<tr>
<td>WD3</td>
<td>lbm, lbm, libquantum, libquantum</td>
<td>3.54</td>
<td>9.29</td>
</tr>
<tr>
<td>WD4</td>
<td>bwaves, leslie3d, omentpp, sphinx3</td>
<td>4.78</td>
<td>11.78</td>
</tr>
<tr>
<td>WD5</td>
<td>GemsFDTD, libquantum, milc, zeusmp</td>
<td>3.73</td>
<td>9.69</td>
</tr>
<tr>
<td>WD6</td>
<td>GemsFDTD, libquantum, milc, milc</td>
<td>2.96</td>
<td>8.12</td>
</tr>
<tr>
<td>WD7</td>
<td>bzip, libquantum, milc, omentpp</td>
<td>2.67</td>
<td>7.53</td>
</tr>
<tr>
<td>WD8</td>
<td>cactusAMD, gcc, gobmk, zeusmp</td>
<td>3.18</td>
<td>8.56</td>
</tr>
<tr>
<td>WD9</td>
<td>astar, gobmk, hmmer, soplex</td>
<td>2.99</td>
<td>8.23</td>
</tr>
</tbody>
</table>
4.6 EVALUATION AND RESULTS

This section presents the memory-level and system-level evaluation of CASTLE. First, we present the memory-level write energy and latency results for different encodings—baseline (binary encoding with DCW), state-of-the-art DEUCE [43] and BΔI+DEUCE+PDM, which is an integration of DEUCE with BΔI and PDM(8,4) encoding 1, compression with encryption (FPC and BΔI), and CASTLE (FPC-CASTLE and BΔI-CASTLE with PDM(8,4) encoding). Second, we present system-level evaluation results for IPC and bandwidth improvements 2.

1Since DEUCE operates at the cache line level, it is natural to integrate BΔI, a cache line compression scheme with DEUCE. Note also that to the best of our knowledge, this is the first work that extends NVM encryption in general (DEUCE in this case) to MLC/TLC NVMs.

2Note that as appropriate and for brevity, we use CASTLE to refer to the average results of FPC-CASTLE and BΔI-CASTLE.
4.6.1 Memory energy/latency

**Summary:** Table 13 summarizes and compares the total module energy, array-level dynamic energy, array-level dynamic latency, and memory overhead for the architectures considered in this chapter. In summary, CASTLE reduces the memory array write energy by 53% and 21%, write latency by 45.7% and 26.4% in comparison to classical binary encoding and state-of-the-art DEUCE, respectively.

**Energy:** Our framework tracks all cell writes from the start of program execution to compute the cumulative energy. Note that the static energy of the peripheral circuits and the memory array are indirectly influenced by the reduction in the latency of each write operation. A lower write latency translates to a lower energy to keep the peripheral circuits active, which is evaluated using NVMain [52]. Fig. 25(a) and (b) show the memory array dynamic energy and the total memory module energy, respectively, for DEUCE, BΔI+DEUCE+PDM, FPC, BΔI, and FPC-CASTLE and BΔI-CASTLE, normalized to binary coding. The last entries of Fig. 25 represent the geometric mean of the energy reduction across all the benchmarks, which is equivalent to simulating all these benchmarks for the same execution time. Note that all the cases integrate DCW [15] to update only the modified cells in the NVM array.

Our simulations show that BΔI+DEUCE+PDM reduces the memory array dynamic (total) energy by 60.7% (19%) and 34.5% (6.6%) over classical binary encoding and DEUCE, respectively. Second, FPC-CASTLE (BΔI-CASTLE) reduces the memory array dynamic
Table 13: Module-level energy, array-level dynamic write energy, array-level write latency, and memory overhead normalized to classical binary coding for the 6 encryption schemes in this chapter. Note that all the cases integrate DCW [15] to update only the modified cells in the NVM array.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical binary coding</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>6.25%</td>
</tr>
<tr>
<td>DEUCE [43]</td>
<td>86.7%</td>
<td>60.0%</td>
<td>88.5%</td>
<td>6.25%</td>
</tr>
<tr>
<td>BΔI+DEUCE+PDM</td>
<td>81%</td>
<td>39.2%</td>
<td>62%</td>
<td>6.25%</td>
</tr>
<tr>
<td>64-bit FPC compression (based on [16])</td>
<td>74.4%</td>
<td>90.1%</td>
<td>97.9%</td>
<td>7.8%</td>
</tr>
<tr>
<td>BΔI compression</td>
<td>82.9%</td>
<td>89.7%</td>
<td>84.1%</td>
<td>7.8%</td>
</tr>
<tr>
<td>FPC-CASTLE</td>
<td>66.5%</td>
<td>55.3%</td>
<td>56.9%</td>
<td>7.8%</td>
</tr>
<tr>
<td>BΔI-CASTLE</td>
<td>74.6%</td>
<td>39.2%</td>
<td>51.6%</td>
<td>7.8%</td>
</tr>
</tbody>
</table>

energy by 44.7% (60.7%) and 7.7% (34.5%) over classical binary coding and DEUCE, respectively. Finally, FPC-CASTLE (BΔI-CASTLE) reduces total energy by 33.4% (25.4%) and 23.2% (14%) over classical binary coding and DEUCE, respectively. Note that BΔI+DEUCE+PDM and BΔI-CASTLE have similar array level energy numbers; however, BΔI-CASTLE reduces total energy by 8% over BΔI+DEUCE+PDM due to the static energy reduction during the read-decrypt stage of writes.

**Latency:** Due to the iterative P&V procedure, the NVM states with lower energy also require lower write latencies (detailed in Sec. 2), which enables CASTLE (using PDM(8,4)) to simultaneously reduce both energy and latency. Our simulations determine the latency for each access by tracking the maximum latency cell write for the access (since the latency for writing a word depends on the cell that requires the longest latency). The overall latency for program execution is then cumulatively computed using the latency of individual accesses. Fig. 26 shows that first, BΔI+DEUCE+PDM reduces the write latency by 38.7% and 29.9% over classical binary coding and DEUCE, respectively. Second, FPC-CASTLE (BΔI-CASTLE) is able to reduce the write latency by 43.1% (48.3%) and 35.6% (41.6%) over classical binary coding and DEUCE, respectively. Note that FPC-CASTLE and BΔI-CASTLE outperform DEUCE and BΔI+DEUCE+PDM due to the additional read-decrypt latency incurred by DEUCE and BΔI+DEUCE+PDM during writes.
4.6.2 Full-system evaluation

![Figure 27: IPC results from full-system simulation (MARSS+DRAMSim2) normalized to classical binary coding. The last bar in each workload represents the arithmetic mean of the IPCs of individual benchmarks. The last set of bars represent the harmonic mean of IPCs for each workload. CASTLE shows an IPC improvement of 60.3% (20.4%) over binary encoding with DCW (DEUCE).]

In this section, we report and discuss the impact of CASTLE on system performance. We use two metrics: (i) instructions-per-cycle (IPC), which is a metric for full-system performance and (ii) main memory bandwidth, which is a metric for main memory performance.

**Summary:** Table 14 summarizes and compares IPC and bandwidth for classical binary encoding (using DCW), read-decrypt-modify-write (binary encoding with cache-line level read-decrypt-modify-write), state-of-the-art DEUCE [43], FPC-CASTLE, BΔI-CASTLE, and counter-cache-BΔI-CASTLE. In summary, CASTLE improves IPC by 60.3% and 20.4% in comparison to classical binary encoding and DEUCE, respectively.

**Instructions-per-cycle (IPC):** To evaluate the impact of CASTLE on IPC, we use a full-system simulator based on MARSS [12] and DRAMSim2 [13]. The simulation setup is described in detail in Section 5.6. We simulate nine composite workloads from Table 12. Fig. 27 shows the IPC for each benchmark in each workload. The last set of bars in each workload in Fig. 27 represents the mean IPC for that workload (since each benchmark is run on a separate and exclusive core, the mean IPC is given by the arithmetic mean of the IPCs of the constituent benchmarks of the workload [74, Ch. 1]). The last set of bars in Fig. 27 represents the harmonic mean of the IPCs of all the workloads (computing the harmonic mean is equivalent to running all the workloads in the same system for a fixed number of instructions).
We make the following observations using Fig. 27: First, our simulations show a good correlation between IPC and both WPKI and MPKI. For example, consider the workloads WD_1 (high MPKI) and WD_6 (low MPKI)—as expected, the high cache miss-rate of WD_1 lowers the IPC in comparison to WD_6 that has a lower cache miss-rate. To understand the dependence of IPC on WPKI, let us consider the workloads WD_1 (high WPKI) and WD_7 (low WPKI). CASTLE^2 reduces latency and improves IPC only during write accesses. Thus the workload with higher WPKI should show a higher improvement in IPC in comparison to the workload with lower WPKI; our simulations, in agreement with the above argument, show that CASTLE improves the IPC by 2× for WD_1 (high WPKI) in comparison to 1.8× for WD_7 (low WPKI). Second, our simulations also show similar correlations at the individual benchmark level. Intuitively, memory-intensive benchmarks like milc (WD_6) or lbm (WD_3) should have lower IPC in comparison to less memory-intensive benchmarks like zeusmp (WD_8) or omnetpp (WD_4); our simulations are in excellent agreement with the above argument. Furthermore, in order to study the impact of the read-decrypt-modify-write sequence on IPC, we simulate cache line level read-decrypt-modify-write encryption architecture.

Our simulations show that the read-decrypt-modify-write architecture can incur up to 50% IPC overhead (WD_2) in comparison to the simple binary encoding architecture. Finally, CASTLE^2 improves IPC by 60.3%, 20.4%, and 14.9% in comparison to classical binary encoding, DEUCE, and BΔI+DEUCE+PDM, respectively. Note that although BΔI-CASTLE and BΔI+DEUCE+PDM have similar dynamic write latency, the read-decrypt-free write procedure of BΔI-CASTLE enables it to improve the IPC by 14.9% over BΔI+DEUCE+PDM.

For systems that cannot tolerate re-encryption overhead of global-counter-based CASTLE, we evaluate BΔI-CASTLE with a 512KB counter cache along [39,63]. Our simulations show that counter-cache-BΔI-CASTLE improves IPC by 52%, 14%, and 4% over classical binary encoding, DEUCE, and BΔI+DEUCE+PDM, respectively; the IPC improvement using counter cache is about 90% in comparison to global-counter-based CASTLE.

**Memory bandwidth:** Fig. 28 shows the main memory bandwidth across nine composite workloads. In Fig. 28, the bars represent FPC-CASTLE and BΔI-CASTLE normalized to
the baseline (binary encoding with DCW). The last set of bars in Fig. 28 represents the geometric mean of the normalized improvements (computing geometric mean is equivalent to running each workload for the same execution time). Intuitively, similar to IPC, workloads that have high WPKI should contribute to higher bandwidth improvements in comparison to those with low WPKI. This can be seen using the example of workload WD$_1$ (high WPKI), which show an improvement of $3.1 \times$ in bandwidth using BΔI-CASTLE in comparison to WD$_7$ (low WPKI), which shows an improvement of only $1.8 \times$. On the whole, our simulations show that CASTLE $^2$ improves the average memory bandwidth by $2.2 \times$, $1.3 \times$, and $1.1 \times$ over classical binary coding, DEUCE, and BΔI+DEUCE+PDM, respectively. Furthermore, counter-cache-BΔI-CASTLE shows bandwidth improvement of $1.9 \times$, $1.1 \times$, and $1.05 \times$ over classical binary coding, DEUCE, and BΔI+DEUCE+PDM, respectively.

**Lifetime:** In this dissertation, we theoretically evaluate the lifetime gains of CASTLE using TLC RRAM as an example. [75] discusses the three primary mechanisms for cell failure in RRAM in detail and each mechanism contributes to limit the lifetime of a cell; at memory array level, lifetime of an RRAM cell is specified as a limit on the number of programming cycles (SET/RESET) it can endure before the cell becomes dysfunctional [75]. Therefore, the lifetime of MLC/TLC RRAM is lower in comparison to SLC RRAM since programming an MLC/TLC RRAM requires higher number of P&V cycles in comparison to SLC RRAM [8]. Since CASTLE effectively reduces the number of P&V cycles for programming an NVM cell (by limiting cell states to only low energy/latency states), it also improves the lifetime of the MLC/TLC NVM. Lifetime evaluation of CASTLE for TLC RRAM—along the lines of [8, 9, 44]—show that CASTLE $^2$ increases the lifetime by $1.8 \times$ over DEUCE.
Table 14: IPC and bandwidth for read-decrypt-modify-write, DEUCE [43], BΔI+DEUCE+PDM, FPC-CASTLE and BΔI-CASTLE, and counter-cache-BΔI-CASTLE (norm. to binary encoding). Note that all the cases integrate DCW [15] on the NVM DIMM to update only the modified cells in the array.

<table>
<thead>
<tr>
<th>Method</th>
<th>IPC</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical binary coding</td>
<td>100%</td>
<td>1×</td>
</tr>
<tr>
<td>read-decrypt-modify-write</td>
<td>92.8%</td>
<td>1.1×</td>
</tr>
<tr>
<td>DEUCE [43]</td>
<td>133%</td>
<td>1.7×</td>
</tr>
<tr>
<td>BΔI+DEUCE+PDM</td>
<td>146%</td>
<td>1.8×</td>
</tr>
<tr>
<td>FPC-CASTLE</td>
<td>168%</td>
<td>2.0×</td>
</tr>
<tr>
<td>BΔI-CASTLE</td>
<td>169%</td>
<td>2.3×</td>
</tr>
<tr>
<td>Counter-cache-BΔI-CASTLE</td>
<td>152%</td>
<td>1.9×</td>
</tr>
</tbody>
</table>

Figure 28: Main memory bandwidth obtained using full-system simulation (MARSS+DRAMSim2), normalized to classical binary coding. CASTLE improves bandwidth by 1.2× (2.2×) over classical binary coding (DEUCE).
4.7 RELATED WORK AND CASTLE

This section discusses prior art on main memory security and clarifies CASTLE’s contributions in relation to them.

**Memory encryption:** Historically, [83] proposed direct memory encryption and authentication using shared keys for protection of the stored data to combat software piracy. However, such a direct encryption scheme is vulnerable to statistical attacks, since data and codewords have a one-to-one mapping. To combat this statistical weakness, [38] proposes data scrambling using a one-time pad (OTP), which is a random vector used only once in an encrypted session. Although OTP is a robust solution, it incurs high encryption latency. To reduce OTP generation latency, [39] proposes to use a small sequence number cache. Similarly, [63] proposes counter-overhead reduction by using split counter mode encryption using a 64-bit major counter for a 4 KB page and a 7-bit minor counter for each line. In addition, [40] proposes to reduce OTP generation latency by modifying the translation look-aside buffer and the page table. Recently, i-NVMM [62] discusses the problems of data persistence in non-volatile main memories (NVMM) and proposes latency improvements by distinguishing between hot and cold data stored in NVMM—the hot data is kept unencrypted until just before power down. Although i-NVMM protects the memory from the stolen-DIMM attack, it is still vulnerable to bus-snooping attacks. In contrast, CASTLE protects the memory from both stolen-DIMM and bus-snooping attacks.

**Bit-write reduction:** Since the encrypted data is effectively uniformly random, this results in increased bit writes, which translates to high energy, high latency, and low lifetime. To mitigate the bit writes, [42] proposes 128-bit (AES granularity) block-level encryption. On a write, only the 128-bit chunks that are modified in comparison to the existing data need to be updated. DEUCE [43] further reduces this re-encryption granularity to just 16 bits. However, in all these techniques, the write operation requires a mandatory read-decrypt sequence to precede the write for reducing bit writes. Furthermore, these techniques are less effective in the case of MLC/TLC NVMs, where bit-write reduction is not sufficient to reduce energy and latency. Effective energy/latency reduction requires the use of PDM, which cannot be
integrated with existing solutions without incurring prohibitively high memory overheads. In contrast, CASTLE proposes a read-decrypt-free scheme that brings together the advantages of a write-only encryption protocol and the low energy/latency of data compression and PDM.

4.8 SUMMARY

Our full-system simulations of a TLC RRAM architecture showed that CASTLE-based encryption reduced memory energy by 19% and write latency by 38.7%; these improvements translate to a 1.2× improvement in IPC, a 2.2× improvement in memory bandwidth, and a 1.8× improvement in memory lifetime over comparable state-of-the-art solutions for NVM security. In summary, this chapter presented a novel low-overhead compression-based read-decrypt-free security architecture for low latency/energy, high performance, and high endurance in MLC/TLC NVMs.
5.0 RAPID: READ ACCELERATION FOR IMPROVED PERFORMANCE AND DURABILITY IN MLC/TLC NVMS

The core contribution of this chapter is RAPID, a no-overhead critical-word-first Read Acceleration architecture for Improved Performance and Durability in MLC/TLC NVMs. At its core, RAPID encodes the critical words (CW) in a cache line using only the MSbs of MLC/TLC NVM cells. As opposed to read-time CW optimization, RAPID leverages the recent progress in write-time CW optimization to accelerate a large fraction of the read accesses to MLC/TLC NVM. To the best of our knowledge, RAPID is the first work to demonstrate a zero-cost solution to improve MLC/TLC NVM read latency. Furthermore, RAPID’s ability to easily integrate with error correction support (ECC/ECP) and memory encryption makes it a simple practical solution to the read latency challenges of MLC/TLC NVMs.
5.1 RAPID: WRITE-TIME CW ENCODING

In this sub-section, we focus on data encoding details of RAPID and on the no-overhead nature of the encoding. Without loss of generality and for ease of understanding, consider an example with word size of 4 bits and a 32-bit cache line using TLC NVM, as shown in Fig. 29. We illustrate 3 encodings in this example—classical binary encoding, state-of-the-art SPCM [2], and RAPID. Note that DBM requires significant OS support in terms of page mapping and migration for its benefits; however, the focus of this dissertation is to provide a solution that is a drop-in replacement for the existing memory architectures. Therefore, we do not discuss DBM for the rest of this dissertation.

![Illustration of data encoding for (a) classical binary encoding, (b) SPCM [2], and (c) RAPID using a 4-bit word 32-bit cache line. The key take away is that unlike SPCM that potentially incurs as much as $3 \times \text{write overhead}$ over classical binary encoding, RAPID has no write overhead.](image)

**Classical binary encoding:** Figure 29(a) illustrates classical binary encoding, where 3 continuous logical bits are packed into every TLC. Classical binary encoding requires updates to $11 (= \lfloor 32/3 \rfloor)$ TLCs for every write in this example. However, it is important to note that a full word is available to be read out only after all the bits encoded in a TLC are decoded using three read strobes.
SPCM [2]: Figure 29(b) illustrates read latency reduction with SPCM. In order to reduce read latency, SPCM writes data from three separate cache lines in a single set of 32 TLCs. Hence, every cache line has 2 partner lines, which are encoded and stored into the same set of 32 TLCs. Although the two cache lines encoded using MSbs and mSbs exhibit reduced access latency in comparison to the classical binary encoding, they come at the cost of increased cell updates. The example shows that SPCM requires updates to 32 TLCs during a write in comparison to only 11 TLCs in classical binary encoding. This cell update overhead translates to $3 \times$ energy and endurance overheads in practice. Furthermore, NVM modules often have strict power budgets, which restrict the number of cells that can be updated at a time. Therefore, in power-limited NVMs, the $3 \times$ write overhead can translate to a $3 \times$ increase in write latency. In addition, the limitations of SPCM and classical binary encoding are aggravated in the presence of denser NVM technologies that can store more than 3 logical bits in a cell.

RAPID: RAPID leverages write-time CW optimization to encode the CW in a cache line, which is a relatively new area of research [49, 84]. In [49], it was observed that the CW in a cache line is highly regular and hence can be predicted with a high degree of confidence at write time. It was reported that word 0 is the CW in a cache line for 67% of the entire read traffic to main memory. This is in agreement with our trace-based evaluation of SPEC CPU2006 benchmarks, shown in Fig. 30, where 70-90% of the read traffic can be accelerated by expediting the service of two words 0 and 1. Furthermore, we observe that the predictability of CWs is much higher for composite workloads using full-system simulations (not reported here for brevity). Such a regularity in the CWs enables write-time encoding of CWs, so that they can be accessed quickly during reads. It is important to note that RAPID is designed to always identify word 0 and word 1 as CWs; this obviates the need for any book-keeping with respect to CW identification.

Figure 29(c) illustrates data encoding using RAPID. First, RAPID predicts the CWs in a cache line (words 0 and 1 in our evaluation) by leveraging critical-word regularity using the offline CW optimization scheme proposed in [49]. Second, the predicted CWs are encoded using the MSbs (circled in red). The mSbs are used to encode the next most CWs in the cache line (circled in purple) and the LSbs are used to encode the remainder of the cache.
Figure 30: The percentage distribution of the indices of the 8 64-bit CWs in a 512-bit cache line for traces generated using SPEC CPU2006 benchmarks. The key take-away is that ‘0’th word in a cache line is the CW for about 51% of the traffic. Furthermore, the top 2 CWs in a cache line account for 70-90% of the traffic.

line (circled in blue). Therefore, during a read, the predicted CWs—encoded using MSbs—are decoded immediately after one read strobe, without having to decode all the three bits stored in the NVM cell. Note that RAPID uses offline CW optimization scheme and therefore requires no book-keeping in either memory module or the memory controller. Furthermore, RAPID requires updates to only 11 TLCs on a cache line write, which requires 0% latency, energy, and endurance overhead over classical binary encoding.

Both SPCM and RAPID rearrange data in MLC/TLC NVM to accelerate the reads of a portion of the data. The key difference is that whereas SPCM rearranges data using three cache lines at a time, RAPID does so at the granularity of a single cache line. As a result RAPID has key advantages over SPCM: (i) it requires no write overhead and (ii) it leverages write-time CW optimization. Therefore, RAPID effectively leverages the low read latency advantage of MSbs in MLC/TLC NVM to improve read performance without impacting write latency and performance.
5.2 RAPID: READ TIMING AND ACCELERATION

Figure 31: Read timing diagram (DDR3) for (a) classical binary encoding, (b) SPCM [2], and (c) RAPID. Since the read latencies of different words in a cache line are pre-determined (during the write time), RAPID memory controller places the column address (along with the CAS signal) at the appropriate time to enable read acceleration. The key take away is that RAPID can accelerate a greater number of read accesses in comparison to both state-of-the-art SPCM and classical binary encoding for no write overhead.

In this section, we discuss the timing details of read accesses for classical binary encoding, SPCM, and RAPID, using the DDR3 timing protocol. The temporal sequence of events during a read access for classical binary encoding, SPCM, and RAPID is detailed using Fig. 31. In classical binary encoding, each data word is encoded using all the bits of an MLC/TLC NVM cell; therefore, none of the data words are ready to be served until all the bits are decoded using iterative read procedure (detailed in Section 2.3.1). Hence, classical binary encoding can begin data read-out only after the completion of strobe-3, as illustrated.

Since NVM DIMM timing protocols are still being standardized, there is a lack of simulation tools offering native support for NVM DDR. Therefore, this work uses the DDR3 protocol as an example to illustrate and evaluate the innovations proposed in this dissertation. However, it should be noted that since RAPID does not rely on the DDR3 read protocol for its benefits, the proposed innovations are agnostic to read protocols customized for NVM technologies.
in Fig. 31(a). In contrast, SPCM encodes cache lines using only one of the three bits in a cell—either MSb, mSb, or LSb. Therefore, a read access can be of three different speed grades depending on the significance of the bits that are used for storing a cache line, as illustrated in Fig 31(b). Thus, on average, SPCM serves a third of all the requests after strobe-1, a third after strobe-2, and the remaining third after strobe-3. Finally, Fig. 31(c) illustrates the read timing diagram for RAPID. RAPID encodes word 0 and word 1, believing them to be CWs, using MSbs; the rest of the words in the cache line are encoded using mSbs and LSbs. The read access using RAPID is accelerated, i.e., served on strobe-1, if the predicted CW and the actual CW match, else, the CW is served as and when it is decoded. RAPID and state-of-the-art SPCM [2] require equivalent modifications to the memory circuitry. First, both RAPID and SPCM require three separate I/O gating circuits to support three different read-outs (no additional sense amplifiers are required). Second, both RAPID and SPCM incur routing overheads to route MSbs/mSbs/LSbs from the word lines to the row buffer. Finally, in DDR3 read protocol, both RAPID and SPCM require column addresses (and CAS signals) to be provided by the memory controller, and the correct temporal placement of these signals requires the memory controller to know whether a given read access can be accelerated. The RAPID memory controller can easily decode this information by checking the word address of the read access, since the CWs are always pre-determined to be word 0 and word 1. In summary, the primary advantage of RAPID is that it can accelerate a major fraction of reads due to the regularity in the CWs, without incurring any latency/energy/endurance overhead.

**Summary:** Table 15 summarizes the normalized read latency of classical binary encoding, SPCM [2], and RAPID along with their overheads. The overheads considered are cell-update overhead, write latency, write energy, and write endurance of TLC NVM. Note that read latency improvements provided in this table are only estimates; a detailed evaluation using SPEC CPU2006 benchmarks on MARSS [12], DRAMSim2 [13], and NVMain [52] is provided in Sec. 5.7.
Table 15: Estimated read latency, cell-update overhead, write latency, write energy, and endurance normalized to classical binary encoding.

<table>
<thead>
<tr>
<th></th>
<th>Read Latency</th>
<th>Cell-update Overhead</th>
<th>Write Latency</th>
<th>Write Energy</th>
<th>Endurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary encoding</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
</tr>
<tr>
<td>SPCM [2]</td>
<td>≈0.5×</td>
<td>3×</td>
<td>3×</td>
<td>3×</td>
<td>0.33×</td>
</tr>
<tr>
<td>RAPID</td>
<td>≈0.43×</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
<td>1×</td>
</tr>
</tbody>
</table>
5.3 RAPID: DATA ORGANIZATION

This section describes the data organization in RAPID. Typically NVM main memory is organized as multiple ranks, where each rank is further sub-divided into banks, arrays, sub-arrays, and rows in decreasing order of size [3]. A typical NVM row accommodates 32/64 512-bit cache lines, where each cache line is stored using $\lceil 512/3 \rceil = 171$ TLCs. These 171 TLCs offer 171 MSbs, mSbs, and LSbs each for organizing 8 64-bit words. For ease of understanding, we illustrate RAPID data organization using an 8 cell 3-bit word 24-bit cache line example, as illustrated in Fig. 32. Figure 32 represents the MSb, mSb, and LSb of each TLC using red (M), amber (m), and green (L) colors, respectively. Note that only two of the eight words ($W_0$ and $W_1$) can be encoded using MSbs alone, i.e., only two of the eight words are ready to be read out after read strobe-1. However, three additional words can be read out after each of the second and third strobes. Extending this example to a 64-bit word 512-bit cache line that requires $\lceil 512/3 \rceil = 171$ TLCs, 2, 3, and 3 full 64-bit words can be read out after strobes 1, 2, and 3, respectively. Therefore, RAPID data organization can be used to accelerate the reads for 2 CWs. Note that since RAPID relies only on intra-cache-line data reorganization, it does not require any additional buffers or circuitry in the memory module for its operation.
Figure 32: Illustration of data organization of an example cache line in a TLC NVM memory array using RAPID.
5.4 RAPID: NVM RELIABILITY

MLC/TLC NVMs are susceptible to both soft and hard errors, which necessitates the use of error detection and correction (EDAC) techniques such as ECC [50] and ECP [51], respectively, for NVM reliability. This section describes the integration of RAPID with ECC/ECP for mitigating soft/hard errors in MLC/TLC NVMs.

**RAPID-ECC:** ECC protects NVMs against soft errors (primarily due to resistance drift), which manifest as independent bit errors, using extra parity bits (EDAC field). Since read data can be forwarded to the memory controller only after the data is checked and corrected for errors, the processing of the EDAC fields can potentially nullify the benefits of RAPID. In order to address this potential shortcoming, we propose that the EDAC field for word-level ECC (like SEC-DED, DEC-DED, etc.) be stored using MSbs alongside the CWs W₀ and W₁ as follows. Without loss of generality and for ease of understanding, consider a 64-bit word, 512-bit cache line with a 64-bit EDAC field. The data and the EDAC fields together constitute 576 bits, which require 192 TLCs for storage. These 192 TLCs offer 192 MSbs that can be used for low latency data encoding. We propose that out of the 192 MSbs, 128 MSbs be used for storing of CWs W₀ and W₁, and the remaining 64 MSbs be used to store the EDAC field. Thus, CWs and their corresponding EDAC bits are accelerated together, which allows complete word-level-ECC decoding without having to wait for the reads of non-CWs. Thus, RAPID preserves/extends its read acceleration benefits in the presence of word-level ECC. Therefore, RAPID can be successfully integrated with existing EDAC techniques with full latency/energy/endurance benefits.

**RAPID-ECP:** This section describes RAPID-ECP—integration of ECP [51] with RAPID—to handle NVM hard errors. We show that RAPID-ECP has equivalent (superior) error correction capability in comparison to classical binary encoding (state-of-the-art SPCM [2]).

ECPs provide alternate locations to store the data from a failed unprogrammable cell. ECPs comprise of (i) a pointer to the failed cell and (ii) an alternate cell to store the data from the failed cell. For SLC NVMs with 512-bit data cache lines and 60-bit ECP fields, each ECP requires 10 bits—9 bits for the pointer and 1 bit for storing the correct value.
Thus, a 60-bit ECP field can store 6 ECPs, which can correct up to 6 hard errors. However, in the case of MLC/TLC NVMs, the size of the pointers and their error correction capability depends on the data organization in memory.

Without loss of generality, for ECPs in TLC NVMs, classical binary coding stores each cache line using all 3 bits of a TLC, requiring 171 TLCs for a 512-bit logical cache line. Thus, for storing 6 ECPs of 11 bits each (\(\lceil \log_2(171) \rceil = 8 \) bits for the pointer and 3 bits for the replacement TLC), classical binary encoding requires 66 bits (22 TLCs); this corresponds to a hard error correction capability of 18 hard errors per 512-TLC physical cache line.

However, the unique data organization of state-of-the-art SPCM [2] allows only 6 ECPs for 3 logical cache lines in comparison to 18 ECPs for 3 logical cache lines in classical binary coding for the following reasons. SPCM stores a complete logical cache line and its corresponding ECPs using either MSb/mSb/LSb exclusively. Thus, a hard error in a physical cache line manifests as a single logical bit error in all the three logical cache lines that are stored in a physical cache line. Further, the logical bit errors in all the three logical cache lines occur at exactly the same location, since the \(i\) th TLC stores the \(i\) th bits of the three logical cache lines. Therefore, the ECPs corresponding to the error bits for all the three cache lines have the same pointer values, and differ only in the replacement data bits. Thus, SPCM can correct only 6 hard errors in a 512-TLC physical cache line, as opposed to 18 hard errors in classical binary coding.

In contrast, by avoiding the pointer replication inherent to SPCM, RAPID-ECP ensures error correction capability of 18 hard errors per 512-TLC physical cache line, equivalent to binary encoding. Since RAPID uses all 3 bits of a TLC to store data bits from a single logical cache line, a hard error is confined to only one logical cache line in RAPID data organization. Due to this, RAPID-ECP separates the ECP storage for each cache line both logically and physically, restoring the error correction capability to 18 hard errors per 512-TLC physical cache line. Note that similar to RAPID-ECC, all the ECP bits in RAPID-ECP can be stored exclusively using fast MSbs, which allows processing of CWs without having to wait for the non-CWs in a cache line.
5.5 RAPID ENCRYPTION

Although NVM non-volatility is desirable, it exposes main memory to new types of security vulnerabilities. Several works propose encryption as a robust solution to counter security vulnerabilities [42,43,62]. State-of-the-art counter mode encryption schemes use a one-time pad (OTP) for data encryption; OTP generation uses a cache-line-level counter (typically 32 bits) to improve its resilience towards statistical attacks. The counter value that is used for OTP generation is stored in plaintext along with the corresponding encrypted cache line. During a read access, the counter value is first read from the memory to regenerate the same OTP that was used to encrypt the data in the first place; the regenerated OTP is XORed with the encrypted read data to obtain the plaintext data. Since OTP generation (during decryption) is a potential bottleneck in encrypted memory systems, it is important to speed up the read of the counter value to enable faster OTP generation. In order to address this potential shortcoming, we propose that the counter be stored using the MSbs along with the CWs $W_0$ and $W_1$ and the EDAC bits.

Without loss of generality and for ease of understanding, consider a 64-bit word 512-bit cache line with a 32-bit encryption counter and a 64-bit EDAC field. The total size of the cache line is 608 bits, which requires 203 TLCs for storage. These 203 TLCs offer 203 MSbs that can be used for low latency data encoding. We propose that out of the 203 MSbs, 32 MSbs be used for storing the counter, 128 MSbs be used for storing the two CWs $W_0$ and $W_1$, and the remaining 43 MSbs be used to store a part of the EDAC field. Since only 43 bits of the 64-bit EDAC field can be stored using MSbs, we propose that the EDAC bits corresponding to $W_0$ and $W_1$ be prioritized over those corresponding to other words in the cache line. Thus, RAPID can be integrated into state-of-the-art encryption architectures to successfully preserve its read acceleration advantages.
5.6 METHODOLOGY

Our evaluation of RAPID is based on (i) full-system simulations to evaluate the system-level performance and (ii) trace-based simulations for deep, multi-billion instruction evaluation of module-level energy and latency of RAPID.

5.6.1 Full-system simulation

RAPID is evaluated using full-system simulations of a system that integrates TLC RRAM memory using MARSS [12], a full-system multi-core simulator, and DRAMSim2 [13], a cycle-accurate main memory simulator.

MARSS uses x86 core models from PTLSim [70], a cycle-accurate x86 micro-architecture simulator and plugs it into QEMU [71], a binary-translation system for emulating full systems. QEMU provides the capability of emulating various I/O devices (HDD, ethernet, HID, etc.) that can be used to bootup entire operating systems without any modification (Linux in this work). MARSS operates in two modes: a fast and simple emulation mode or a detailed simulation mode using simulator commands. In this work, we operate in emulation mode until the region of interest is reached before switching to the detailed simulation mode. Additionally, MARSS tracks only the address component of a memory access, whereas the data component of a memory access is maintained only by QEMU. However, since we are interested in the actual data component of a memory access, we modify MARSS to propagate data along with the address throughout its memory hierarchy.

We use DRAMSim2, a cycle-accurate main memory simulator for simulating the DDR3 NVM main memory system. MARSS and DRAMSim2 are integrated to provide a monolithic, seamless, cycle-accurate simulation of the entire system. Since each access in a TLC NVM memory can potentially have different access latencies, we modify DRAMSim2 to account for memory access latencies.
**MARSS setup:** MARSS was configured to simulate a standard 4-core out-of-order system running at 3GHz. Each core has its own L1 cache with 2 separate instances of 32kB SRAM for data and instructions; the L2 cache is private, with each core having its own instance of 256kB SRAM; finally, the L3 cache is a single, shared, write-back cache of size 8MB. The latencies of each level of cache is in Tab. 16.

Table 16: **Configuration of the evaluation architecture.**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Attributes</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>4 cores, dual issue out-of-order</td>
<td>—</td>
</tr>
<tr>
<td>L1 (instruction) cache</td>
<td>32kB, 2 way</td>
<td>2ns</td>
</tr>
<tr>
<td>L1 (data) cache</td>
<td>32kB, 2 way</td>
<td>2ns</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256kB, private, 8 way</td>
<td>5ns</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8MB, shared, 16 way</td>
<td>20ns</td>
</tr>
<tr>
<td>Cache line size</td>
<td>64 bytes</td>
<td>—</td>
</tr>
<tr>
<td>Row buffer size</td>
<td>64 bytes</td>
<td>—</td>
</tr>
<tr>
<td>Main memory</td>
<td>16GB, 8 banks, 1 channel (DRAMSim2), 1GHz</td>
<td>160 ns (W) (average)</td>
</tr>
</tbody>
</table>

**DRAMSim2 setup:** For accurate timing simulation, we modified DDR timing parameters along the lines of [3] to substitute DRAM by TLC RRAM with latency parameters extracted and summarized in [10,85].

**Workloads:** We evaluate RAPID using SPEC CPU2006 [48] benchmark suite. These benchmarks reflect a variety of integer and floating-point based workloads used by modern computing systems. To evaluate real-world usage, we use 9 composite workloads with each workload containing 4 benchmarks from the SPEC CPU2006 benchmark suite. These composite workloads are derived from [72], where benchmarks are selectively picked due to their memory intensive nature. Table 17 lists the constituent benchmarks for each composite workload and their corresponding reads per kilo-instruction (RPKI) and misses per kilo-instruction (MPKI) as reported by MARSS. Furthermore, in order to study the IPC contribution of each benchmark separately, we tie each benchmark to a core in our simulated system.
Table 17: Composite workloads comprising of 4 benchmarks from SPEC CPU2006 benchmark suite (derived from [72]) for full-system evaluation of RAPID. The RPKI and MPKI numbers are sensitive to the CPU and memory system architecture; the numbers presented in this table are compiled for the architecture defined in Table 16 using MARSS.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Constituent benchmarks</th>
<th>RPKI</th>
<th>MPKI</th>
</tr>
</thead>
<tbody>
<tr>
<td>WD₁</td>
<td>leslie3d, leslie3d, mcf, mcf</td>
<td>6.78</td>
<td>11.62</td>
</tr>
<tr>
<td>WD₂</td>
<td>lbm, leslie3d, libquantum, mcf</td>
<td>5.87</td>
<td>9.48</td>
</tr>
<tr>
<td>WD₃</td>
<td>lbm, lbm, libquantum, libquantum</td>
<td>5.88</td>
<td>9.55</td>
</tr>
<tr>
<td>WD₄</td>
<td>bwaves, leslie3d, omentpp, sphinx3</td>
<td>7.63</td>
<td>12.97</td>
</tr>
<tr>
<td>WD₅</td>
<td>GemsFDTD, libquantum, milc, zeusmp</td>
<td>7.05</td>
<td>11.9</td>
</tr>
<tr>
<td>WD₆</td>
<td>GemsFDTD, libquantum, milc, milc</td>
<td>6.36</td>
<td>10.44</td>
</tr>
<tr>
<td>WD₇</td>
<td>bzip, libquantum, milc, omentpp</td>
<td>5.82</td>
<td>9.36</td>
</tr>
<tr>
<td>WD₈</td>
<td>cactusAMD, gcc, gobmk, zeusmp</td>
<td>5.01</td>
<td>7.85</td>
</tr>
<tr>
<td>WD₉</td>
<td>astar, gobmk, hmmer, soplex</td>
<td>5.84</td>
<td>5.84</td>
</tr>
</tbody>
</table>

5.6.2 Trace-driven simulation

For running deep, multi-billion-instruction simulations, RAPID is also evaluated using NVMain [52]—an architectural-level main memory simulator for emerging NVMs—that estimates the total energy at the memory module level. We modified NVMain to reflect the variable-read-latency behavior of RAPID and also configured NVMain to simulate an architecture equivalent to that in Table 16. The traces are generated from the SPEC CPU2006 [48] benchmark suite using the Intel Pin binary instrumentation tool [53] on a machine running a 3.3 GHz Intel Core i7 CPU. Our simulation framework captures memory accesses from the processor, recording only those accesses sent to main memory. During trace generation, the benchmarks are first run through $5 \times 10^5$ memory accesses, to ignore the accesses from program initialization; they are then run until $10^7$ memory operations (equivalent to about 4 billion instructions on average) have been recorded, or program termination, if it occurs earlier.
5.7 EVALUATION AND RESULTS

Figure 33: Total memory module energy results (using NVMain [52]) for PL-SPCM, SPCM [2], RAPID, and ideal-RAPID, normalized to classical binary encoding. RAPID reduces the total energy by 41% and 24% in comparison to classical binary encoding and state-of-the-art SPCM, respectively.

Figure 34: (a) Module-level latency and (b) array-level read latency results for PL-SPCM, SPCM [2], RAPID, and ideal-RAPID, normalized to classical binary encoding. RAPID reduces average module-level latency by 21% and 11% and array-level latency by 48% and 22% in comparison to classical binary encoding and state-of-the-art SPCM, respectively.

This section presents the evaluation of RAPID at the memory and system levels. First, in Sec 5.7.1, we present the energy and latency results at the memory level for with different architectures—(i) baseline (classical binary encoding with DCW), (ii) power-limited SPCM (PL-SPCM), which accounts for the write latency overheads of SPCM in a power-limited NVM module, (iii) state-of-the-art SPCM [2], which assumes that there is no write latency...
Figure 35: IPC results from full-system simulation (MARSS+DRAMSim2). The last bar in each workload represents the arithmetic mean of the IPCs of individual benchmarks. The last set of bars represent the harmonic mean of IPCs for each workload. RAPID shows an IPC improvement of 15.2% and 7.5% over classical binary encoding and SPCM, respectively.

Second, in Sec 5.7.2, we present the results for system-level evaluation of RAPID; primarily to evaluate the impact of latency improvements of RAPID on system performance.

---

2 Ideal-RAPID illustrates a scenario where every read access requests the CW first and thus the reads are always accelerated. This is an upper bound on the read acceleration of RAPID.
5.7.1 Memory energy/latency

**Summary:** Table 18 summarizes and compares the total module energy, module-level latency, array-level latency, and write overhead for the five architectures considered in this chapter. In summary, RAPID reduces the total module energy by 41% and 24%, module-level latency by 21% and 11%, and array read latency by 48% and 22% in comparison to classical binary encoding and state-of-the-art SPCM, respectively.

**Energy:** Our simulation framework tracks all the cell accesses that occur from the beginning of program execution to compute the cumulative energy required. The static energy from peripheral circuits and the memory array are indirectly influenced by the improvement in the latency of each read operation. A lower read latency translates to lower energy to keep the peripheral circuits active, which is evaluated using NVMain [52]. Fig. 33 shows the total memory module energy for classical binary encoding, PL-SPCM, SPCM [2], RAPID, and ideal-RAPID, normalized to classical binary encoding. The last entries of Fig. 33 represent the geometric mean of the energy improvement across all the benchmarks, which is equivalent to simulating all these benchmarks for the same execution time. Simulations show that RAPID reduces the total module energy by 41%, 49%, and 24% over classical binary encoding, PL-SPCM, and SPCM, respectively. Note that ideal-RAPID (bounding case) reduces the total energy by 5% over RAPID.

**Read latency:** We evaluate the impact on module-level latency and array-level read latency due to the five encodings evaluated in this chapter. As discussed in Section 5.2, the read latency for the encodings based on SPCM and RAPID vary depending on the accessed address. For PL-SPCM and SPCM, depending on the cache-line address (cache-line address modulo 3), the read response time is 15ns, 30ns, or 45ns; the read latency per strobe is about 10-15 ns [85]. In addition the write delay for PL-SPCM is configured to be 3× the normal write latency for TLC RRAM [10] to account for the 3× write overhead. In contrast, the read latency for RAPID is dependent on the CW at the read-time. If the write-time predicted CWs —word 0 and word 1 in our evaluations—match with the read-time CW, then the reads are accelerated, else, the read latency of the access depends on whether the CW is stored using mSbs or LSbs. However, for ideal-RAPID, every read access is configured
to be serviced using low read latency (MSb latency). It is important to note that although the read latency numbers can vary widely across different NVM technologies and memory array designs [68], RAPID can accelerate the reads as long as the read procedure follows the iterative sensing technique (described in Sec. 2.3.1). The latency for each read is then cumulatively computed to obtain the overall latency for program execution. Fig. 34 shows that RAPID is able to reduce the module-level latency (array-level read latency) by 21% (48%), 50% (22%), and 11% (22%) in comparison to classical binary encoding, PL-SPCM, and SPCM, respectively. Furthermore, ideal-RAPID reduces the module-level (array-level) latency by 3% (24%) over RAPID.
Table 18: Total energy, module latency, array latency, and write latency normalized to classical binary encoding. Note that all the cases use DCW [15] to update only the modified cells in the NVM array.

<table>
<thead>
<tr>
<th></th>
<th>Total Energy</th>
<th>Module Latency</th>
<th>Array Latency</th>
<th>Write Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical binary encoding</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>1×</td>
</tr>
<tr>
<td>PL-SPCM</td>
<td>144%</td>
<td>158%</td>
<td>74%</td>
<td>3×</td>
</tr>
<tr>
<td>SPCM [2]</td>
<td>78.2%</td>
<td>88.4%</td>
<td>74%</td>
<td>1×</td>
</tr>
<tr>
<td>RAPID</td>
<td>59.3%</td>
<td>79.2%</td>
<td>43.7%</td>
<td>1×</td>
</tr>
<tr>
<td>ideal-RAPID</td>
<td>56.5%</td>
<td>77.1%</td>
<td>33%</td>
<td>1×</td>
</tr>
</tbody>
</table>
5.7.2 Full-system evaluation

In this section, we report and discuss the impact of RAPID on system performance. We use two metrics: (i) instructions-per-cycle (IPC), which is a metric for the full-system performance, and (ii) main memory bandwidth, which is a metric for main memory performance. Additionally, we also compare the memory read latency distribution statistics to validate the intuition developed in Sec. 5.2.

Summary: Table 19 summarizes IPC and bandwidth for classical binary encoding (using DCW), power-limited SPCM (PL-SPCM), SPCM [2], RAPID, and ideal-RAPID. In summary, RAPID improves IPC (bandwidth) by 15% (23%), 11.6% (9.7%), and 7.5% (6.7%) in comparison to classical binary encoding, PL-SPCM, and SPCM, respectively.

Instructions-per-cycle (IPC): To evaluate the impact of RAPID on IPC, we use a full-system simulator based on MARSS [12] and DRAMSim2 [13]. The simulation setup is described in detail in Section 5.6. We simulate nine composite workloads from Table 17. Fig. 35 shows the IPC for each benchmark in each workload. The last set of bars in each workload in Fig. 35 represents the mean IPC for that workload (since each benchmark is run on a separate and exclusive core, the mean IPC is given by the arithmetic mean of the IPCs of the constituent benchmarks of the workload [74, Ch. 1]). The last set of bars in Fig. 35 represents the harmonic mean of the IPCs of all the workloads (computing the harmonic mean is equivalent to running all the workloads in the same system for a fixed number of instructions).

Table 19: IPC and bandwidth for PL-SPCM, SPCM [2], RAPID, and ideal-RAPID, normalized to binary encoding.

<table>
<thead>
<tr>
<th></th>
<th>IPC</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical binary encoding</td>
<td>1×</td>
<td>1×</td>
</tr>
<tr>
<td>PL-SPCM</td>
<td>1.03×</td>
<td>1.12×</td>
</tr>
<tr>
<td>SPCM [2]</td>
<td>1.08×</td>
<td>1.15×</td>
</tr>
<tr>
<td>RAPID</td>
<td>1.15×</td>
<td>1.23×</td>
</tr>
<tr>
<td>Ideal-RAPID</td>
<td>1.16×</td>
<td>1.24×</td>
</tr>
</tbody>
</table>
Figure 36: Main memory bandwidth using full-system simulation (MARSS+DRAMSim2). RAPID improves bandwidth by 23%, 9.7%, and 6.7% over classical binary encoding, PL-SPCM, and SPCM [2], respectively.

We make the following observations from the Fig. 35: First, our simulations show a good correlation between IPC and both RPKI and MPKI. For example, consider the workloads WD4 (high MPKI) and WD9 (low MPKI)—as expected, the high cache miss-rate of WD4 lowers the IPC in comparison to WD9 that has a lower cache miss-rate. To understand the dependence of IPC on RPKI, let us consider the workloads WD4 (high RPKI) and WD3 (low RPKI). RAPID reduces latency and improves IPC only during read accesses. Thus the workload with higher RPKI should show a higher improvement in IPC in comparison to the workload with lower RPKI; our simulations, in agreement with the above argument, show that RAPID improves the IPC by 32% for WD4 (high RPKI) in comparison to 5% for WD3 (low RPKI). Second, our simulations also show similar correlations at the individual benchmark level. Intuitively, memory-intensive benchmarks like milc (WD6) or lbm (WD3) should have lower IPC in comparison to less memory-intensive benchmarks like zeusmp (WD8) or omnetpp (WD4); our simulations are in excellent agreement with the above argument. In order to study the impact of the power-limited NVM on IPC, we simulate PL-SPCM architecture that uses 3× write latency. In practice, the real behavior of SPCM should be somewhere in between that of PL-SPCM and SPCM. Our simulations show that RAPID improves IPC by 15.2%, 11.6%, and 7.5% in comparison to classical binary encoding, PL-SPCM, and SPCM [2], respectively. Furthermore, ideal-RAPID shows negligible improvement in IPC (¡0.5%) in comparison to RAPID.
Memory bandwidth: Figure 36 shows the main memory bandwidth across nine composite workloads for the 5 encodings considered in this chapter, normalized to classical classical binary encoding. The last set of bars in Fig. 36 represents the geometric mean of the normalized improvements (computing geometric mean is equivalent to running each workload for the same execution time). Intuitively, similar to IPC, workloads that have high RPKI should
contribute to higher bandwidth improvements in comparison to those with low RPKI. This can be seen using the example of workload WD_4 (high RPKI), which show an improvement of $1.4 \times$ in bandwidth using RAPID in comparison to WD_8 (low RPKI), which shows an improvement of only 3%. On the whole, our simulations show that RAPID improves the average memory bandwidth by 23%, 9.7% and 6.7%, in comparison to classical binary encoding, PL-SPCM, SPCM, respectively. Furthermore, ideal-RAPID has only 1% improvement in bandwidth over RAPID.

**Full-system read latency distribution:** Figure 37 shows the distribution of module-level read latency for the 5 encodings considered in this chapter. It is interesting to see the effect of different encodings on the read latency distributions. First, classical binary encoding always has a fixed latency for read; hence, we see that majority of the responses are all crowded in the [60-69] ns range. In contrast, SPCM has 3 different read response times depending on the cache line address. This is clearly seen in Fig. 37(b) and (c)—the reads are distributed almost uniformly between the ranges [30-39] ns and [60-69] ns. Finally, the read responses due to RAPID depend on its efficiency in predicting the CW at the write time. Since RAPID predicts the CW fairly accurately, the read response statistics show a bump in the [30-39] ns range. Furthermore, the bump in the [30-39] ns range is more pronounced in the case of ideal-RAPID, where the CW prediction is assumed to be 100% accurate. Thus, the read response statistics from the full-system simulation of all encodings considered in this chapter correlate with the intuition built in Sec 5.2.

**Lifetime:** We evaluate the NVM lifetime of (i) classical binary coding, (ii) state-of-the-art SPCM [2], and (iii) RAPID-ECP using a statistical simulator for TLC RRAM. Along the lines of [51], our simulation uses the following parameters: coefficient of variance (COV) of cell lifetime = 0.25, logical bit-flip probability = 0.5, which translates to a TLC update probability of 0.875, and mean TLC lifetime = $10^8$ writes [10]. There are two reasons why SPCM [2] degrades NVM lifetime in comparison to binary coding and RAPID-ECP. First, whereas binary coding and RAPID-ECP updates only 171 ($\lceil 512/3 \rceil$) data TLCs for every 512-bit cache line write, SPCM [2] updates 512 TLCs on every write, which increases the wear on each TLC by 3×. Second, whereas binary coding and RAPID-ECP can correct up
Figure 38: Capacity of TLC RRAM memory for SPCM [2] and RAPID-ECP. RAPID-ECP improves the memory lifetime by $4.12 \times$ over SPCM [2]. Note that RAPID-ECP and binary coding have equivalent memory lifetime.

to 18 hard errors, SPCM [2] can only correct up to 6 hard errors in a 512-TLC physical cache line. Fig. 38 shows that RAPID-ECP, which has error correction capability equivalent to classical binary coding, improves NVM lifetime by $4.12 \times$ in comparison to state-of-the-art SPCM [2].
5.8 RELATED WORK AND RAPID

Historically, a broad set of solutions were proposed to reduce the write latency challenges associated with SLC NVM. Although the solutions for SLC NVMs ease the pathway for the full-scale adoption of MLC/TLC NVMs, they do not address the specific challenges of working with the iterative read and write procedures of MLC/TLC NVMs. We begin by describing the solutions focused on reducing the MLC/TLC NVM read latency. This is followed by a discussion on solutions that aim at reducing the MLC/TLC write latency. Finally, we conclude this section with a discussion on related works on critical word forwarding schemes.

Read latency improvement: MLC/TLC NVMs exhibit asymmetric read latencies due to the nature of the read procedure, which applies a sequence of read strobes for decoding the stored bits sequentially. Line Pairing (LP) [86], proposed for MLC STT-MRAM caches, combines two physical cache lines and reorganizes the data such that the cache lines are encoded using only MSbs/LSbs exclusively, creating read-fast-write-slow (RFWS) and read-slow-write-fast (RSWF) cache lines. Line Swapping (LS) [86], proposed alongside LP, maps read-/write-intensive data to RFWS/RSWF cache lines to reduce both read/write latencies.

Extending the observations of LS and LP from embedded caches and applying to main memories, decoupled bit mapping (DBM) [1] proposes encoding of contiguous chunks of logical addresses to MSbs/LSbs alone, creating fast-read/fast-write regions. DBM uses asymmetric page mapping (APM), which works with the OS to map read-/write-intensive pages to fast-read/fast-write memory regions. Although DBM reduces the read and write latency of the main memory, they require OS-level software support for its latency advantages. These page migration policies incur additional write overheads. Furthermore, since the data stored in a physical cache line cells belong to completely different addresses, it reduces the row buffer hits due to lack of spatial locality. In addition to these limitations, DBM is also limited by write amplification; a write to an address is spread out on the MSbs/LSbs of twice the number of cells.

In contrast, Striped PCM (SPCM) [2] proposes mapping of odd/even cache line addresses using MSbs/LSbs exclusively. Unlike DBM, SPCM does not require OS support,
no data migration overhead, and has spatial locality. SPCM uses paired-writes to contain the write amplification due encoding using only MSbs/LSbs, however, they are scale poorly for TLC or higher level cell NVMs. Furthermore, since NVM memory modules require high current/power, they are generally power-limited systems that allow only a limited number of writes to proceed at a given time [25]. In such power-limited systems, SPCM would require its writes to proceed sequentially. Although RAPID’s encoding also to leverages the fast MSbs for read acceleration, it does so for zero write overheads.

Write pausing and write cancellation (WPWC) [17] recognizes the criticality of read operations and prioritizes reads over writes. Since, long latency write operations can potentially block other requests until the operation is completed, WPWC pauses the write operation if a read operation is initiated after the write has already begun. The innovations of WPWC are orthogonal to RAPID.

**Write latency improvement:** Due to the resistance drift and process variation, MLC/TLC NVMs employ iterative program-and-verify (P&V) procedure for programming an NVM cell. Due to the nature of P&V certain states in MLC/TLC NVMs require more latency and energy to program in comparison to the other states. Write truncation (WT) [87] identifies that writing a cache line can be limited by few small number of cell updates that take long duration, even though the other cells were written to in smaller duration. WT proposes that we truncate these few long latency writes and invest on a stronger ECC; in effect, WT is a trade-off of memory area for latency improvement. Another class solutions have identified ways to limit data encoding to only the low latency/energy states of MLC/TLC NVM [35–37, 78, 79] —trading off memory area for latency encoding. Furthermore, recent works in this direction have worked towards leveraging data compression to recover memory area and trade it for latency improvement [9, 24, 88]. RAPID is orthogonal to all these innovations for write latency improvements in NVMs.

**Critical word (CW) regularity:** It is well known that optimizing CW read response improves system performance [74, 89]. Recent works have observed regularity in CW, which makes it easy to predict the CWs at write time, as opposed to read time CW optimization [49, 84, 90]. [84] uses extensive program data analysis and reports different types of CW
regularity and proposes several cache-based optimizations for expediting CWs. [49] extends the observations of [84] to store CWs separately in a low latency memory to improve the read performance. Although RAPID leverages these innovations in identifying CWs during write time, it is agnostic to the CW-optimization technique.

5.9 SUMMARY

Our full-system evaluation of a TLC RRAM architecture show that RAPID reduces read latency by 11% and 21%, and energy by 24% and 41% in comparison to state-of-the-art striped PCM (SPCM) and conventional binary encoding with data-comparison write (DCW, i.e., read-modify-write), respectively. This translates to a 7.5% (15%) improvement in IPC and a 4.12× improvement in endurance over SPCM (binary encoding) across a comprehensive workload of SPEC CPU2006 benchmarks. In summary, this chapter presented a simple and holistic solution to mitigate the high read latency of MLC/TLC NVMs for zero write latency and write energy.
6.0 CONCLUSIONS

MLC/TLC NVMs such as phase-change RAM and resistive RAM are the subject of active research and development as replacement candidates for DRAM, which is limited by its high refresh power and poor scaling potential. Besides the benefits of non-volatility (low refresh power) and improved scalability, MLC/TLC NVMs offer high data density and memory capacity over DRAM. However, the viability of MLC/TLC NVMs is limited due to (i) high programming energy and latency as well as the low endurance of NVM cells, (ii) data security vulnerability due to non-volatility, and (iii) high read latency. This dissertation addressed the aforementioned challenges by making the following contributions:

First, this dissertation described CompEx coding, a low overhead, dynamic tradeoff framework that synergistically integrates pattern-based compression with expansion coding to realize simultaneous energy, latency, and lifetime improvements in MLC/TLC NVMs. The core idea of CompEx coding is to selectively apply expansion codes, i.e., linear block codes that encode data using only the low energy states of an MLC/TLC cell to compressed data, thereby ensuring that the resulting data in expansion-coded form will not exceed the original data width. CompEx coding is agnostic to the choice of compression technique; in this dissertation, we evaluated CompEx coding using both frequent pattern compression (FPC) and base-delta-immediate (B∆I) compression. CompEx coding integrates FPC/B∆I with \((k,m)_q\) ‘expansion’ coding; expansion codes are a class of \(q\)-ary linear block codes that encode data using only the low energy states of a \(q\)-ary NVM cell. This dissertation also presented CompEx++ coding, which extends CompEx coding by leveraging the variable compressibility of pattern-based compression techniques. CompEx++ coding integrates custom expansion codes to each of the compression patterns to exploit the maximum energy/latency benefits of CompEx coding.
Second, this dissertation presented CASTLE, a compression-based architecture that provides a read-decrypt-free, i.e., write-only block-level secure solution for low latency, low energy durable NVMs. CASTLE adopted a block-level write-only sequence to completely eliminate the read latency of the read-decrypt steps in state-of-the-art methods for NVM security. Whereas a write-only approach increases cell updates, and thereby energy and latency, CASTLE integrated memory compression and partial data mapping to realize energy reductions and lifetime improvements over state-of-the-art methods for NVM security. CASTLE incurs only 1.5% memory overhead and negligible logic overhead; CASTLE is also compatible with soft/hard error detection and correction support (ECC/ECP) in NVMs.

Third, this dissertation presented RAPID, a no-overhead critical-word-first Read Acceleration architecture for Improved Performance and Durability in MLC/TLC NVMs. We showed that RAPID reduces the latency of the read operation by write-time encoding of the critical word (CW) in a cache line using only the most significant bits of the TLC; as opposed to read-time CW optimization, RAPID relies on write-time CW optimization, which is a relatively new area of research [49]. In addition, RAPID is a zero-cost solution; we showed that unlike existing solutions that incur up to 3\times write overhead, RAPID does not incur any write overhead. Finally, we showed that RAPID seamlessly integrates with error correction techniques like ECC [50] and ECP [51] and state-of-the-art counter mode memory encryption schemes.

Thus, this dissertation addressed the core challenges of write/read energy and latency, endurance, and security of MLC/TLC NVM main memories and provides a complete and holistic solution to these challenges.
7.0 FUTURE WORK

Although this dissertation presents solutions to address the core problems of MLC/TLC NVMs like energy, latency, endurance, and security, there are other problem areas of MLC/TLC NVMs requiring attention. The remainder of this chapter is a summary of these problem areas and potential directions for the future research.

1. CompEx/CompEx++ coding addressed the write energy/latency and endurance problems by leveraging data compression to avoid undesirable states of MLC/TLC NVM; however, since data compression is not possible for 100% of the writes, CompEx/CompEx++ coding is not successful 100% of the time. Furthermore, several applications in the domains of media processing, computer vision, data mining, etc., exhibit poor data compressibility [91]. Thus, future investigations directed towards novel data encoding strategies, which use data shaping to inherently concentrate entropy in the desirable states, can address the data compression related limitations of CompEx/CompEx++ coding.

2. CASTLE provided a high performance, low energy/latency, and durable alternative to state-of-the-art security solutions; however, counter mode encryption schemes (including CASTLE) are limited by the periodic re-encryption overheads, which causes system freeze, and requires future research and development effort. The re-encryption overhead can be looked at as a periodic denial of service problem, which can be addressed using both system-level strategies (graceful system degradation, traffic shaping, etc.) and bank-level strategies (preemptive re-encryption during idle cycles, time-staggered re-encryption of individual banks, etc.).

3. In addition to NVM encryption, NVM main memory authentication is becoming important to secure NVMs against data tampering attacks. State-of-the-art NVM authen-
tication schemes using merkle trees incur high memory area and write energy/latency overheads, since they require multiple writes to the merkle tree for each write access; thus, this is an important potential area of future research. Furthermore, obliviousness, a property where the all the memory traces are computationally indistinguishable is also becoming an important security requirement. Oblivious RAMs (ORAM) are used to achieve memory obliviousness, which incur extremely high memory area and bandwidth overheads. This presents us with opportunities for the future research.

4. RAPID addresses the read latency challenges of MLC/TLC NVMs using critical word (CW) identification, which is not possible for 100% of the reads. The future work in this direction could focus on leveraging processor and compiler support for accelerating a larger fraction of the read accesses. In addition, architectural solutions involving hybrid memory technology, where a faster low capacity memory and a slower high capacity NVM, are emerging to address the latency challenges of NVMs.

5. MLC/TLC NVMs suffer from high resistance drift, which manifests as high bit error rates. This requires stronger error correcting codes (ECC), which incur high memory overheads, to recover from errors. In addition, ECC are designed with the assumption that every error pattern is equally likely; however, in practice, some error patterns are more common in comparison to the others due to the program data behavior, process variation, etc. Thus, investing in error-pattern-aware ECC, which can target specific set of error patterns can provide improved error correction for lower overhead. A potential direction towards developing such solutions is the application of the principles of machine learning to ECC.

6. Finally, NVMs like RRAM are finding their applications not only in data storage, but also in logic circuits, demonstrating its potential in computing systems. Neumorphic computing, which is a computing paradigm mimicking neurobiological architecture, is gaining research momentum; recent research suggests that RRAM is highly suitable for the development of such neuromorphic computers [92]. Thus, future work in this direction could increase the viability of NVMs in neuromorphic computing systems.
BIBLIOGRAPHY


