CompEx++: Compression-Expansion Coding for Energy, Latency, and Lifetime Improvements in MLC/TLC NVMs

POOVAIAH M. PALANGAPPA and KARTIK MOHANRAM, University of Pittsburgh

Multilevel/triple-level cell nonvolatile memories (MLC/TLC NVMs) such as phase-change memory (PCM) and resistive RAM (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. In addition to the benefits of nonvolatility (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 primarily due to the high programming energy and latency as well as the low endurance of NVM cells; these are primarily attributed to the iterative program-and-verify procedure necessary for programming the NVM cells.

This article proposes compression-expansion (CompEx) coding, a low overhead scheme that synergistically integrates pattern-based compression with expansion coding to realize simultaneous energy, latency, and lifetime improvements in MLC/TLC NVMs. CompEx coding is agnostic to the choice of compression technique; in this work, we evaluate CompEx coding using both frequent pattern compression (FPC) and base-delta-immediate (BDI) compression. CompEx coding integrates FPC/BDI 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. CompEx coding simultaneously reduces energy and latency and improves lifetime for negligible-to-no memory overhead and negligible logic overhead ($\approx 10k$ gates, which is $< 0.1\%$ per NVM module). Furthermore, we also propose 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 maximum energy/latency benefits of CompEx coding. Our full-system simulations using TLC RRAM show that CompEx/CompEx++ coding reduces total memory energy by 57%/61% and write 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 $\times$ improvement in lifetime over classical binary coding using data-comparison write. CompEx/CompEx++ coding thus addresses the programming energy/latency and lifetime challenges of MLC/TLC NVMs that pose a serious technological roadblock to their adoption in high-performance computing systems.

CCS Concepts: • **Hardware** $\rightarrow$ *Memory and dense storage*;

Additional Key Words and Phrases: Nonvolatile memories, main memory, compression, energy, latency

ACM Reference Format:

Poovaiah M. Palangappa and Kartik Mohanram. 2017. CompEx++: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVMs. *ACM Trans. Archit. Code Optim.* 14, 1, Article 10 (April 2017), 30 pages.

DOI: <http://dx.doi.org/10.1145/3050440>

1. INTRODUCTION

The ITRS projects that resistance-class nonvolatile memory (NVM) technologies such as phase-change memory (PCM) [Lee et al. 2009] and resistive RAM (RRAM) [Baek

This research was supported in part by NSF CAREER Award CCF-1208933 and in part by NSF Award CCF-1217738.

Authors' addresses: P. M. Palangappa and K. Mohanram, Department of Electrical and Computer Engineering, 1238 Benedum Hall, Pittsburgh, PA 15261; emails: pmp30@pitt.edu, kartik.mohanram@gmail.com.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/04-ART10 \$15.00

DOI: <http://dx.doi.org/10.1145/3050440>

Fig. 1. (a) Programming energy and latency trends for SLC, MLC, and TLC RRAM [Niu et al. 2013]. (b) Performance penalties for MLC/TLC RRAM in comparison to SLC RRAM for the Stream benchmarks [McCalpin 1995] using MARSS [Patel et al. 2011] and DRAMSim2 [Rosenfeld et al. 2011]. The key take-away is that the move to denser technologies like MLC/TLC usually degrades IPC, bandwidth, and energy.

et al. 2004] are suitable replacement candidates for DRAM due to their low refresh power and better scaling potential [ITRS 2011]. 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, multilevel/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 [Bedeschi et al. 2009]. MLC/TLC NVMs offer higher data density in comparison to single-level cell NVMs (SLC NVMs) since they can store more than one logical data bit. However, MLC/TLC NVMs aggravate the challenges of working with SLC NVMs, primarily due to (i) increased resistance drift, which diminishes nonvolatility [Yoon et al. 2013], and (ii) the complex, iterative program-and-verify (P&V) procedure necessary to program MLC/TLC NVM cells [Nirschl et al. 2007], which increases access latencies as well as power/energy requirements [Xu et al. 2013] and reduces cell lifetime due to high currents and multiple iterations on writes [Jiang et al. 2012a].

Problem. 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. Figure 1(a) shows the energy and latency versus capacity trends for SLC, MLC, and TLC RRAM [Niu et al. 2013]. This increase in energy and latency for programming an MLC/TLC NVM cell translates to system-level performance penalties in terms of instructions per cycle (IPC), bandwidth, and energy as shown in Figure 1(b), which is measured for RRAM by using the Stream benchmark suite [McCalpin 1995] on MARSS [Patel et al. 2011] and DRAMSim2 [Rosenfeld et al. 2011]. These challenges of MLC/TLC NVMs pose a serious technological roadblock to their adoption in high-performance computing systems, which is the core focus of this article.

Related work and motivation. 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 [Cho and Lee 2009; Yang et al. 2010; Palangappa et al. 2015; Palangappa and Mohanram 2015], write scheduling [Qureshi et al. 2010, 2012; Kim et al. 2012; Kwon et al. 2012], data migration using address translation [Qureshi et al. 2009; Wu et al. 2012; Seong et al. 2010], and architectural improvements [Jiang et al. 2012c; Yue and Zhu 2012; Dgien et al. 2014]. 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 those of Sardashti and Wood [2013], Ahn et al. [2001], Chen et al. [2010], Yang et al. [2000], Yang and Gupta [2002], Mahaptra et al. [2005], Yim et al. [2004], Dgien et al. [2014], Ekman and Stenstrom [2005], and Arelakis and Stenstrom [2014] approach this problem by

reducing the number of cell writes per write access, data encoding solutions such as those of Jiang et al. [2012a, 2012b], Lee et al. [2012], Wang et al. [2011], Yoon et al. [2013], and Niu et al. [2013] 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 simultaneously realize the advantages of both data compression and data coding for no memory overhead.

Contributions. This article is an extension of the work of Palangappa and Mohanram [2016], which first introduced compression-expansion (CompEx) coding, a low overhead scheme that synergistically integrates pattern-based 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 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 in 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 work CompEx coding is evaluated using two compression techniques: frequent pattern compression (FPC) [Alameldeen and Wood 2004] and base-delta-immediate (BDI) compression [Pekhimenko et al. 2012]. 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 offline/online application profiling. BDI is a compression technique that is similar to FPC but operates at cache line level instead of word level. BDI takes advantage of regularity of data storage and the limited dynamic range of stored data. Following compression, the compressed data is selectively encoded using $(k, m)_q$ expansion coding. Expansion codes are a class of q -ary linear block codes that encode data using only p lowest energy/latency cell states ($p < q$). This article evaluates two examples of CompEx coding: $(3, 2)_8$ and $(6, 5)_8$ expansion coding. Expansion codes are studied theoretically in the context of MLC/TLC NVMS, and theoretical estimates for the maximum achievable energy reductions for MLC/TLC NVMS are derived. The expected energy reduction for $(3, 2)_8/(6, 5)_8$ expansion coding is $2.3 \times / 1.4 \times$ over classical binary encoding; in close agreement with these estimates, our results on the SPEC CPU2006 [SPEC 2006] benchmark suite show that it is possible to realize energy reductions of $2.1 \times / 1.3 \times$ in practice.

Extended contributions. This article extends the work of Palangappa and Mohanram [2016] with the following contributions. First, we propose 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 maximum energy/latency benefits of CompEx coding. Along the lines of CompEx coding, we evaluate CompEx++ coding for FPC and BDI by integrating $(3, 2)_8$ and $(3, 1)_8$ expansion coding. Second, 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.

Evaluation. In this work, the efficiency of CompEx/CompEx++ coding is evaluated on a 64-bit word, 64-byte cache line architecture—that is, every word is stored using

22 ($\lceil 64/3 \rceil$) TLCs, and every cache line is stored using 171 ($\lceil 512/3 \rceil$) 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 B Δ I. 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 B Δ I and 65 logical bits for FPC) and absorbed into the last TLC at no cost ($\lceil 513/3 \rceil = \lceil 512/3 \rceil = 171$ TLCs for B Δ I and $\lceil 65/3 \rceil = \lceil 64/3 \rceil = 22$ 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++ codec¹ is $\approx 10\text{k}/12.5\text{k}$ gates ($<0.1\%$ per NVM module); although the codec has a latency of two to three cycles, we show that this can be hidden through memory access pipelining.

Results. Our full-system simulations of a system that integrates TLC RRAM using MARSS [Patel et al. 2011] and DRAMSim2 [Rosenfeld et al. 2011] for workloads from the SPEC CPU2006 [SPEC 2006] 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) [Yang et al. 2010] (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, multibillion-instruction evaluation of CompEx/CompEx++ coding, we use NVMain [Poremba and Xie 2012] with the memory traces of SPEC CPU2006 benchmarks generated using the Intel Pin toolset [Luk et al. 2005]. 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 B Δ I, respectively. Simultaneously, 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 B Δ I, respectively, and improves TLC RRAM lifetime by $1.8\times$.

This article is organized as follows. Section 2 provides the background for energy/latency problems in MLC/TLC NVMs. Section 3 describes CompEx coding and CompEx++ coding with examples. Section 4 presents the evaluation methodology and the simulation setup. Section 5 presents evaluation results. Section 6 discusses related work, and Section 7 presents our conclusions.

2. BACKGROUND AND MOTIVATION

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 [Lee et al. 2008; Kang et al. 2011; Wong et al. 2012; Yue and Zhu 2013]. However, most MLC/TLC NVMs suffer from higher write energy [Yue and Zhu 2013; Xu et al. 2013] and latency [Jiang et al. 2012d] per cell, limited lifetime [Lee et al. 2009], and asymmetric read/write latencies [Kang et al. 2007; Choi et al. 2012; Jog et al. 2012; Xu et al. 2013; Yue and Zhu 2012]. In this

¹In this work, only the logic overhead of the FPC-based CompEx codec is evaluated. The logic overhead of the B Δ I-based CompEx codec is based on the original proposal of classical B Δ I [Pekhimenko et al. 2012].

Fig. 2. Energy and latency for TLC RRAM [Niu et al. 2013] using P&V across eight states. States 0 through 3 and 4 through 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 takeaway is that the overall energy and latency can be reduced by encoding data using only the low energy states.

section, we discuss the procedure used for programming an MLC/TLC NVM cell and its associated limitations.

MLC/TLC NVM P&V. MLC/TLC NVM exhibits nondeterministic behavior due to process variation, variation in material composition, and resistance drift. This nondeterminism 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. However, since a TLC NVM has eight 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 depending on the proximity of the target state to the full SET/RESET states as shown in Figure 2. Therefore, the write energy and latency for MLC/TLC NVM can vary depending on the final state to which the cell is being programmed. For example, Figure 2 shows the average energy and latency for programming a TLC RRAM into different states [Niu et al. 2013]. States 0 and 7 require the lowest energy/latency since they can be programmed using a single P&V iteration, whereas states 3 and 4 require maximum energy/latency since they require the most number of P&V iterations to be brought to the desired resistance range.

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 requires much higher current and latency than their SLC counterparts due to the iterative P&V method. Second, 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 [Yue and Zhu 2012]. 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 [Xu et al. 2013]. 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 [Xu et al. 2013] in comparison to SLC NVM, which has a lifetime of 10^{8-10} writes [Lee et al. 2009; Wong et al. 2012; Yue and Zhu 2013].

Table I. The 64-Bit FPC Patterns (Inclusive of 3-Bit Prefix, Indicated in Red)

Prefix	Pattern Encoded	Example	Compressed Example	Encoded Size	Value Space	Compression Contribution
000	Zero run	0x0000000000000000	0x0	3 bits	1 value	29.7%
001	8-bit sign extended	0x000000000000007F	0x17F	11 bits	255 values	1%
010	16-bit sign extended	0xFFFFFFFFFFFFB6B6	0x2B6B6	19 bits	65,280 values	1%
011	Half-word sign extended	0x0000000076543210	0x376543210	35 bits	$\approx 2^{32}$ values	2.1%
100	Half-word, padded with a zero half-word	0x7654321000000000	0x476543210	35 bits	$\approx 2^{32}$ values	16.4%
101	Two half-words, each a byte sign extended	0xFFFFBEEF00003CAB	0x5BEEF3CAB	35 bits	$\approx 2^{32}$ values	8.3%
110	Word consisting of four repeated double bytes	0xCAFECAFECAFECAFE	0x6CAFE	19 bits	65,534 values	1.1%

Note: Illustrative examples are given in columns 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 [SPEC 2006] benchmark suite.

3. CONTRIBUTIONS

Moving to an MLC/TLC technology from an SLC technology (and similarly to a TLC technology from an MLC technology) enables static trade-offs between memory density/capacity and memory energy/latency. We motivated the technology considerations and the system-level implications of these trade-offs in Figure 1. However, to the best of our knowledge, there has been no systematic effort to dynamically leverage these trade-offs 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/TLC 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 trade-off 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 [Alameldeen and Wood 2004] and BAI [Pekhimenko et al. 2012]. 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 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/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—FPC and BAI—possess these desirable traits for integration into CompEx/CompEx++ coding.

Frequent pattern compression. 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 [Alameldeen and Wood 2004]. More recently, FPC was extended to reduce bit writes in NVMs [Dgien et al. 2014]. In this work, we extend FPC to a 64-bit word using the patterns tabulated in Table I. 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 that is represented by column 1 of the table. A 1-bit tag is used to indicate whether the written data is its compressed/uncompressed 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 show examples for each of the data patterns along with their compressed form and compressed data size. Column 6 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 [SPEC 2006] benchmark suite.

Base-Delta-Immediate. B Δ I was originally proposed for on-chip cache data compression [Pekhimenko et al. 2012] by storing the compressed data using a “base” (B) and “delta” (Δ), a series of offsets with respect to B . B Δ I proposes that a cache line $C = \{V_0, V_1, \dots, V_{n-1}\}$ can be compressed and written as $X = \{B, \Delta_0, \Delta_1, \dots, \Delta_{n-1}\}$,² along with a 4-bit tag, where $B = V_0$ (definition), $\Delta_i = V_i - B$, $n = \text{sizeof}(\text{cache line})/k$, and $k = \text{sizeof}(\Delta)$. It is reported that data stored in a cache line is often regular, with limited dynamic range [Pekhimenko et al. 2012]. 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, B Δ I maintains a pattern table for compression. B Δ I leverages regularity in data to compress cache lines using 64-byte patterns; Table II shows the eight different cache line patterns that B Δ I is capable of compressing, along with illustrative examples for each of the patterns. The last column represents the percentage of data words compressed by each pattern (B Δ I cumulatively compresses about 46% of write accesses) in trace-based simulations of the SPEC CPU2006 [SPEC 2006] benchmark suite. Given a cache line, it is matched against each row (pattern) of the table. 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 Δ 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.

3.2. Expansion Coding

As described in Section 2, the iterative 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. 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 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 codewords of length k encoding q^m q -ary message words. Expansion code encodes data using p

²This work integrates B Δ I with a single base, without the implicit second base, for simplicity. This is similar to the base+delta technique proposed in Pekhimenko et al. [2012]. 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 B Δ I pattern that was used for compression.

Table II. The 64-Byte BAI Patterns (with 3-Bit Prefix, Indicated in **Red**) Along with Illustrative Examples and Encoded Size Postcompression

Prefix	Pattern	Example	Compressed Example	Encoded Size*	Compression Contribution
000	Zeros	0x000000000000 000000000000 ... 000000000000	0x0	0 bytes	24.7%
001	Rep. values	0xFEEDCOFFEECOFFEE FEEDCOFFEECOFFEE ... FEEDCOFFEECOFFEE	0x1FEEDCOFFEECOFFEE	8 bytes	6.8%
010	Base8-Δ1	0xABCDABCDABCDAB00 ABCDABCDABCDAB01 ... ABCDABCDABCDAB08	0x2ABCDABCDABCDAB00, 01, ..., 08	15 bytes	0.2%
011	Base8-Δ2	0xABCDABCDABCD0000 ABCDABCDABCD00FA ... ABCDABCDABCD5A5A	0x3ABCDABCDABCD0000, 00FA, ..., 5A5A	22 bytes	0.2%
100	Base8-Δ4	0xABCDABCD00000000 ABCDABCD00000000 ... ABCDABCD00000000	0x4ABCDABCD00000000, BEEFFED, ..., COFFEECDC	36 bytes	1.3%
101	Base4-Δ1	0xABCDAB00 ABCDAB01 ABCDAB02 ... ABCDABFF	0x5ABCDAB00, 01, 02, ..., FF	19 bytes	3.2%
110	Base4-Δ2	0xABCD0000 ABCDFEED ABCDFOOF ... ABCDEEDC	0x6ABCD0000, FEED, FOOF, ..., EEDC	34 bytes	6.6%
111	Base2-Δ1	0xAB00 ABFF ABED AB0F ... ABDC	0x7AB00, FF, ED, OF, ..., DC	35 bytes	2.7%

*First, the encoded size is 3 bits in addition to this column's contents (for prefix). Second, since $B = V_0$ by definition, storing Δ_0 is redundant and hence omitted.

Note: On the whole, BAI can compress about 46% of write accesses for the benchmarks from the SPEC CPU2006 [SPEC 2006] benchmark suite. Note that although the compression contributions of FPC and BAI are comparable, the compression ratios are very different. For example, the bulk of the compression from “zero values” in both FPC and BAI have comparable compression contributions. Whereas FPC compresses data from 64 bits to 3 bits, BAI compresses 64 bytes (512 bits) to 3 bits. Hence, we expect to see higher reduction in energy using BAI in comparison to FPC. However, since latency is limited by the worst-case cell-write delay, we expect latency improvement using BAI to be comparable to that of FPC.

Fig. 3. $(k,m)_q$ expansion coding. An m -cell q -ary message is encoded to a k -cell p -ary ($p < q$) message.

Binary Data	1	0	1	0	1	0	0	1	1	1	0	0	1	1	0	0	(a)
Binary coding	5				2			3			4			6		0	
Energy = 120.1pJ																TLCs= 6	
Binary Data	1	0	1	0	1	0	0	1	1	1	0	0	1	1	0	0	(b)
(3,2) ₈ exp. coding	6	6			6			1	7		0		7			0	
Energy = 39.2pJ																TLCs= 8	
Binary Data	1	0	1	0	1	0	0	1	1	1	0	0	1	1	0	0	(c)
(6,5) ₈ exp. coding	3				3			1			1		1			0	0
Energy = 94.3pJ																TLCs= 7	

Fig. 4. Comparison of binary, $(3,2)_8$ expansion coding and $(6,5)_8$ expansion coding (IDM [Niu et al. 2013]). Energy for $(3,2)_8$ expansion coding < $(6,5)_8$ expansion coding < binary coding.

lowest energy states of total q states ($p < q$), where $p = \lceil q^{(m/k)} \rceil$, incurring memory overhead of at least $(k/m) - 1$ (see Chapter 3 of Costello and Lin [2004]).

We elaborate on the $(k,m)_q$ expansion code with the help of Figure 3. 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 this 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 figure. All of these messages are mapped to k cells, where each cell can represent only p states logically (although physically they are still capable of storing q states). This is represented using the right-hand side of the figure. The code expands an m -cell word to a k -cell word after encoding, hence the name *expansion code*. Expansion coding lowers both NVM energy and latency in practice, as the lowest energy TLC states also require fewer P&V cycles (Figure 2).

Example. For ease of understanding, we illustrate $(3,2)_8$ expansion coding and incomplete data mapping (IDM) [Niu et al. 2013], 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 Figure 4. In the regular case, data is encoded using all eight TLC states as shown in Figure 4(a). Every slice of three 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 four ($\lceil 8^{(2/3)} \rceil$) out of the eight TLC states to encode the data (Figure 4(b)). Every slice of two logical data bits is stored in one TLC, requiring eight ($=\lceil 16/2 \rceil$) TLCs and 39.2pJ. In this example, although expansion coding results in energy and latency reduction, it incurs 50% memory overhead. IDM [Niu et al. 2013] seeks to lower the memory overhead to 20% by using the $(6,5)_8$ expansion code utilizing the six lowest energy TLC states. Whereas binary coding encodes three information bits into each TLC, IDM encodes five data bits into two TLCs. Since two 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 (Figure 4(c)), binary encoding uses six TLCs to represent 16 bits and requires 120.1pJ. However, using IDM, 16 data bits can be encoded using seven TLCs (using six states) and 94.3pJ.

Fig. 5. Illustration of CompEx coding. The key takeaway is that CompEx coding reduces energy for no overhead since the tag bit is encoded within the last TLC.

Although IDM in combination with dynamic data remapping has been shown to improve the lifetime of MLC/TLC NVMs [Niu et al. 2013], 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.

3.3. CompEx Coding

The core idea of this work 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, 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 BAI are excellent candidates for this purpose due to their ability to compress a wide range of data for low overhead (refer to Tables I and II). Furthermore, since the largest size of an FPC-compressed 64-bit word (or BAI-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 a 16-bit word as an example, which is compressed to 8 bits using FPC as shown in Figure 5. In the first case, we encode the data using all 8 TLC states from Figure 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 Figure 5(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 BAI-compressed 64-byte cache line to a maximum of 146 ($\lceil (36 \times 8 + 4)/2 \rceil$) 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 BAI-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 BAI). Consider the 16-bit example shown in Figure 5(b), where the tag bit is appended to the end of the data word and encoded into the last TLC. Since 2^n ($n \geq 1$) 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.

Fig. 6. (a) 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 outweighs this overhead. (b) Logic overhead for synthesized CompEx codec.

3.3.1. Theoretical Analysis of CompEx Coding. As we have established that 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 Figure 2. We start by calculating the average energy for TLCs with eight states, followed by the average energy for only four lowest energy TLC states. First, we assume that during writes, all permitted states of a TLC are equally likely to be programmed to Cho and Lee [2009]. Thus, the average/expected energy for a TLC with eight equally likely states (i.e., binary coding) is given by the sum of the entries for energy for all states in Figure 2 divided by 8 ($= 16\text{pJ}$ for this case)—that is, $E[\text{Energy}_{\text{binary}}] = \sum_{i=0}^7 e_i = 16\text{pJ}$. Here, e_i is the energy required for programming the TLC RRAM to the i^{th} state. Second, the average energy for a TLC that utilizes only zero, one, six, and seven 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 ($= 4.7\text{pJ}$ for this case)—that is, $E[\text{Energy}_{(3,2)_8}] = \sum_{i=0,1,6,7} e_i = 4.7\text{pJ}$. 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 energy reduction using $(3,2)_8$ expansion code is $E[\text{Energy}_{\text{binary}}]/E[\text{Energy}_{(3,2)_8}] = (16/4.7) (2/3) \approx 2.3$. 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 $E[\text{Energy}_{\text{binary}}]/E[\text{Energy}_{\text{CompEx}}] = 2.3 p$, 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 Section 5.1) closely agrees with the theoretical results—that is, $2.1 \times / 1.3 \times$ practical energy reduction in comparison to $2.3 \times / 1.4 \times$ derived theoretically for $(3,2)_8 / (6,5)_8$ expansion coding.

3.3.2. CompEx Codec Architecture. The architecture³ 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 Figure 6(a). The CompEx codec (encoder-decoder) logic is embedded completely inside the NVM module controller, between the NVM array and the data bus, to seamlessly encode and decode the accessed data as shown in Figure 6(a). 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

³For a multichannel memory system, each channel uses a dedicated codec, which results in a small overhead (Section 3.3.3) for the realized performance gains.

Fig. 7. Schematic diagram for an 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 the 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.

followed by decompression logic. FPC-based CompEx coding uses word-size write/read accesses, whereas BAI-based CompEx coding uses cache line size accesses.

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 to all of 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 BAI), 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. However, 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 (Figure 7). The design of the BAI-based compressor is assumed from the original BAI proposal [Pekhimenko et al. 2012]. The estimated logic overhead for the CompEx codec tabulated in Figure 6(b) is $\approx 10k$ 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.

3.4. CompEx++ Coding

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 BAI use multiple

Fig. 8. (a) Illustration of CompEx++ coding using (3,1)₈ expansion coding as opposed to (3,2)₈ expansion coding used in CompEx coding. The key takeaway 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. (b) Illustration of compression prefix decoding for CompEx++ coding.

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 (BAI) compresses 64-bit (512-bit) data using 22 (171) TLCs to 3 to 35 bits (3 to 292 bits), as illustrated in Table I (Table II). Although the (3,2)₈ expansion code is a good choice when we consider FPC or BAI as a whole, it might not be the optimum choice for each pattern individually. To find the best choice for each of the patterns, we need to integrate a custom expansion code, which is 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 six TLCs for storage, as illustrated in Figure 8(a). The incoming write data, as illustrated in Figure 8(a)(i), requires six 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 Figure 8(a)(ii). Second, the regular CompEx coding that uses (3,2)₈ expansion coding encodes the compressed data by packing two logical bits into each TLC; each 2-bit group is encoded using the four low energy states of the TLC (0, 1, 6, and 7) resulting in 16.7pJ, as illustrated in Figure 8(a)(iii). Finally, since the compressed data size is about a quarter of the original size, it is possible to encode data using (3,1)₈ expansion coding. Figure 8(a)(iv) illustrates data encoding using (3,1)₈ expansion coding; this requires five TLCs for data encoding, which is less than the originally available six TLCs, for only 8.5pJ. Thus, by making a smart choice of expansion codes (shown in Table III) that is customized for each compression pattern, CompEx++ coding reduces the overall energy/latency 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

Table III. Chosen Expansion Codes for the Compression Patterns of FPC and BΔI

FPC Pattern	Compressed Size	Expansion Code	BΔI Pattern	Compressed Size	Expansion Code
Zero run	3 bits	$(3,1)_8$	Zeros	0 bytes	$(3,1)_8$
8-bit sign extended	11 bits	$(3,1)_8$	Rep. values	8 bytes	$(3,1)_8$
16-bit sign extended	19 bits	$(3,1)_8$	Base8-Δ1	15 bytes	$(3,1)_8$
Half-word sign extended	35 bits	$(3,2)_8$	Base8-Δ2	22 bytes	$(3,1)_8$
Half-word, padded with a zero half-word	35 bits	$(3,2)_8$	Base4-Δ4	36 bytes	$(3,2)_8$
Two half-words, each a byte sign extended	35 bits	$(3,2)_8$	Base4-Δ1	19 bytes	$(3,1)_8$
Word consisting of four repeated double bytes	35 bits	$(3,2)_8$	Base4-Δ2	34 bytes	$(3,2)_8$
			Base2-Δ1	35 bytes	$(3,2)_8$

coding, and hence it becomes important to carefully encode the prefix bits to eliminate ambiguity in decoding and latency overheads during encoding.

Figure 8(b) illustrates the encoding and decoding of compression prefixes. Since both CompEx coding and CompEx++ coding do not require any modifications to the read circuit, the encoded data is always decoded for conventional binary encoding—that is, each TLC is decoded to be in one of the eight 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 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 two TLCs, and $(3,1)_8$ expansion coding encodes the same using three 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 four compression patterns using each of $(3,2)_8$ and $(3,1)_8$ expansion coding. However, if more than four compression patterns are encoded using one type of expansion coding (e.g., BΔI-based CompEx++ coding, as shown in Table III), then additional prefix bits become necessary.

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 $i = 0, 1, \dots, (n-1)$. The i^{th} pattern P_i has a compression ratio of r_i ($0 \leq r_i \leq 1$) and matches the w_i ($0 \leq w_i \leq 1$) fraction of the whole write traffic. Such a compression scheme would reduce bit writes to $\sum_{i=0}^{n-1} r_i w_i$ (≤ 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 to Section 3.2 for details) low energy states (the set of these low energy states is defined as S_i) out of the eight states for TLC RRAM. Therefore, the energy reduction from encoding a TLC using $(L, r_i.L)_8$ expansion code is $\sum_{j \in S_i} e_j / \sum_{j=0}^7 e_j$, 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 expressions for both pattern-based compression and expansion coding, 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 preceding 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, 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.

3.5. CompEx/CompEx++ Coding, ECC, and Encryption

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 encryption in CompEx/CompEx++ coding.

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 EDAC techniques such as ECC [Awasthi et al. 2012] and ECP [Schechter et al. 2010]. Since a write 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 I and II, we observe that the worst-case compressed data size for FPC-based (BAI-based) CompEx/CompEx++ coding is 35 (292) bits, which corresponds to 18 (146) TLCs out of the available 22 (171) TLCs. Therefore, FPC-based (BAI-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 Figure 9. The 64-bit data and the 8-bit ECC require $\lceil 64/3 \rceil + \lceil 8/3 \rceil = 25$ TLCs using conventional binary encoding, as illustrated in the figure. If the data is compressible, then the compressed data field requires at most $\lceil 35/2 \rceil = 18$ TLCs, whereas the 8-bit ECC requires $\lceil 8/2 \rceil = 4$ TLCs using $(3,2)_8$ expansion coding (i.e., one additional TLC in comparison to conventional binary encoding). However, the additional TLC required to encode the 8-bit ECC can be repurposed

Fig. 9. Illustration of ECC in CompEx/CompEx++ coding. The key takeaway is that CompEx/CompEx++ coding can preserve its energy and latency benefits of in the presence of ECC for no additional overhead.

from extra TLCs remaining unused in the data field after encoding using CompEx/CompEx++ coding, as illustrated in the figure. 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 the 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 Figure 9. Therefore, FPC-based (BAI-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.

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. 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 rewritten to memory [Awasthi et al. 2012]. 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. However, CompEx/CompEx++ coding has the potential to reduce performance overhead due to periodic scrub for no overhead. CompEx/CompEx++ coding encodes about 46% to 60% of the data residing in the memory using $(3,2)_8/(3,1)_8$ expansion coding. The fraction of the memory encoded using CompEx/CompEx++ coding enjoys superior error correction strength⁴ due to stronger ECC, which improves its resilience toward 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 Figure 10. Note that strengthening 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 [Chen et al. 2013; Palframan et al. 2015]. 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 theoretical framework for evaluating CompEx++ ECC. Consider a k -bit data line and an m -bit EDAC field, which together

⁴FPC-based CompEx/CompEx++ coding extends existing error correction capability (SECDED) to double error correction per word. Similarly, BAI-based CompEx/CompEx++ coding enables correction of three additional errors per cache line over the existing error correction capability.

Fig. 10. NVM scrub to mitigate errors due to resistance drift. The key takeaway is that CompEx/CompEx++ coding increases useful time and reduces downtime.

Table IV. Scrub Intervals for Different ECC Schemes and Their Corresponding Memory Area Overheads

ECC Scheme	Scrub Interval (s)	Scrub Overhead Improvement	ECC Overhead
Binary encoded Hamming code	2	1×	12.5%
Binary encoded BCH	5.92	2.96×	12.5%
CompEx++ECC Hamming code	2.18	1.09×	12.5%
CompEx++ECC BCH	8.33	4.16×	12.5%
Free ECC [Chen et al. 2013]	2	1×	0%
COP [Palframan et al. 2015]	2	1×	0%

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 [Awasthi et al. 2012], 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:

$$\text{UBER}(t) = 1 - \sum_{i=0}^{n_e} \binom{n}{i} p(t)^i (1 - p(t))^{(n-i)}. \quad (1)$$

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, $\text{UBER}(t)$ 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 its 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, and so forth. Thus, the upper limit on $\text{UBER}(t)$, $\text{UBER}_{\max}$, is an important system-level design parameter ($\text{UBER}_{\max} = 10^{-10}$ in this work), which constrains the maximum time between successive scrubs (i.e., $t_{\text{scrub}} = \text{UBER}^{-1}(\text{UBER}_{\max})$).

Evaluation. In the rest of this section, we compare and contrast the scrub intervals across various error correction schemes for TLC PCM [Seong et al. 2013; Stanisavljevic et al. 2016] using Equation (1); this is tabulated in Table IV. Although the rest of this work 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) single error correction and double error detection (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—that can correct up to six soft errors in a cache line; this corresponds to a scrub interval of 5.92s, which is an improvement of 2.96× 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/CompEx++ECC Hamming code can improve the scrub interval by $1.09\times (= (2 + 2.36)/2)$. Fourth, we similarly evaluate CompEx/CompEx++ECC BCH code, which can correct up to nine soft errors in a cache line as opposed to six 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 [Chen et al. 2013] and Compress and Protect (COP) [Palframan et al. 2015], which are the state-of-the-art compression-based ECC schemes. Both Free ECC and COP integrate 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 system performance and availability.

3.5.3. CompEx/CompEx++ Coding and Encryption. NVM nonvolatility has emerged as a serious data security concern [Chhabra and Solihin 2011; Young et al. 2015; Kong and Zhou 2010]—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 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, 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 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.

4. METHODOLOGY

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

Table V. (a) Configuration of the Evaluation Architecture (b) Composite Workloads Comprising Four Benchmarks from the SPEC CPU2006 Benchmark Suite (Derived from Ham et al. [2013]) for Full-System Evaluation of CompEx/CompEx++ Coding

Parameter	Attributes	Latency	Workload	Constituent Constituent	WPKI	MPKI
Core	4 cores, dual-issue out of order	—	WD ₁	leslie3d, leslie3d, mcf, mcf	4.61	9.66
L1 (instruction) cache	32kB, 2 way	2ns	WD ₂	lbm, leslie3d, libquantum, mcf	6.01	12.55
L1 (data) cache	32kB, 2 way	2ns	WD ₃	lbm, lbm, libquantum, libquantum	8.60	17.57
L2 cache	256kB, private, 8 way	5ns	WD ₄	bwaves, leslie3d, omentpp, sphinx3	3.68	7.95
L3 cache	8MB, shared, 16 way	20ns	WD ₅	GemsFDTD, libquantum, milc, zeusmp	3.72	7.93
Cache line size	64 bytes	—	WD ₆	GemsFDTD, libquantum, milc, milc	5.00	10.44
Main memory	16GB, 8 banks, 1 channel (DRAMSim2), 1GHz	160ns (average)	WD ₇	bzip, libquantum, milc, omentpp	7.41	15.32
			WD ₈	cactusAMD, gcc, gobmk, zeusmp	4.48	10.07
			WD ₉	astar, gobmk, hmmer, soplex	5.01	10.80

(a)

(b)

Note: 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 V(a) using MARSS.

4.1. Full-System Simulation

CompEx++ coding is evaluated using full-system simulations of a system that integrates TLC RRAM memory using MARSS [Patel et al. 2011], a full-system multicore simulator, and DRAMSim2 [Rosenfeld et al. 2011], a cycle-accurate main memory simulator.

MARSS uses x86 core models from PTLSim [Yourst 2007], a cycle-accurate x86 microarchitecture simulator, and plugs it into QEMU [Bellard 2005], 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 here, 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 four-core out-of-order system running at 3GHz. Each core has its own L1 cache with two 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 V(a).

DRAMSim2 setup. For accurate timing simulation, we modified the DDR timing parameters along the lines of Lee et al. [2009] for substituting DRAM with NVM. We use TLC RRAM with latency parameters extracted and summarized in Niu et al. [2013].

Workloads. We evaluate CompEx/CompEx++ coding using the SPEC CPU2006 [SPEC 2006] benchmark suite, which reflects a variety of integer and floating-point workloads used by modern computing systems. To evaluate real-world usage, we use nine composite workloads, with each workload containing four SPEC CPU2006 benchmarks. These composite workloads are derived from Ham et al. [2013], where benchmarks are selectively picked due to their memory-intensive nature. Table V(b) 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.

4.2. Trace-Driven Simulation

For running deep, multibillion-instruction simulations, CompEx/CompEx++ coding is evaluated using both an in-house trace-driven simulator for evaluating the memory

Table VI. Comparison of Memory Array Write Energy, Latency, and Overhead Normalized to Classical Binary Coding for the Nine Schemes Considered in this Article

Encoding	Total Energy	Dynamic Energy	Dynamic Latency	Overhead
Binary coding	100%	100%	100%	0%
Dgien et al. [2014] extended to 64-bit FPC	96%	79%	96%	0%
B Δ I compression	57%	31%	98%	0%
(3,2) ₈ expansion coding	33%	49%	44%	50%
(6,5) ₈ expansion coding	79%	75%	71 %	20%
FPC-based CompEx coding	80.8%	67%	78.2%	0%
B Δ I-based CompEx coding	43%	24%	75%	0%
FPC-based CompEx++ coding	80.7%	67%	77.7%	0%
B Δ I-based CompEx++ coding	39%	22%	70.4%	0%

Note: All cases use classical read-modify-write (DCW) [Yang et al. 2010] to update only the modified cells in the NVM array.

array-level dynamic energy and NVMain [Poremba and Xie 2012]—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/CompEx++ coding and also configured NVMain to simulate an architecture equivalent to that in Table V(a). The traces are generated from the SPEC CPU2006 [SPEC 2006] benchmark suite using the Intel Pin binary instrumentation tool [Luk et al. 2005] on a machine running a 3.3GHz Intel Core i7 CPU. Note that we also use the Gem5 system simulator [Binkert et al. 2011] to validate these results using an equivalent architecture; the results are consistent with what is reported here and not discussed for brevity. 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 5×10^5 memory writes to ignore the write accesses from program initialization; they are then run until 4×10^6 memory write operations (equivalent to about 4 billion instructions on average) have been recorded or until the program terminates.

5. 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 B Δ I), expansion coding ((3,2)₈ and (6,5)₈), CompEx coding (FPC-based and B Δ I-based CompEx coding, using (3,2)₈ expansion coding), and CompEx++ coding (FPC based and B Δ I 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.

5.1. Memory Energy/Latency

Summary. Table VI summarizes and compares the total module energy, memory array dynamic energy, latency, and overhead for the nine encoding techniques considered in this article. In summary, FPC-based and B Δ I-based CompEx coding reduce the memory array write energy by 33% and 76% and 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 Δ 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 Δ I-based CompEx coding, respectively.

Fig. 11. Dynamic energy at the memory array level (a) and total memory module energy results (using NVMain [Poremba and Xie 2012]) (b) for FPC, (3,2)_s expansion coding, (6,5)_s expansion coding, FPC-based CompEx coding, and BAI-based CompEx coding, normalized to classical binary coding. FPC-based and BAI-based CompEx coding reduce dynamic energy (total energy) by 33% (18%) and 76% (57%) in comparison to binary encoding. As expected (in Table II), the dynamic energy performance of BAI-based CompEx coding is better than FPC-based CompEx coding. Furthermore, FPC-based CompEx++ coding and BAI-based CompEx++ coding reduce 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.

Energy. Our simulation framework tracks all cell writes that occur from the beginning of program execution to compute the cumulative energy required. 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 [Poremba and Xie 2012]. Figure 11(a) and (b) show the memory array dynamic energy and the total memory module energy, respectively, for FPC, BAI, (3,2)_s and (6,5)_s expansion coding, and FPC-based and BAI-based CompEx coding, normalized to binary coding. The last entries of Figure 11 represent the geometric mean (GM) of the energy reduction across all benchmarks, which is equivalent to simulating all of these benchmarks for the same execution time. Note that all cases use classical read-modify-write (DCW) [Yang et al. 2010] 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, whereas BAI-based CompEx coding reduces energy by 76% (57%) and 16% (25%) in comparison to binary coding and BAI, respectively. Additionally, (3,2)_s and (6,5)_s 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 BAI-based CompEx++ coding extend 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 in energy numbers is because of the low percentage of pattern match

Fig. 12. Latency results for FPC, $(3,2)_8$ expansion coding, $(6,5)_8$ expansion coding FPC-based CompEx coding, and BAI-based CompEx coding, normalized to binary coding. FPC-based and BAI-based CompEx coding reduce latency by 22% and 25% in comparison to binary encoding. As expected (in Table II), the latency performance of FPC-based CompEx coding and BAI-based CompEx coding are comparable to each other. Furthermore, FPC-based and BAI-based CompEx++ coding extend the latency improvements by 0.6% and 6.7%, respectively, over CompEx coding.

for FPC patterns that can leverage $(3,1)_8$ expansion coding. However, CompEx++ coding can potentially have a larger influence in comparison to CompEx coding for compression schemes that offer higher $(3,1)_8$ expansion coding possibilities, as shown by BAI-based CompEx++ coding.

Latency. We evaluate the impact on memory array write latency due to CompEx/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 cells, the latency for writing a word depends on the cell that requires the longest latency. Therefore, since CompEx/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. Figure 12 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 BAI-based CompEx coding reduces overall write latency by about 25% and 26% in comparison to binary coding and BAI, respectively. Furthermore, CompEx++ coding extends the latency benefits of CompEx coding by 0.6% and 6.7% over FPC-based and BAI-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.

5.2. Full-System Evaluation

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

Summary. Table VII summarizes and compares IPC and bandwidth for classical binary encoding, $(3,2)_8$ expansion coding, FPC-based CompEx coding, BAI-based CompEx coding, and BAI-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. In summary, $(3,2)_8$ expansion coding, FPC-based and BAI-based CompEx coding,

Table VII. IPC and Bandwidth for FPC-Based CompEx Coding, BΔI-Based CompEx Coding, and BΔI-Based CompEx++ Coding Normalized to Classical Binary Encoding

Encoding	IPC	Bandwidth
Binary coding	100%	100%
$(3,2)_8$ expansion coding	134%	176%
FPC-based CompEx coding	106.3%	111.1%
BΔI-based CompEx coding	105.1%	112.6%
BΔI-based CompEx++ coding	110.6%	119.9%

Note: All cases use DCW [Yang et al. 2010] to update only the modified cells in the NVM array.

Fig. 13. 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 represents the HM 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.

and BΔI-based CompEx++ coding improve 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 classical binary encoding.

Instructions per cycle. To evaluate the impact of CompEx/CompEx++ coding on IPC, we use a full-system simulator based on MARSS [Patel et al. 2011] and DRAM-Sim2 [Rosenfeld et al. 2011]. The simulation setup is described in detail in Section 4. We simulate nine composite workloads from Table V(b). Figure 13 shows the IPC for each benchmark in each workload. The last set of bars in each workload in Figure 13 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 (see Chapter 1 of Hennessy and Patterson [2011])). The last set of bars in Figure 13 represents the harmonic mean (HM) of the IPCs of all workloads (computing the HM is equivalent to running all 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 WD₃ (highest MPKI) and WD₅ (lowest MPKI). As expected, the high cache miss rate of WD₃ lowers the IPC in comparison to WD₅, which has a lower cache miss rate. To understand the dependence of IPC on WPKI, let us consider the workloads WD₇ (highest WPKI) and WD₄ (lowest WPKI).

Fig. 14. Comparison of main memory bandwidth (normalized to binary coding) obtained using full-system simulation (MARSS+DRAMSim2). The overall bandwidth improvement is 11.1%, 12.6%, and 19.9% for FPC-based CompEx coding, BΔI-based CompEx coding, and BΔI-based CompEx++ coding, respectively. Furthermore, the bandwidth improvement of (3,2)₈ expansion coding, which is an upper bound for CompEx coding, is 76%.

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 preceding argument, show that FPC (BΔI)-based CompEx coding improves the IPC by 17.5% (14.8%) for WD₇ (high WPKI) in comparison to 5.5% (1%) for WD₄ (low WPKI). 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 gemsFDTD (WD₆) or omnetpp (WD₄); our simulations are in excellent agreement with the preceding argument. Finally, our simulations show that the overall IPC improvement using FPC-based CompEx coding, 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.⁵ In addition, (3,2)₈ expansion coding, an upper bound for CompEx coding, improves IPC by 34% over classical binary encoding.

Memory bandwidth. Figure 14 shows the main memory bandwidth across nine composite workloads for the baseline and CompEx/CompEx++ coding in comparison to classical binary encoding. In the figure, the bars represent FPC-based CompEx coding and BΔI-based CompEx/CompEx++ coding normalized to the baseline (binary encoding with DCW). The last set of bars in the figure represents the 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

⁵We see a good correlation between the trace-based latency results and the full-system IPC results. Benchmarks like milc and libquantum, which 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. However, 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.

$(3,2)_8$ expansion coding, which is an upper bound for CompEx coding, over classical binary encoding is 76%.⁶

Lifetime. In this article, we theoretically evaluate the lifetime gains of CompEx/CompEx++ coding using TLC RRAM as an example. Chen et al. [2011] discuss 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 number of programming cycles (SET/RESET) that it can endure before the cell becomes dysfunctional. Therefore, the lifetime of MLC/TLC RRAM is lower in comparison to SLC RRAM, since programming an MLC/TLC RRAM requires a higher number of P&V cycles in comparison to SLC RRAM [Xu et al. 2013]. 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 of CompEx/CompEx++ coding for TLC RRAM—along the lines of Qureshi et al. [2010], Xu et al. [2013], and Jiang et al. [2012a]—show that CompEx/CompEx++ coding increases the lifetime by $1.8\times$ over classical binary coding.

6. RELATED WORK

MLC/TLC NVMS definitely benefit from the broad set of solutions developed to improve energy, latency, and lifetime of SLC NVMS [Zhao et al. 2014]. However, SLC-based solutions do not address the energy/latency problems of MLC/TLC NVMS that are primarily due to the iterative nature of the 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 [Yang et al. 2000; Alameldeen and Wood 2004; Wang et al. 2011].

Memory compression. Increasing cache capacity by various compression techniques that leverage different localities have been studied [Sardashti and Wood 2013; Ahn et al. 2001; Chen et al. 2010; Yang et al. 2000; Yang and Gupta 2002]. On similar lines, there are solutions that compress main memory traffic for the benefits of bandwidth, power, and capacity [Mahapatra et al. 2005; Yim et al. 2004; Dgien et al. 2014; Ekman and Stenstrom 2005; Arelakis and Stenstrom 2014; Shafee et al. 2014]. In the context of MLC/TLC NVMS, compressing main memory traffic yields fewer cell writes per write access, thus lowering energy. Whereas some solutions reduce energy/latency by excluding undesirable states for additional memory area (summarized in the following), CompEx coding, to the best of our knowledge, is the first work to explicitly investigate the trade-offs between the area recovered from compression and solutions that exclude undesirable MLC/TLC states.

Excluding undesirable states. Arjomand et al. [2011] and Dong and Xie [2011] propose circuit and architectural changes to dynamically configure MLC PCM cells as either MLC or SLC for latency benefits. In contrast, Jiang et al. [2012b] excludes the hard-to-reset cells that require high programming current and recovers the lost data using ECC, thereby reducing power consumption and improving lifetime. Extending the observations of Jiang et al. [2012b], Elastic RESET (ER) [Jiang et al.

⁶Note that the improvement in bandwidth is not due to compression of a cache line/word to a smaller size—data transfer between the memory controller and the NVM module is always in the uncompressed format in CompEx/CompEx++ coding architecture; the bandwidth improvement is primarily attributable to the faster cell writes due to expansion coding, which avoids high latency states of MLC/TLC NVM cells. Thus, the data transfer of a 64-byte cache line always requires eight 64-bit word transfers (in a 64-bit bus architecture) irrespective of whether the data is compressible or incompressible.

2012a] proposes data coding that eliminates the undesirable terminal RESET state (high programming current) and uses only two or three of the four states of a cell to realize lifetime/power/latency improvements. Similar to ER, hybrid MLC/SLC [Lee et al. 2012] proposes to opportunistically use PCM cells as either MLC or SLC for energy and latency benefits. Independently, for MLC PCM, Wang et al. [2011] propose 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, Wang et al. [2011] require online computation and storage of the most frequent patterns for every memory line at runtime, incurring compute, memory, and logic overhead. For MLC PCM, Yoon et al. [2013] propose data coding that eliminates one of the intermediate resistance states; this improves cell retention but incurs memory overhead. For TLC RRAM, Niu et al. [2013] propose data coding that uses six out of eight TLC RRAM states to improve latency and energy by eliminating the use of intermediate resistance states [Wang et al. 2011; Niu et al. 2013]. 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. Xu et al. [2015] observed a superlinear 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) [Jiang et al. 2012c] 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, as it depends on the compression technique used and the energy/latency profile of the NVM cell. In practice, it is necessary to balance dynamic trade-offs 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.

7. CONCLUSIONS

This article described CompEx coding—a low overhead, dynamic trade-off 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 work, we evaluated CompEx coding using both FPC and BAI compression. We also proposed 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. 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 $1.8\times/1.8\times$ improvement in lifetime over classical binary coding using DCW. 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.

REFERENCES

Edward Ahn, Seung-Moon Yoo, and Sung-Mo Steve Kang. 2001. Effective algorithms for cache-level compression. In *Proceedings of the 2001 Great Lakes Symposium on VLSI*.

- A. Alameldeen and D. Wood. 2004. *Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches*. Technical Report. University of Wisconsin–Madison.
- Angelos Arelakis and Per Stenstrom. 2014. SC2: A statistical compression cache scheme. In *Proceedings of the 2014 International Symposium on Computer Architecture*. 145–156.
- M. Arjomand, A. Jadidi, A. Shafiee, and H. Sarbazi-Azad. 2011. A morphable phase change memory architecture considering frequent zero values. In *Proceedings of the 2011 International Conference on Computer Design*.
- Manu Awasthi, Manjunath Shevgoor, Kshitij Sudan, Bipin Rajendran, Rajeev Balasubramonian, and Viji Srinivasan. 2012. Efficient scrub mechanisms for error-prone emerging memories. In *Proceedings of the 2012 International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA.
- I. G. Baek, M. S. Lee, S. Seo, M.-J. Lee, D. H. Seo, D.-S. Suh, J. C. Park, et al. 2004. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Proceedings of the 2004 International Electron Devices Meeting*.
- Ferdinando Bedeschi, Rich Fackenthal, Claudio Resta, Enzo Michele Donze, Meenatchi Jagasivamani, Egidio Cassiodoro Buda, Fabio Pellizzer, et al. 2009. A bipolar-selected phase change memory featuring multi-level cell storage. *IEEE Journal of Solid-State Circuits* 44, 1, 217–227.
- Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2, 1–7.
- B. Chen, Y. Lu, B. Gao, Y. H. Fu, F. F. Zhang, P. Huang, Y. S. Chen, et al. 2011. Physical mechanisms of endurance degradation in TMO-RRAM. In *Proceedings of the 2011 International Electron Devices Meeting*.
- L. Chen, Y. Cao, and Z. Zhang. 2013. Free ECC: An efficient error protection for compressed last-level caches. In *Proceedings of the 2013 International Conference on Computer Design*. 278–285.
- X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. 2010. C-Pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 18, 8, 1196–1208.
- Siddhartha Chhabra and Yan Solihin. 2011. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *Proceedings of the 2011 International Symposium on Computer Architecture*.
- S. Cho and H. Lee. 2009. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 2009 International Symposium on Microarchitecture*.
- Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, et al. 2012. A 20nm 1.8v 8gb PRAM with 40mb/s program bandwidth. In *Proceedings of the 2012 International Solid-State Circuits Conference*.
- D. Costello and Shu Lin. 2004. *Error Control Coding*. Pearson Higher Education.
- David B. Dgien, Poovaiah M. Palangappa, Nathan A. Hunter, Jiayin Li, and Kartik Mohanram. 2014. Compression architecture for bit-write reduction in non-volatile memory technologies. In *Proceedings of the 2014 International Symposium on Nanoscale Architectures*.
- Xiangyu Dong and Yuan Xie. 2011. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. In *Proceedings of the 2011 Asia and South Pacific Design Automation Conference*.
- M. Ekman and P. Stenstrom. 2005. A robust main-memory compression scheme. In *Proceedings of the 2005 International Symposium on Computer Architecture*.
- Tae Jun Ham, Bharath K. Chelepalli, Neng Xue, and Benjamin C. Lee. 2013. Disintegrated control for energy-efficient and heterogeneous memory systems. In *Proceedings of the 2013 International Symposium on High Performance Computer Architecture*.
- J. L. Hennessy and D. A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann.
- ITRS. 2011. International Technology Roadmap for Semiconductors. Available at <http://www.itrs.net/2011-itrs.html>
- Lei Jiang, Youtao Zhang, Bruce R. Childers, and Jun Yang. 2012b. FPB: Fine-grained power budgeting to improve write throughput of multi-level cell phase change memory. In *Proceedings of the 2012 International Symposium on Microarchitecture*.
- Lei Jiang, Youtao Zhang, and Jun Yang. 2012a. ER: Elastic RESET for low power and long endurance MLC. In *Proceedings of the 2012 International Symposium on Low Power Electronics and Design*.
- Lei Jiang, Bo Zhao, Youtao Zhang, Jun Yang, and Bruce R. Childers. 2012c. Improving write operations in MLC phase change memory. In *Proceedings of the 2012 International Symposium on High Performance Computer Architecture*.

- Lei Jiang, Bo Zhao, Youtao Zhang, Jun Yang, and Bruce R. Childers. 2012d. Improving write operations in MLC phase change memory. In *Proceedings of the 2012 International Symposium on High Performance Computer Architecture*.
- A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das. 2012. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Proceedings of the 2012 Design Automation Conference*.
- M. Kang, T. J. Park, Y. W. Kwon, D. H. Ahn, Y. S. Kang, H. Jeong, S. J. Ahn, et al. 2011. PRAM cell technology and characterization in 20nm node size. In *Proceedings of the 2011 International Electron Devices Meeting*.
- Sangbeom Kang, Woo Yeong Cho, Beak-Hyung Cho, Kwang-Jin Lee, Chang-Soo Lee, Hyung-Rok Oh, Byung-Gil Choi, et al. 2007. A 0.1- μm 1.8-V 256-Mb phase-change random access memory (PRAM) with 66-MHz synchronous burst-read operation. *IEEE Journal of Solid-State Circuits* 42, 1, 210–218.
- Y. Kim, S. Yoo, and S. Lee. 2012. Write performance improvement by hiding R drift latency in phase change RAM. In *Proceedings of the 2012 Design Automation Conference*.
- Jingei Kong and Huiyang Zhou. 2010. Improving privacy and lifetime of PCM-based main memory. In *Proceedings of the 2010 International Conference on Dependable Systems and Networks*.
- Suknam Kwon, Sungjoo Yoo, Sunggu Lee, and Jinpyo Park. 2012. Optimizing video application design for phase-change RAM-based main memory. *IEEE Transactions Very Large Scale Integration (VLSI) Systems* 20, 11, 2011–2019.
- Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 2009 International Symposium on Computer Architecture*.
- H. G. Lee, S. Baek, J. Kim, and C. Nicopoulos. 2012. A compression-based hybrid MLC/SLC management technique for phase-change memory systems. In *Proceedings of the 2012 IEEE Annual Symposium on VLSI*.
- Kwang-Jin Lee, Beak-Hyung Cho, Woo-Yeong Cho, Sangbeom Kang, Byung-Gil Choi, Hyung-Rok Oh, Chang-Soo Lee, et al. 2008. A 90 nm 1.8 V 512 Mb diode-switch PRAM with 266 MB/s read throughput. *IEEE Journal of Solid-State Circuits* 43, 1, 150–162.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, et al. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation*.
- Nihar R. Mahapatra, Jiangjiang Liu, Krishnan Sundaresan, Srinivas Dangeti, and Balakrishna V. Venkatao. 2005. A limit study on the potential of compression for improving memory system performance, power consumption, and cost. *Journal of Instruction-Level Parallelism* 7, 1–37.
- John D. McCalpin. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter* 12, 19–25.
- T. Nirschl, J. B. Philipp, T. D. Happ, G. W. Burr, B. Rajendran, M. H. Lee, A. Schrott, et al. 2007. Write strategies for 2 and 4-bit multi-level phase-change memory. In *Proceedings of the 2007 International Electron Devices Meeting*.
- Dimin Niu, Qiaosha Zou, Cong Xu, and Yuan Xie. 2013. Low power multi-level-cell resistive memory design with incomplete data mapping. In *Proceedings of the 2013 International Conference on Computer Design*.
- Poovaiah M. Palangappa, Jiayin Li, and Kartik Mohanram. 2015. WOM-Code solutions for low latency and high endurance in phase change memory. *IEEE Transactions on Computers* 65, 4, 1025–1040. DOI : <http://dx.doi.org/10.1109/TC.2015.2506555>
- Poovaiah M. Palangappa and Kartik Mohanram. 2015. Flip-mirror-rotate: An architecture for bit-write reduction and wear leveling in non-volatile memories. In *Proceedings of the 2015 Great Lakes Symposium on VLSI*.
- Poovaiah M. Palangappa and Kartik Mohanram. 2016. CompEx: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVM. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*. 90–101.
- D. J. Palframan, N. S. Kim, and M. H. Lipasti. 2015. COP: To compress and protect main memory. In *Proceedings of the 2015 International Symposium on Computer Architecture*. 682–693.
- Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 2011 Design Automation Conference*.
- Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 2012 International Conference on Parallel Architectures and Compilation Techniques*.

- Matt Poremba and Yuan Xie. 2012. NVMain: An architectural-level main memory simulator for emerging non-volatile memories. In *Proceedings of the 2012 IEEE Computer Society Annual Symposium on VLSI*. 392–397.
- M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras. 2012. PreSET: Improving performance of phase change memories by exploiting asymmetry in write time. In *Proceedings of the 2012 International Symposium on Computer Architecture*.
- M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of the 2010 International Symposium on High Performance Computer Architecture*.
- M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montano, and J. P. Karidis. 2010. Morphable memory system: A robust architecture for exploiting multi-level phase change memories. *ACM SIGARCH Computer Architecture News* 38, 3, 153–162.
- M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. 2009. Enhancing lifetime and security of phase change memories via start-gap wear leveling. In *Proceedings of the 2009 International Symposium on Microarchitecture*.
- P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1, 16–19.
- Somayeh Sardashti and David A. Wood. 2013. Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching. In *Proceedings of the 2013 International Symposium on Microarchitecture*.
- Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. 2010. Use ECP, not ECC, for hard failures in resistive memories. In *Proceedings of the 2010 International Symposium on Computer Architecture*.
- N. Seong, S. Yeo, and H.-H. S. Lee. 2013. Tri-level-cell phase change memory: Toward an efficient and reliable memory system. In *Proceedings of the 2013 International Symposium on Computer Architecture*.
- N. H. Seong, D. H. Woo, and H.-H. S. Lee. 2010. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 2010 International Symposium on Computer Architecture*.
- A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis. 2014. MemZip: Exploring unconventional benefits from memory compression. In *Proceedings of the 2014 International Symposium on High Performance Computer Architecture*.
- SPEC. 2006. SPEC CPU2006. Retrieved February 23, 2017, from <https://www.spec.org/cpu2006/>.
- M. Stanisavljevic, H. Pozidis, A. Athmanathan, N. Papandreou, T. Mittelholzer, and E. Eleftheriou. 2016. Demonstration of reliable triple-level-cell (TLC) phase-change memory. In *Proceedings of the 2016 International Memory Workshop*. 1–4.
- Jue Wang, Xiangyu Dong, Guangyu Sun, Dimin Niu, and Yuan Xie. 2011. Energy-efficient multi-level cell phase-change memory system with data encoding. In *Proceedings of the 2011 International Conference on Computer Design*.
- H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai. 2012. Metal oxide RRAM. *Proceedings of the IEEE* 100, 6, 1951–1970.
- Gang Wu, Huxing Zhang, Yaozu Dong, and Jingtong Hu. 2012. CAR: Securing PCM main memory system with cache address remapping. In *Proceedings of the 2012 International Conference on Parallel and Distributed Systems*.
- Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings of the 2015 International Symposium on High Performance Computer Architecture*.
- Cong Xu, Dimin Niu, Naveen Muralimanohar, Norman P. Jouppi, and Yuan Xie. 2013. Understanding the trade-offs in multi-level cell ReRAM memory design. In *Proceedings of the 2013 Design Automation Conference*.
- B. Yang, J. Lee, J. Kim, J. Cho, S. Lee, and B. Yu. 2010. A low power phase change random access memory using a data-comparison write scheme. In *Proceedings of the 2010 International Symposium on Circuits and Systems*.
- Jun Yang and Rajiv Gupta. 2002. Energy efficient frequent value data cache design. In *Proceedings of the 2002 International Symposium on Microarchitecture*.
- Jun Yang, Youtao Zhang, and Rajiv Gupta. 2000. Frequent value compression in data caches. In *Proceedings of the International Symposium on Microarchitecture*.
- Keun Soo Yim, Jihong Kim, and Kern Koh. 2004. Performance analysis of on-chip cache and main memory compression systems for high-end parallel computers. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*.

- D. H. Yoon, J. Chang, R. S. Schreiber, and N. P. Jouppi. 2013. Practical nonvolatile multilevel-cell phase change memory. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage, and Analysis*.
- Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. 2015. DEUCE: Write-efficient encryption for non-volatile memories. In *Proceedings of the 2015 International Conference on Architectural Support for Programming Languages and Operating Systems*.
- Matt T. Yourst. 2007. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software*.
- J. Yue and Y. Zhu. 2012. Making write less blocking for read accesses in phase change memory. In *Proceedings of the 2012 International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*.
- J. Yue and Y. Zhu. 2013. Accelerating write by exploiting PCM asymmetries. In *Proceedings of the 2013 International Symposium on High Performance Computer Architecture*.
- Mengying Zhao, Lei Jiang, Youtao Zhang, and Chun Jason Xue. 2014. SLC-enabled wear leveling for MLC PCM considering process variation. In *Proceedings of the 2014 Design Automation Conference*.

Received July 2016; revised January 2017; accepted January 2017