Exploring the Potential for Collaborative Data Compression and Hard-Error Tolerance in PCM Memories

Amin Jadidi†, Mohammad Arjomand‡, Mohammad Khavari Tavana‡, David R. Kaeli†, Mahmut T. Kandemir†, Chita R. Das‡

†School of Electrical Engineering and Computer Science, Pennsylvania State University, USA
‡Department of Electrical and Computer Engineering, Northeastern University, USA
Email: {axj945, mxa51, kandemir, das} @cse.psu.edu, {mkhavaritavana, kaeli} @ece.neu.edu

Abstract—Limited write endurance is the main obstacle standing in the way of using phase change memory (PCM) in future computing systems. While several wear-leveling and hard-error tolerant techniques have been proposed for improving PCM lifetime, most of these approaches assume that the underlying memory uses a very simple write traffic reduction scheme (e.g., buffering, differential writes). In particular, most PCM prototypes/chips are equipped with an embedded circuit to support differential writes (DW) — on a write, only the bits that differ between the old and new data are updated. With DW, the bit-pattern of updates in a memory block is usually random, which limits the opportunity to exploit the resulting bit pattern for lifetime enhancement at an architecture level (e.g., using techniques such as wear-leveling and hard-error tolerance). This paper focuses on this inefficiency and proposes a solution based on data compression. Employing compression can improve the lifetime of the PCM memory. Using state-of-the-art compression schemes, the size of the compressed data is usually much smaller than the original data written back to memory from the last-level cache on an eviction.

By storing data in a compressed format in the target memory block, first, we limit the number of bit flips to fewer memory cells, enabling more efficient intra-line wear-leveling and error recovery; and second, the unused bits in the memory block can be reused as replacements for faulty bits given the reduced size of the (compressed) data. It can also happen that for a portion of the memory blocks, the resulting compressed data is not very small. This can be due to increased data entropy introduced by compression, where the total number of bit flips will be increased over the baseline system. In this paper, we present an approach that provides collaborative operation of data compression, differential writes, wear-leveling and hard-error tolerant techniques targeting PCM memories.

We propose approaches that reap the maximum benefits from compression, while also enjoying the benefits of techniques that reduce the number of high-entropy writes. Using an approach that combines different solutions, our mechanism tolerates 2.9× more cell failures per memory line and achieves a 4.3× increase in PCM memory lifetime, relative to our baseline state-of-the-art PCM DIMM memory.

Keywords—Resistive memory; hard-error tolerance; compression; phase change memory;

I. INTRODUCTION

During the last three decades, DRAM has been used as the dominant technology in main memory for computing systems. However, enter the deep nanometer era, where DRAM faces serious scalability and power consumption problems. These are well-documented problems with DRAM [1], [2] that have attracted several solutions, ranging from device-level [3] to architectural-level [4] solutions. Researchers from both academia and industry have suggested that phase change memory (PCM) is one of the leading technologies to be used as a scalable alternative for DRAM in future systems [5], [6]. PCM has some advantages and disadvantages over DRAM. It has the advantages of lower power dissipation, higher density and better scalability. The disadvantages include longer write latency and higher write energy, as well as limited write endurance. While PCM write latency and energy decrease as cell sizes decrease, write endurance remains the main obstacle to adopt PCM in future commercial systems.

The solutions for the PCM endurance problem can be categorized into two groups: 1. hard-error postponement and 2. hard-error tolerant techniques. Postponement techniques can be further divided into two sub-categories: 1. write traffic reduction schemes and 2. wear-leveling schemes. In recent years, several wear-leveling schemes [7] and hard-error tolerant techniques [8]–[11] have been proposed for PCM memories. However, little attention has been paid to techniques for memory write reduction. Most previously proposed PCM memories use very simple schemes such as buffering [12] or differential writes (DW) [6] for this purpose. DW is a circuit-level mechanism supported by the PCM chips – each PCM chip has read-modify-write (RMW) logic [13] that reads old data from the target memory block and compares it with new values (bit by bit), in order to only write the bits that need to be updated. Although DW schemes significantly decrease the number of bit flips for every memory write-back, the resultant bit-level updates usually have a fairly random pattern (i.e., bit flips are randomly scattered over the entire memory block). Figure 1 illustrates this behavior, showing the number of bit-level updates from consecutive writes to a specific and randomly-chosen 64-byte memory block for the gobmk application – we can see that the update pattern is fairly random in nature.

Because of this behavior, designers usually look at DW as an orthogonal mechanism, combined with other tech-
Figure 1: Distribution of updated bits for consecutive writes to a specific and randomly-chosen 64-byte memory block for the gobmk application.

Consecutive Writes

<table>
<thead>
<tr>
<th>Bit Flips per Write</th>
<th>0</th>
<th>128</th>
<th>256</th>
<th>384</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consecutive Writes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Consecutive Writes

Techniques implemented at higher levels. We believe such a collaborative design between low-level and high-level PCM endurance enhancement techniques is missing. Further, a multi-level approach can also be very useful in future technologies, where PCM lifetime will become even more challenging (due to huge process variations and/or the use of multi-bit PCM memories, which have lower endurance). To this end, we suggest to exploit compression for decreasing the size of the written data.

Using compression, we limit the bit flips to a smaller window rather than the entire block. We refer to this window as the compression window throughout this paper. The compression window confines bit flips to a small number of bits, and its size is variable, depending on the compressibility of the written data. Exploiting compression, we hope to improve lifetime of each memory block from two perspectives.

1) When a bit fails, the memory controller has two options:
   (i) a large class of prior hard-error tolerant schemes (e.g., SAFER [9] of Aegis [11]) rely on partitioning, in which they try to partition the entire memory block such that each partition has at most one failed bit. Finding such partitions are easier as errors are limited to a smaller window in this condition, and (ii) If the hard-error technique in use cannot save the block, the remaining bits in the target block can be exploited as replacements for the failed bits. In other words, the memory controller can continue writing into the target block by sliding the compression window in order to have enough healthy cells for writing data.

2) In order to postpone fast wear-out in the bits within a compression window of a block, intra-line wear-leveling can be employed, thereby avoiding putting pressure on a limited number of bits, and hence, evenly distributing writes over the entire block. In such a design, wear-leveling within blocks can be implemented by sliding the compression window periodically. We will describe how the memory controller can achieve nearly perfect intra-line wear-leveling without using any dedicated write counters.

Then the memory controller can overcome the aforementioned limits of DW, using compression to extend PCM lifetimes by revisiting intra-line wear-leveling and hard-error tolerant techniques.

Blindly using compression in PCM memory may affect lifetime negatively. We generally expect that by reducing the size of the written data using compression (and after applying DW at chip-level), the number of bit flips decreases with respect to the baseline PCM. However, this is not always the case, as we will show in this paper. For 20% of memory writes, the number of bit flips increases, since compression increases data entropy (i.e., the chance of over-writing a ‘0’ with a ‘1’ or a ‘1’ with a ‘0’). The increase in the number of bit flips leads to both increased energy consumption and decreased lifetime of the PCM memory. Thus, the controller may decide not to compress the selected data if it leads to an increase in the number of bit flips. Nonetheless, as we will discuss, this is not easy to implement since compression is performed by the memory controller (i.e., residing on CPU chip), while the number of bit flips is determined after applying DW at the chip-level. Therefore, it is very costly to have the memory controller perform the compression in current PCM DIMM memory organizations and protocols.

Moreover, the compression-based design optimizations in PCM should take into account this limitation when trying to reduce the bit flips.

The main contributions of this paper are as follows:

- We begin by describing the limitations of widely-used DW schemes in PCM memories. We also discuss how compression can help us overcome these issues. We also qualitatively and quantitatively discuss the benefits and side effects of using compression for lifetime enhancement in PCM memories.
- We propose a novel PCM memory architecture, and corresponding changes at memory controller, that efficiently leverage compression in collaboration with hard-error tolerant and intra-line wear-leveling schemes. The cost of managing the compression of metadata and implementing intra-line wear-leveling in the proposed design is insignificant.
- Using a heuristic approach in our mechanism, we reduce the cost of increased bit flips due to compression, allowing the memory controller to predict which conditions will result in an increased number of bit flips.
- Using an extensive simulation-based study, the proposed design yields a 90% reduction in uncorrectable errors and a 20× increase in PCM memory lifetimes, relative to the baseline state-of-the-art PCM DIMM memory.

Finally note that our proposed design assumes that any prior compression algorithm (such as [14]–[17]), hard-error tolerant technique (such as [8]–[11]), or inter-line wear-leveling (such as [7]) can be used by the memory controller.
II. BACKGROUND AND RELATED WORK

A. PCM Basics and Baseline Organization

Figure 2a shows a PCM cell structure and its access circuits. PCM stores binary data in a chalcogenide material (i.e., Ge$_2$Sb$_2$Te$_5$ or GST for short), in either a high-resistance (RESET) or low-resistance (SET) state. Reading a PCM cell’s contents requires us to apply a low current for a short period of time to sense its resistivity. However, the write process will depend on the written value: a SET pulse is a low-amplitude and long-duration current, while a RESET pulse is a large and short-time current. In contrast to a SET operation, a RESET is fast, consumes more energy, and significantly contributes to PCM wear-out.

We assume a PCM main memory organization similar to a traditional DRAM as our baseline (i.e., the PCM has multiple channels, each connected to multiple memory modules, a Dual In-line Memory Module or DIMM). Figure 2b illustrates a DIMM-based PCM memory using the DDRx configuration. Each DIMM has multiple ranks, and each rank consists of a set of PCM chips, that together feed the data bus. In the common case, a rank in a PCM DIMM contains nine 8-bit chips (8 data bits of input/output on every clock edge), providing a 72-bit access. This organization is called an ECC-DIMM, where the ninth chip stores an Error Correcting Code (ECC) 2. A rank is partitioned into multiple banks, where different banks can process independent memory requests. As shown in Figure 2b, each bank is distributed across all chips in a rank. When the memory controller issues a request for a cache line, all PCM chips in a rank are activated, and each sub-bank contributes a portion of the requested block. 3 For example, if we assume a 64-byte cache line, and as each chip provides 8 bits of information on every clock edge (a total of 72 bits for both data and ECC), it takes a burst of 8 cycles to transfer a cache line.

1PCM can store either one bit (Single-Level Cell or SLC) or multiple bits (Multi-Level Cell or MLC) within each cell. Although the proposed approach can be applied to both, we assume an SLC PCM as our baseline. This is because MLC endurance (10$^{10}$–10$^{12}$) and performance is worse than that of SLC PCM, which complicates its use for main memory.

2Error correction capability is a must in PCM products, due to their high hard-error rate. All PCM-based reliability designs adhere to a 12.5% capacity overhead, the same as in ECC-DIMMs in DDRx DRAMs. While we describe our scheme for only ECC-based DIMMs, it can be used in any memory system with other protection mechanisms (e.g., chipkill [19]).

3In this paper we use the terms block and line interchangeably.

Note that an ECC-DIMM only provides additional storage for ECC, and on each memory request, the ECC information and the original data are sent to the memory controller on the processor die, where the actual error detection/correction takes place. This leaves the decision of which error protection mechanism to implement to the system designer.

B. Wear-out in PCM

Similar to other non-volatile memories, PCM has the problem of limited write endurance. This problem is primarily due to the heating and cooling process during write operations – after a finite number of writes, the cell loses its programmability and becomes “stuck” at either the SET or RESET state. Stuck-at SET and stuck-at RESET faults occur for different reasons [20]. Stuck-at SET happens because the GST material loses its crystalline quality over time. Experimental device-level studies have shown that a cell with a stuck-at SET fault can be recovered by applying a reverse electron field [20]. Stuck-at RESET, on the other hand, happens because the heating electrode detaches from the GST. Stuck-at RESET is unrecoverable [20], [21]. Prior studies report that SAR is the dominant failure mode in PCMs, especially in small-geometry cells where the contact between the heating electrode and GST becomes weak – according to a recent ITRS report [22], SLC PCM devices at a 9nm technology node will fail after around 10$^7$ SET-to-RESET transitions. However, our proposed technique is general, and we assume that either fault model can occur.

C. Prior Work on Improving PCM Lifetime

To cope with the limited endurance in PCM, prior studies take different approaches, working at different abstraction levels. We can categorize the relevant studies into two main groups: (i) mechanisms that postpone hard errors, and (ii) mechanisms that correct hard errors. We discuss the state-of-the-art prior work in each category.

Hard-Error Postponement: One can improve the lifetime of a PCM system by reducing write traffic through either exploiting write buffers [12], using encoding schemes [23] or applying compression [24]. Even recently proposed PCM prototypes/chips have an embedded read-modify-write, (RMW) circuit [13] (shown in Figure 2b) which performs differential writes (DW) to reduce bit flips, improve PCM lifetime, and reduce energy consumption. The Flip-N-Write strategy [25] is a similar, but more efficient, approach. On
each write, this strategy checks whether writing the original data or the complement of the data produces fewer bit flips relative to the stored data. Therefore, at most, half of the bits will ever have to be written on any write. The PCM memory controller may use wear-leveling algorithms in order to further postpone reaching PCM lifetime limits. The idea is to uniformly spread the write traffic over all memory blocks, avoiding premature wear-out due to write-intensive memory blocks. Different wear-leveling techniques have been proposed for PCM main such as Start-and-Gap [7]. This scheme is an inter-line (or inter-page) scheme that is reasonably efficient and imposes negligible cost. In this paper, we assume that the PCM baseline system already exploits DW and the Start-and-Gap schemes as hard-error postponement schemes, and then applies our approach to work on top of these schemes.

**Hard-Error Tolerance (Correction):** For all the above fault-postponement mechanisms, when a cell becomes faulty, the PCM memory continues to work. Conventional SECDED (Single Error Correction, Double Error Detection) schemes, which has been widely used in DRAM memories, are not a good choice for PCM for two reasons. First, SECDED are write-intensive (any update in data needs to update ECC), and so, even in the presence of DW, it is likely that an ECC chip fails before a data-chip fails. Second, in contrast to DRAM’s fault model, where faults are random and rarely happen, the number of stuck-at faults in PCM increases over time. Thus, PCM may need to correct more than one bit error. Relying on the error detection property of hard errors in PCMs, several proposed solutions exist in literature for error correction (e.g., ECP [8], SAFER [9], FREE-p [10], and Aegis [11]). We assume that our baseline PCM memory uses ECP [8], due to its simplicity and reasonably high error coverage. We also evaluate the efficiency of our proposal in a system with SAFER [9] and Aegis [11] schemes, both of which trade off design simplicity for stronger correction capabilities. Here, we describe these three mechanisms in more detail.

- **ECP** [8]: For every faulty bit, ECP keeps one pointer and one replacement bit. Error correction is performed after the read operation, restoring the faulty bit (the location where the ECC points) with the replacement bit. With 12.5% overhead for error correction code (ECC-DIMM in Figure 2b), ECP is capable of correcting 6 faulty bits. We refer to this design as ECP-6.

- **SAFER** [9]: SAFER dynamically partitions each faulty memory block in order to ensure that each partition has at most one faulty bit. Then it decides whether to store data in its original or complement form, in order to mask the faulty bit in each partition. With a 12.5% metadata overhead, SAFER deterministically corrects 6 faulty bits, and up to 32 bits probabilistically, in a memory block of size 64 bytes. The chances of correcting more than 8 bit failures are very small, as mentioned in the original paper.

- **Aegis** [11]: Aegis employs a similar strategy to SAFER (partitioning memory block, storing data in a form in order to mask errors), but is able to correct more errors with fewer partitions.

### III. THE PROPOSED APPROACH: EMPLOYING COMPRESSION FOR ROBUST PCM DESIGN

As shown in Figure 1, by employing DW in a PCM-based memory, the resultant bit flips (or changed bits) on each write have a random pattern. The main contribution of this paper is to employ compression for reducing this randomness and to show how this can aid other lifetime enhancement solutions (such as wear-leveling and hard-error tolerant) to deliver longer PCM lifetimes. Indeed, when data is stored in compressed form, the bit flips are limited to a small window of a memory block and thus the faulty bits are localized, which in turn presents a higher chance to locate errors at a lower cost and more intuitive wear-leveling.

Our approach here is general. We assume that the memory controller can use any technique for compression. There are several techniques for data compression through the entire stack of the memory system. Each achieves different optimization goals (i.e., either lowering access latency, lowering energy consumption, increasing memory bandwidth, or reducing utilized capacity). Most existing compression algorithms for cache and main memory [14]–[17] rely on value popularity. The implementation of these schemes usually trade-off compression ratio (i.e., size of compressed data divided by size of the uncompressed data) for design complexity (i.e., compression/decompression latency). In this paper, without loss of generality, we assume that our baseline PCM memory controller uses two of compression techniques: **BDI** (Base-Delta Immediate) [16] and **FPC** (Frequent Pattern Compression) [15]. Table I provides the key specifications and settings for BDI and FPC. Here, we briefly describe the functionality of each approach.

- **BDI** exploits the fact that value dynamism of the words in a memory block is usually small in most applications. BDI stores the value of one word (as the base) and differences of other words with the base (as deltas) for compression. BDI is a very fast compression scheme (taking 1 CPU cycle) and produces a fairly good compression ratio – the size of the compressed data is between 1 to 40 bytes for a 64-byte data block.

- **FPC** is based on specific predefined data patterns and uses them for data compression. The frequent patterns are used for compression at the block-level. FPC compresses 4-byte

<table>
<thead>
<tr>
<th>Compression Technique</th>
<th>BDI [16]</th>
<th>FPC [15]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target values</td>
<td>Narrow values</td>
<td>Frequent patterns</td>
</tr>
<tr>
<td>Input chunk size</td>
<td>64 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Compression size</td>
<td>1~40 bytes</td>
<td>3~8 bits</td>
</tr>
<tr>
<td>Decompression latency</td>
<td>1 cycle</td>
<td>5 cycles</td>
</tr>
</tbody>
</table>

---

Table I: The characteristics of BDI and FPC techniques.
data chunks into 3–8 bits of data. FPC’s decompression is reasonably fast (taking 5 CPU cycles).

In our memory model, the memory controller (or the CPU chip) has separate compression units for BDI and FPC that work in parallel on every data write-back. Having the results of both compression units, the memory controller chooses the compressed data of the one with smaller size for writing to the PCM memory (we will discuss metadata management later). On read accesses, the memory controller gets the compressed data (and the corresponding metadata) and delivers it to either the BDI or FPC decompression logic. Thus, with this implementation, the memory controller always stores the best of BDI and FPC outputs (in terms of compressed data size) for every memory block. Figure 3 shows the average compressed data size for BDI, FPC, and the best (i.e., it can be BDI or FPC for every data block) for a set of SPEC-CPU 2006 applications. The memory block size is set to 64 bytes in all of our evaluations in this work (Section IV provide details of our evaluation methodology, system configuration and workload characteristics).

Using the best between BDI and FPC leads to a better use of the contents of a given memory block, helping to achieve the highest compression ratio. The resultant compressed size varies for different applications, as shown in Figure 3 – applications such as zeusmp and cactusADM have the highest compression ratio (i.e., about 3/64, and 2/64, respectively) and applications such as lbm or leslie3d have the lowest (saving 13 and 19 bytes, respectively, for 64 bytes). On average, the best of BDI-FPC gives a compression ratio of 0.43.

A. The Proposed Mechanism and its Lifetime Impacts

Figure 4 presents the details of the proposed mechanism for a single memory block. Circled numbers in the text below refer to the operations shown in this figure. The assumed memory block size is \( B \) bytes and uses ECP-6 as the error correction code. As described earlier, ECP-6 is able to correct up to 6 faulty cells, regardless of their position in the block. We also assume that the memory controller reduces the amount of written data by \( W \) bytes when using compression, giving a compression ratio of \( \frac{W}{B} \). The compressed data is then stored in a small part of the block, which is called the compression window in our design. Clearly, the size of the compression window is determined by the compressibility of the written data, and thus needs to remain variable. We begin with a simple approach in which the compression window is initially mapped to the \( W \) least significant bits of the line (1). The memory controller does not shift the window (to either side) for consecutive write-backs\(^4\) if the number of faulty cells inside the window is lower than 6 in our assumed ECP-6 correction model (2).

When the number of faulty cells in the currently active compression window exceeds 6 bits in ECP-6 (or more generally, the maximum correction strength of the selected hard-error tolerant scheme), the controller shifts the compression window to the higher order bits and uses the remaining cells (that are healthy) as replacements for faulty ones (3). In other words, by shifting the compression window, the controller ensures that the number of faulty cells residing inside the window is less than, or equal to, 6 correctable errors using ECP-6. With this mechanism, the number of tolerable faulty bits per memory block is not limited to 6 (in ECP-6) anymore. The memory controller keeps writing into a memory block if it finds a contiguous region where: (i) its size is equal to or larger than the compressed write-back, and (ii) it does not have more than 6 faulty cells. This means that our scheme has the potential to substantially increase the number of correctable bits without changing the hard-error tolerant scheme.

1) Impact of compression on bit flips: Using compression in PCM memory leads to one of the two scenarios that impact the number of cells that need to be updated. If the compression ratio is high for a given write, the number of bit flips due to writing the compressed data is substantially fewer than when writing uncompressed data. On the other hand, if the compression ratio of the write-back data is low, in some cases we observe higher bit flips in compressed memory compared to the conventional memory. This can decrease the energy efficiency and lifetime benefits of our mechanism. The reason is that most compression algorithms generate somewhat random data patterns, and this can increase bit entropy (i.e., for compressed data, the probability of writing a ‘0’ bit with a ‘1’ bit, and vice versa, is generally higher than that in the uncompressed data). Figure 5 shows this behavior for our workload set when using the

\(^4\)Due to the variable size of the compression window, this means that, in our simplified design, the memory controller does not change the position of the lowest bit for consecutive writes.
BEST compression algorithm (see Figure 3). We report the percentage of write-backs with increased, decreased or untouched (i.e., number of bit flips is only about 5% lower or higher than baseline) bit flips in the compressed data, with respect to the uncompressed data. The block size is 64 bytes. Our results support the above discussion. For most of the writes in the applications that have a high compression ratio (e.g., sjeng, milc and cactusADM), the number of bit flips decreases after compression – thus, compression is not harmful to the lifetime of most memory blocks in these applications. In contrast, most of the writes in applications with reasonably low compression ratios (such as lbm and GemsFDTD) lead to an increase in the number of bit flips. An exception to this trend is in the leslie3d workload, where compression does not affect the number of bit flips, even though its compression ratio is low. Note that the bzip2 and gcc applications do not follow our expectations, experiencing a high number of bit flips even though they have good compression ratios. This issue is due to frequent changes in the size of compressed data.

In order to avoid this shortcoming of compression, we should avoid writing compressed data if it increases the number of bit flips compared to the baseline. This is not easy to implement in today’s DDR-based memory systems. The reason is that the choice of whether to compress or not is the responsibility of the memory controller, which resides on the CPU chip, while the actual number of bit flips is determined at the chip level by using the DW mechanism. Thus, the memory controller does not have information about number of bit flips ahead of time. Alternatively, one may suggest that, before writing data, we read the entire memory block, and perform a bit-wise compression of the new and old data at the memory controller in order to determine the number of bit flips before and after compression on the CPU chip. This approach has two downsides: (i) it increases the memory traffic, as each write should be preceded by a read to the same block, and (ii) we need to change the memory direction on each write, which takes some cycles in current DDR protocols. In the following, we introduce a heuristic approach to relax this problem without imposing these costs.

Our heuristic is based on two characteristics of the write patterns in real applications:

1) As discussed earlier, the number of bit flips usually decreases if the compression ratio is high.

2) On the other hand, for a written data block with low compression ratio, the number of bit flips may or may not increase. We found that the main reason for an increase is due to consecutive writes to a given memory block that has variable sizes after compression. To better understand this behavior, Figure 6 presents the probability that two consecutive writes to the same block have different sizes after compression. From Figures 5 and 6, we deduce that there is a relationship between an increase in the number of bit flips and the probability of writing different sizes into a block (example applications exhibiting this behavior are bzip2 and gcc). Figure 7 provides some better insight by showing the changes in compressed data

Figure 4: An example of the proposed mechanism.

Figure 5: Percentage of write-backs that exhibit increased, decreased and untouched bit flips after compression.

Figure 6: Probability that two consecutive writes to the same block have different sizes after compression.
sizes for consecutive writes to three blocks in bzip2 and hammer programs. We choose these two applications as they have similar compression ratios (Figure 3). The selected blocks are good representatives for similar blocks in other applications. We see that the size of the compressed data for most blocks in bzip2 significantly varies over time. The behavior is different in hammer, where writes to different blocks do not vary much after compression. Although both applications have similar compression ratios, hammer does not see a large increase in the number of bit flips as compared to bzip2.

According to these behaviors, we derive a heuristic algorithm for controlling the number of bit flips. Figure 8 shows our heuristic. Circed numbers in the text below refer to steps shown in this figure. In our design, we keep a small saturating counter per each block in order to track changes of written data (shown as SC, 2 bits in width). If the resultant compressed data size for new data is less than a threshold (Threshold1), we always write data in compressed form (Step 1). Otherwise, if SC is saturated (SC="11"), this means that the associated block has experienced variable writes (i.e., variable in size), and so the memory controller decides to write uncompressed data to avoid extra bit flips (Step 2). If SC is not saturated, (i) data is written in compressed form, and (ii) SC is updated to track data write sizes (Step 3) – if the size of new and old data does not differ significantly (i.e., their difference is less than a threshold, Threshold2), SC is decremented to reflect minor/no changes in sizes; otherwise, SC is incremented.

2) The need for intra-line wear-leveling: If the write patterns to a memory block suddenly change, the proposed naive design may encounter a serious lifetime issue. To examine this issue in more detail, let us assume that writes to a specific block have two phases. During the first phase, writes have a high compression ratio and the writes span

...a limited portion of the block (using the above proposed approach). We suppose that the first phase is long enough to have more than 6 faulty cells in the block (note that in our approach, we can tolerate many faulty bits, in the case where the compression ratio remains high). During the second phase, the write pattern changes and the modified data possesses a low compression ratio. In this scenario, it is very likely that the memory controller cannot find a contiguous compression window with less than 6 faulty bits that is capable of holding the new compressed data. Thus, our aggressive compression scheme fails to work in this scenario. To resolve this problem, we need an intra-line wear-leveling solution to evenly distribute write pressure over more bits of the same line, and hence, avoid localized wear-out within a memory block.

We employ a counter-based mechanism for intra-line wear-leveling, but in order to decrease the hardware cost, we propose to use a single large counter per memory bank (instead of having a per-line counter). On every write to blocks in a bank, the memory controller increments the corresponding wear-leveling counter. If we reach counter saturation, all new writes to that bank then use intra-line wear-leveling. Based on our sensitivity analysis, we use 16-bit wear-leveling counters with a step size of one byte.

3) Using a worn-out block after changes in compression ratio: Based on the above discussion, the memory controller marks a memory block as a "dead block" if it cannot find a contiguous section with less than 6 faulty cells to fit the compressed/uncompressed data. However, due to variations in compressed data sizes over time, it may happen that the size of the written data decreases and can be fit in an already-dead block. Although beneficial, this mechanism is very costly to implement, as we would need to check the status blocks on each write. Alternatively, we propose to check for this condition (checking if we can fit the compressed data in an already-dead block) only during inter-line wear-leveling. With this implementation, we can still achieve some benefits from reusing dead blocks, while imposing little cost.

4) Interaction with error tolerant schemes: One of the potential benefits with our approach is that we can tolerate many faulty cells, without adding any extra overhead for...
hard-error tolerance. We have discussed this benefit for the ECP-6 scheme in Figure 4 and described how our design is able to tolerate more than 6 faulty cells in a single block. We expect that our system will behave more efficiently for more advanced hard-error correction schemes such as SAFER and Aegis, which work based on partitioning, as described in Section II. The reason is that, by collocating faulty cells to a small window (i.e., the compression window) in our design, these schemes can easily partition a memory block so that there is no more than a single faulty bit in each partition. In other words, as these techniques can tolerate many faulty bits in theory, the compression algorithm increases this chance.

To examine this in practice, we conducted a set of Monte Carlo simulations for a single memory block. The error correction techniques considered include ECP-6, SAFER-32, and Aegis 17x31. In these experiments, the block size is 64 bytes (512 bits), the number of errors is changed from 1 to 128 (which are uniformly distributed over an entire block in order to model perfect intra-line wear-leveling), and the amount of written data size is changed from 1B to 64B (to model different compression ratios). The Monte Carlo results are collected after running 100,000 fault injections, and the failure probability \( (1 - \text{Reliability}) \) is reported as the figure of merit for different error correction schemes. Figure 9 presents the results for the three schemes. We make two main observations from this data. First, the higher the compression ratio, the greater the chance of using a memory block in the presence of higher bit error rates. This is true for all schemes. Second, the proposed scheme works more efficiently when introducing more advanced hard-error techniques (e.g., SAFER and Aegis). For example, if the compressed data size is 32 bytes, assuming a 0.5 failure probability, the average number of tolerable faulty bits for ECP-6, SAFER and Aegis is 18, 38, and 41, respectively.

### B. Metadata management

In our proposed mechanism, for each memory line, we need to keep three kinds of meta-data: (i) a pointer to the start of compressed block (6 bits), (ii) encoding information for the compressed data (5 bits), and (iii) a saturation counter (2 bits). Therefore, for each memory block, we need to keep 13 bits of metadata. This information is stored at the beginning of each memory line. On a read operation, after reading the whole block, the pointer bits determine which part of the block should be passed to the decompression logic, and the encoding information is used by the decompression logic to generate uncompressed data. Note that for each memory block, one bit is required to determine whether the line is compressed (or is not). Since one chip (i.e., 64 bits for each memory block) is dedicated to error correction logic, and ECP6 only uses 61 bits from that space, there are already 3 unused bits within that chip. We use one of those bits to indicate if a line is compressed. Overall, the metadata does not cause any hardware overhead on the memory side.

Once a write request is received by the memory controller, we need to determine if the data should be stored in a compressed format. As discussed earlier, a saturation counter is kept in memory, even though the decision is made at the controller. Any interaction in between could cause extra delays and is not tolerable. To resolve this issue, once a memory block is sent to the last level cache, we append to that message the size of the corresponding compressed data in memory, along with its saturation counter. This data introduces a one byte overhead per 64-bytes of memory. Since the memory controller and the last level cache are tightly connected, this modification is practical in terms of cost. Based on this design, once a cache line is written back to memory, the memory controller already knows its saturation counter value and the size of the stored value (old value) within memory. Therefore, using this information, the memory controller can decide whether that data should be stored in a compressed or uncompressed format.

Note that, cell-failure in metadata is infrequent because its update frequency is generally lower than the rate of written data. More accurately, (i) The start-pointer is updated every \( 2^{16} \) writes to the memory-bank, \( 2^{10} \) writes to a line, on average, (ii) The coding bits and counter get updated when the size of compressed-data changes, which is infrequent based on the results in Figure 6 (i.e., every 4-5 writes).
Table II: The system specification of our simulated system.

<table>
<thead>
<tr>
<th>Processor</th>
<th>ISA</th>
<th>ARM</th>
<th>CMP</th>
<th>16-core, out-of-order, 2.5GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interconnect Network</td>
<td>4×4 Mesh network</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table III: Characteristics of the evaluated workloads.

<table>
<thead>
<tr>
<th>Workload</th>
<th>WPKI</th>
<th>CR</th>
<th>Workload</th>
<th>WPKI</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar (M)</td>
<td>1.04</td>
<td>0.53</td>
<td>bwaves (M)</td>
<td>9.78</td>
<td>0.34</td>
</tr>
<tr>
<td>bzip2 (M)</td>
<td>4.6</td>
<td>0.53</td>
<td>cactuADM (H)</td>
<td>8.09</td>
<td>0.03</td>
</tr>
<tr>
<td>calculix (M)</td>
<td>1.08</td>
<td>0.37</td>
<td>gcc (M)</td>
<td>8.05</td>
<td>0.05</td>
</tr>
<tr>
<td>GemsFDTD (L)</td>
<td>4.15</td>
<td>0.70</td>
<td>gobmk (M)</td>
<td>1.14</td>
<td>0.39</td>
</tr>
<tr>
<td>hmmer (M)</td>
<td>1.9</td>
<td>0.50</td>
<td>lesie3d (L)</td>
<td>8.12</td>
<td>0.70</td>
</tr>
<tr>
<td>lbm (L)</td>
<td>15.6</td>
<td>0.79</td>
<td>mcf (M)</td>
<td>10.35</td>
<td>0.55</td>
</tr>
<tr>
<td>milc (H)</td>
<td>3.4</td>
<td>0.29</td>
<td>sjeng (H)</td>
<td>4.38</td>
<td>0.08</td>
</tr>
<tr>
<td>zeusmp (H)</td>
<td>5.46</td>
<td>0.05</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IV. EXPERIMENTAL SETUP

Evaluation infrastructure: We use the Gem5 simulator [26] for performance evaluation of a chip multiprocessor. Each core in the target system uses the ARM instruction set architecture. The memory hierarchy is based on the Ruby model provided in Gem5. We also collect traces of main memory accesses in Gem5, which are then fed to a lightweight memory simulator for lifetime analysis. The lifetime simulator models the behavior of all lifetime enhancement techniques, including chip-level DW, intra and inter-line wear-leveling, hard-error correction codes, as well as the details of our compression-based enhancements.

Baseline configuration: The evaluated CMP is a 16-core processor, using the configuration described in Table II. The memory hierarchy has two levels of caching: the L1 caches are private to each core, and the last level cache (LLC) is a 4MB cache, shared between all 16 cores. This capacity is large enough to filter out a large portion of the traffic destined for the PCM-based main memory. Below the caches is a 4GB SLC-based PCM memory, configured using the latency parameters used in NVsim [27]. The main memory has 4 channels, each interconnected to a separate memory controller on the CPU chip, one rank per each channel, with nine × 8 chips per rank (Figure 2), and 4 banks in each rank. We use the DDR3 standard, with a burst length of eight and timing parameters taken from Lee et al. [5]. Each memory controller uses separate per-bank read and write buffers. The write buffer is a 32-entry FIFO, and the read buffer is a 8-entry FIFO.

Fault model: Lifetime analysis is performed using our trace-driven PCM lifetime simulator. We feed the simulator a trace file of memory accesses collected from an application and replay the trace until it reaches the PCM lifetime limit. The PCM cell lifetime limit is set to an average of 10^{17} [5], with a variance of 0.15 (based on the model in prior work [8], [10]) in order to model process variation in the PCM array. The simulator continuously writes into a memory block, up to the point that the provided error correction mechanism (i.e., ECP-6) cannot recover the error, and consequently, the block meets its first uncorrectable faulty bit. As modeled in a prior study [8], we assume the system fails when 50% of the memory capacity is worn out.

Workloads: For our workloads, we selected 15 memory-intensive programs from the SPEC-2006 suite [28]. Each of our applications has at least one L2 cache write-back per one kilo instructions (WPKI). Each workload is composed of running the same program on all 16 cores of CMP. For each workload, we fast-forward approximately 2 billion instructions, warmup the caches by running 100 million instructions, and then collect results by simulating the next 100 million instructions on each core. Table III characterizes each workload with respect to its WPKI and compression ratio (CR). The CR values reported are for the BEST compression scheme in Figure 3. We classify our workloads into high compressibility (H), medium compressibility (M), and low compressibility (L) categories. We refer to a workload as highly compressible, if its CR is below 0.3; if the CR of a workload is above 0.7, we say that the workload has low compressibility. Otherwise, we refer to the workload as medium compressible.

Evaluated systems: We evaluate and compare the results of four systems with different lifetime enhancements:

- **Baseline**: The model described in Table II.
- **Comp**: This system uses our naive data compression scheme (i.e., *without* intra-line wear-leveling and advanced definition of hard-error correction). The compression is based on best of BDI and FPC.
- **Comp+W**: This system uses our naive data compression scheme and the proposed intra-line wear-leveling.
- **Comp+WF**: This system uses all our proposed schemes, i.e., compression-based write, intra-line wear-leveling, and the advanced definition of hard-error tolerant.

All four systems use DW at the chip level for bit-level write reduction, Start-Gap [7] for inter-line wear-leveling, and ECP-6 [8] as the hard-error tolerant scheme. Throughout our analysis, the reported results for different systems are normalized to those of the baseline system.
Figure 10: Lifetime of different systems, normalized to the baseline system.

V. EVALUATION

A. Memory Lifetime Analysis

Figure 10 presents lifetime results, with all values normalized to the baseline. In general, we find that the degree of lifetime improvement is highly dependent on the compressibility of the running application. Below, we discuss the behavior of each configuration in detail.

1) Impact of using compression (Comp): In Figure 10, we find that half of the applications experience shorter lifetimes under the Comp configuration. For instance, in gzip2 and gcc, memory lifetimes are shortened by almost half. The underlying reason for such a dramatic impact is that compression localizes write operations to the least significant bytes in a memory block, meaning that the least significant and most significant bytes (LSB and MSB) within a block wear out at wildly different rates. This issue will lead to early failures once an uncompressed block is mapped to an unevenly worn-out memory block. Note that, for highly compressible data blocks, such non-uniformity is less detrimental because once the least significant bytes are worn out, the compressed data blocks can still be placed in another working compression window. This non-uniformity, however, is not tolerable for uncompressed/less-compressible blocks.

Figure 11 can be used to quantitatively explain what types of applications still experience lifetime improvements using the Comp system. During the execution of an application, write-intensive memory addresses receive multiple write operations with different compressed sizes. In Figure 11, for each memory address, we show the size of the largest compressed block written to that address. As can be seen, in milc, 80% of the write operations are smaller than 25 bytes. Therefore, even though in the Comp configuration we have a non-uniform wear-out pattern within a memory block, 80% of the memory addresses still have a good chance to fit in those unevenly worn-out memory blocks. In gcc, however, even though this application achieves a high compression ratio, we observe a uniform distribution from 25 bytes to 64 bytes, where only 10% of the memory addresses receive write operations smaller than 25 bytes (Figure 11a). Therefore, in gcc, only a small percentage of the memory addresses can tolerate wear-out due to non-uniformity within the memory blocks.

In general, in highly compressible applications such as zeusmp, we observe similar data patterns as found in milc. On the other hand, for low compressible applications, we observe lifetime degradation. For medium compressible applications, the lifetime may decrease or marginally increase, depending both on the compressibility and the distribution of the compressed block sizes, as discussed in Figure 11.

2) Impact of intra-line wear-leveling (Comp+W): To achieve a uniform cell wear-out pattern within a line, Comp+W adopts an intra-line wear-leveling scheme for compressed write operations. As can be seen, in contrast to Comp, Comp+W does not hurt PCM lifetimes for any of the applications tested, and in fact it prolongs the average lifetime by 3.2X. In the Comp+W configuration, gcc and mcf experience considerable improvements (5.5 and 2.8 times, respectively), as compared to the Comp configuration. In other words, Comp+W resolves the premature block failure problem, which frequently occurs in Comp for less-compressible data blocks.

3) Impact of advanced hard-error tolerance (Comp+WF): The Comp+WF configuration in Figure 10 represents the lifetime improvements achieved by our advanced fault tolerance approach. Comp+WF increases the system lifetime by
4.3X on average. The first time a memory block is unable to store a new value, it is marked as permanently dead. Our Comp+WF approach however, does not mark failed memory blocks as permanently dead because those dead blocks could potentially return from the dead and be used by memory addresses with higher data compressibility.

As seen in Figure 10, not all applications benefit from this approach to the same extent. Figure 11 presents two extreme cases in that regard. For instance, in milc, 20% of the memory addresses have low compressibility when using a compressed size of larger than 40 bytes. If a particular memory block from this group of blocks fails on a write operation, it still has a high chance of being practically useful for a memory address from the remaining 80% of the memory addresses (which have a compressed size of less than 25 bytes). On the other hand, in gcc, we do not observe such a wide gap between the compressed block sizes over different memory addresses.

4) Summary: While the Comp configuration improves the lifetime of PCM on the system by 35% on average, it can also decrease the lifetime for less-compressible applications. The basic weakness in the Comp configuration is the localization of write operations to the least significant bytes within a memory block. The Comp+W configuration exploits an intra-line wear-leveling scheme to address this non-uniformity. The Comp+WF configuration does not reduce the lifetime of any of our applications, and on average, prolongs the system lifetime by 3.2X. Finally, the Como+WF configuration redefines the notion of a permanently dead block, and increases the system lifetime 4.3X on average. Table IV reports the impact of our proposed architecture on the system lifetime in terms of months.

5) Number of Tolerable errors: In the ECP technique, the storage overhead linearly increases based on the number of recoverable errors per memory block. Since ECP6 (with the capability of correcting 6 errors) introduces less than a 12.5% storage overhead, it can be considered to be a practical ECP configuration for standard memory systems, assuming one extra chip to hold ECC. However, our proposed architecture tolerates more cell failures per memory block. This is because compressed data blocks take up less space, and thus have a higher chance of fitting in a working section (i.e., a working section is a contiguous section of a block containing less than, or equal to, 6 faulty cells) of the memory block.

![Figure 12: The average number of faulty cells in a failed 512-bit memory block.](image)

Figure 12 shows the average number of tolerable errors per memory block in Comp+WF. Our proposed architecture can, on average, tolerate 3X more errors per memory block. To sustain this level of error tolerance using ECP, one would see a 40% increase in storage, which is hard to justify for many systems. Figure 12 also shows the correlation between compressibility of an application, and the number of tolerable errors within a memory block. As we observe, sjeng, milc, and cactusADM, which are categorized as highly compressible, can, on average, tolerate 25, 32, and 35 faulty cells per block, respectively.

B. Performance Overhead Analysis

In this work, the memory system is augmented with a data compression mechanism. The use of compression is not specifically trying to improve system performance (though it can increase the PCM’s effective memory capacity), but instead as a knob to improve the lifetime of the resistive memory system. As discussed in Section IV, a typical PCM-based memory system has a 32-entry write queue. Therefore, data compression can be performed in the background upon receiving write requests. Data decompression, however, is on the critical path. Read requests to the compressed memory blocks will experience longer access times. This extra delay is 1 cycle if the block is compressed using the BDI, and 5 cycles if it is compressed using FPC. Based on our experimental observations, read accesses to compressed data blocks are delayed by up to 2%, on average. Considering all the extra delays caused by the decompression process, we observe less than a 0.3% performance degradation in the applications studied.

C. Sensitivity to the Effect of Process Variation

As reported in previous work [29], process variation affects different physical dimension parameters in PCM cells, resulting in variability in their electrical characteristics. The variability in fabricated devices will become even more pronounced with aggressive feature size scaling. Imperfect wear-leveling exacerbates this problem, given the uneven distribution of write operations. Figure 13 highlights the impact of our proposed architecture on the lifetime of a memory system assuming a higher process variation. For this analysis, our PCM simulator assigns the cell lifetimes with a variance of 0.25.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Baseline</th>
<th>Comp+WF</th>
<th>Workload</th>
<th>Baseline</th>
<th>Comp+WF</th>
</tr>
</thead>
<tbody>
<tr>
<td>aastar</td>
<td>52.1</td>
<td>150.2</td>
<td>bwaves</td>
<td>8.6</td>
<td>23.6</td>
</tr>
<tr>
<td>bzip2</td>
<td>13.4</td>
<td>19.8</td>
<td>cactus...</td>
<td>9.2</td>
<td>119.6</td>
</tr>
<tr>
<td>calculix</td>
<td>51</td>
<td>159.4</td>
<td>gcc</td>
<td>8.7</td>
<td>36.2</td>
</tr>
<tr>
<td>Gems...</td>
<td>15.6</td>
<td>19.6</td>
<td>gobmk</td>
<td>30.4</td>
<td>131.7</td>
</tr>
<tr>
<td>hammer</td>
<td>32.1</td>
<td>70.6</td>
<td>Leslie3d</td>
<td>8.3</td>
<td>13.5</td>
</tr>
<tr>
<td>lbm</td>
<td>20.7</td>
<td>28.8</td>
<td>mcf</td>
<td>18.7</td>
<td>48</td>
</tr>
<tr>
<td>milc</td>
<td>16</td>
<td>184</td>
<td>sjeng</td>
<td>13.2</td>
<td>50.4</td>
</tr>
<tr>
<td>zeusmp</td>
<td>11.7</td>
<td>128.7</td>
<td>Avg</td>
<td>22</td>
<td>79</td>
</tr>
</tbody>
</table>
Current challenges involving write endurance seem to be the main obstacle preventing wide development of PCM memories. Most prior studies concentrated on improving PCM lifetime by using wear-leveling and hard-error correction schemes, implemented at the architecture level. However, little attention has been paid to lower-level optimizations—current proposals usually rely on the PCM chip’s capability of performing differential writes (DW). While beneficial in reducing the write traffic in the memory array, DW normally results in random updates of cells over the entire memory block. To reduce this randomness, and also to provide the chance of combining device-level and architecture-level optimizations, we propose to employ compression. Although beneficial in most applications, blindly using compression may increase data entropy in some memory blocks of certain applications. Motivated by this, we propose a novel compression-based PCM memory system that relaxes the problem of increased bit flipping. Our scheme can work with other wear-leveling and hard-error tolerance schemes to boost PCM lifetime. Our proposed design, on average, increases PCM memory lifetimes by 4.3×, compared to a state-of-the-art PCM DIMM memory.

ACKNOWLEDGMENT

This work is supported in part by NSF grants 1526750, 1302557, 1213052, 1439021, 1626251, 1409095, 1629915, and 1302225 and a grant from Intel.

VI. CONCLUSIONS

This work is supported in part by NSF grants 1526750, 1302557, 1213052, 1439021, 1626251, 1409095, 1629915, and 1302225 and a grant from Intel.

ACKNOWLEDGMENT

This work is supported in part by NSF grants 1526750, 1302557, 1213052, 1439021, 1626251, 1409095, 1629915, and 1302225 and a grant from Intel.

REFERENCES