COP: To Compress and Protect Main Memory

David J. Palframan   Nam Sung Kim   Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
palframan@wisc.edu, nskim3@wisc.edu, mikko@engr.wisc.edu

Abstract

Protecting main memories from soft errors typically requires special dual-inline memory modules (DIMMs) which incorporate at least one extra chip per rank to store error-correcting codes (ECC). This increases the cost of the DIMM as well as its power consumption. To avoid these costs, some proposals have suggested protecting non-ECC DIMMs by allocating a portion of memory space to store ECC metadata. However, such proposals can significantly shrink the available memory space while degrading performance due to extra memory accesses. In this work, we propose a technique called COP which uses block-level compression to make room for ECC check bits in DRAM. Because a compressed block with check bits is the same size as an uncompressed block, no extra memory accesses are required and the memory space is not reduced. Unlike other approaches that require explicit compression-tracking metadata, COP employs a novel mechanism that relies on ECC to detect compressed data. Our results show that COP can reduce the DRAM soft error rate by 93% with no storage overhead and negligible impact on performance. We also propose a technique using COP to protect both compressible and incompressible data with minimal storage and performance overheads.

1. Introduction

With the advent of multi-core processors and growing application footprints, there is increasing demand for larger main memories. Storing more data, however, also increases the chances of corruption due to soft errors caused by cosmic radiation [3]. A soft error occurs when an energetic particle strikes a vulnerable circuit node, depositing charge in its wake. The absorbed charge can result in bit flips in storage elements or transient pulses in combinational logic, either of which can lead to silent data corruption [13]. In DRAM, the standard solution to this reliability concern is to employ error-correcting codes (ECC) that can correct bit flips. However, the check bits required by such codes come at a cost, and special dual-inline memory modules (DIMMs) are typically required to facilitate their storage.

To boost DRAM reliability, single error correcting, double error detecting (SECDED) codes are often employed to protect each 64-bit word with 8 parity bits [9]. A standard ×8 non-ECC DIMM uses 8 DRAM chips per rank, while an ECC-enabled DIMM uses 9 chips, incurring a 12.5% hardware overhead. The extra hardware makes ECC DIMMs more expensive, in addition to substantially increasing power consumption relative to non-ECC DIMMs. For certain applications, these costs can be prohibitive, making non-ECC DIMMs a more sensible choice, particularly for non-critical systems. Even for systems that use standard DIMMs, however, low-cost error protection techniques are attractive. To reduce the hardware costs of error protection, recent work has proposed an approach for protecting non-ECC DIMMs [22]. This approach dedicates memory to store ECC check bits. In addition to substantially reducing the usable main memory space, this approach degrades performance due to the additional memory accesses required to retrieve the ECC check bits.

In this work, we propose COP, a low-cost mechanism "to compress and protect" non-ECC DIMMs from soft errors. We compress each 64-byte block by a small amount to include ECC parity bits along with the data within the original block size. This approach incurs minimal performance overhead, since no extra DRAM accesses are needed to retrieve parity bits. Additionally, since not all blocks are compressible, we propose a highly effective mechanism to distinguish between compressed and uncompressed blocks by checking for parity bits. We show that COP is able to reduce the soft error rate by 93% on average with *zero* DRAM storage overhead. COP is very effective because it only needs low compression ratios, rendering the majority of blocks compressible, while requiring only simple hardware to perform compression and decompression. For cases in which higher reliability is desired, COP can use a small portion of memory to store ECC metadata for incompressible blocks. We discuss how this ECC region can be efficiently managed to minimize the storage overhead. Compared to a baseline implementation without compression, we show that COP has less performance impact and can reduce the ECC storage overhead by 80% on average. This paper provides the following contributions:

- Discussion of our approach to protect memory from soft errors by integrating block-level compression and ECC, while distinguishing between compressed and uncompressed data;

ISCA'15, June 13-17, 2015, Portland, OR USA
Copyright 2015 ACM 978-1-4503-3402-0/15/06$15.00
http://dx.doi.org/10.1145/2749469.2750377

- Proposed compression scheme optimizations for COP and an efficient method to extend protection to incompressible data;
- Analysis of COP, including its reliability befits as well as improvements in performance and storage overheads.

The remainder of this paper is organized as follows: Section 2 provides background on ECC in DRAMs and motivates our proposal. Section 3 presents COP, our approach for highly-efficient protection against DRAM errors. Section 4 discusses the simulation methodology and results including reliability and performance. Finally, Section 5 concludes the paper.

2. Background and Related Work

To make main memory more robust and protect against soft errors, many memory controllers are able to support ECC-enabled DIMMs. Such ECC-enabled DIMMs add an extra chip or chips to each rank to store ECC check bits. For instance, a standard ×8 DIMM uses ranks composed of 8 chips each, while an ECC DIMM adds a 9th DRAM chip to each rank. This allows the storage of one byte of check bits for every 8-byte word, in accordance with the commonly-used (72,64) SECDED code [9]. Adding the extra DRAM chip makes ECC DIMMs more expensive than their standard counterparts in terms of the up-front cost as well as power consumption.

For DRAM systems requiring very high reliability, chipkill codes can also be employed [7]. These codes typically employ ECC DIMMs and are able to correct whole-chip failures, which are often the result of hard errors. Although robust, these codes come at the cost of complexity, which is often merited in high-end systems. In this work, however, we consider mid-range systems that do not employ ECC DIMMs. Given the lack of dedicated hardware, we attempt to improve system reliability as much as possible and focus our discussion on soft error mitigation.

To reduce the costs of error protection, prior work has suggested approaches to improve the resilience of non-ECC DIMMs. Such approaches allocate dedicated portions of main memory to hold ECC metadata. For instance, Yoon et al. propose Virtualized ECC, which allocates full memory pages for ECC [22]. When Virtualized ECC is used with a non-ECC DIMM, each data block retrieved from DRAM requires an additional read to retrieve the ECC check bits. To locate the ECC page containing the check bits, a page-table-like structure is used. Finally, to avoid the high cost of an ECC address table walk, a 2-level ECC address translation cache is employed.

There are a number of downsides to this in-memory ECC storage approach. First, it significantly reduces the overall usable memory space. For instance, if a (72,64) SECDED is used to protect an 8GB main memory, 910MB must be reserved for ECC bits. The DRAM accesses to retrieve ECC check bits can also reduce performance by increasing contention and access latencies. If a page-based scheme is used, extra hardware is also required for ECC address translation. To reduce the latency overheads of accessing ECC metadata stored in non-ECC DIMMs, prior work also suggests distributing the metadata throughout memory so that it is collocated in the same DRAM row as the data [23]. With an open-row policy, this “embedded ECC” configuration can improve the ECC access latency, although the same storage overhead as Virtualized ECC is imposed.

With COP, we provide ECC protection for main memory without impacting the available memory space and with minimal performance degradation. This is accomplished by compressing each 64-byte block of memory enough to include the ECC. Prior work by Chen et al. has suggested exploiting the unused fragments inherent in compressed last-level caches (LLCs) to hold ECC bits [6]. This approach is effective for caches, but does not extend well to main memories where it is harder to add compression-tracking metadata without dedicating space in DRAM. When using embedded ECC in memory, an approach called MemZip suggests using per-block compression to move ECC bits from the end of the row so that compressible 64-byte blocks contain both data and ECC bits [18]. This approach is only a performance optimization, and space must still be reserved for ECC regardless of compressibility. Additionally, dedicated storage space in each row is also required to track the compression status of each block. For COP, however, dedicated compression metadata is not required to track compression, and ECC storage does not significantly reduce the usable memory space. A US patent by Stracovsky et al. also describes how to combine compression and ECC in memory, but is unable to track incompressible data without extra storage, as COP can, and the patent provides no quantitative evaluation or comparison to prior work [20].

Prior work on enhancing performance through cache and memory compression targets a compression ratio of on the order of 2x [1, 14, 15]. Because such approaches typically aim to increase the effective cache or memory capacity, high compression ratios are required for any meaningful benefit. With COP, however, we can optimize compression approaches to compress high-entropy blocks with limited compressibility, since we only need to free a few bytes per block to accommodate the ECC bits. In cases where exhaustive protection is desired, we also propose an efficient mechanism for managing the small amount of metadata required.

3. Economical ECC for non-ECC DIMMs

3.1. Overview

The cost of ECC-enabled DRAM DIMMs can be unattractive for commodity systems, yet resilience to bit flips from soft errors is still desirable. We therefore propose the COP approach to significantly enhance the reliability of non-ECC DIMMs with negligible overhead. We compress each 64-byte block of memory just enough to insert ECC check bits to protect the block against errors. Because the compressed data and ECC bits are the same size as the original data block, no additional memory accesses are needed and no extra storage space is required to accommodate the ECC. In addition, maintaining block alignment in memory means that addressing is not affected, which is a potential issue with memory compression.

Traditional intuition regarding cache and memory compression says that a portion of application data is likely incompressible. This is partially because low compression ratios are not considered useful, particularly in context of compressed caches, which may be statically segmented. Figure 1 shows the compressibility of blocks for selected SPEC benchmarks using the frequent pattern compression (FPC) algorithm [1]. When only a low compression ratio is required, many more blocks can be considered compressible. As shown, the data for certain applications such as libquantum is not very compressible overall, yet the majority of blocks can be compressed by a small amount (e.g. 10%). COP works well because it only requires a small amount of compression per block, unlike traditional compression applications that seek to provide at least a 2x (50%) compression ratio overall. Furthermore, we observe that algorithms such as FPC are engineered to provide high compression ratios overall, and are less optimized for lower compression ratios. To address this inefficiency, we discuss more efficient algorithms and propose optimizations later in this section. These optimized compression approaches are able to compress over 90% of blocks on average, allowing high error coverage.

In COP, the amount of compression achieved for each block determines the number of ECC check bits that can be inserted, and therefore which codes can be used. Although it is theoretically possible to use stronger codes for more compressible data blocks, for simplicity, we target the same compression ratio for each block. In our evaluation, we experiment with freeing 8 bytes or 4 bytes per block to use for ECC. We find that when only 4 bytes must be freed, we are able to compress most blocks, constituting a compression ratio of 6.25%. For this reason, the remainder of our discussion will describe the 4-byte case, although our mechanism can be applied for other configurations. In our preferred scenario, each 64-byte block in memory contains 60 bytes of compressed data with room to add 4 bytes of ECC check bits. Instead of using a single code to protect the block, COP divides each block into four (128,120) SECDED codes. Each code requires one byte of check bits to protect 15 bytes of data. This design benefits COP in two primary ways. First, it allows us to use a relatively simple SECDED code, since the (128,120) code is the full version of the commonly-used (72,64) truncated code. Second, this approach allows COP to detect whether or not a given block is compressed (and protected) or was stored unprotected because it could not be compressed.

Although COP requires only a small amount of compression for each block (e.g. 6.25%), some blocks may not be compressible at all. The default COP approach cannot protect these blocks, since check bits cannot be included. One option in this case is to simply leave incompressible blocks unprotected, since we can (ideally) protect the vast majority of other blocks.

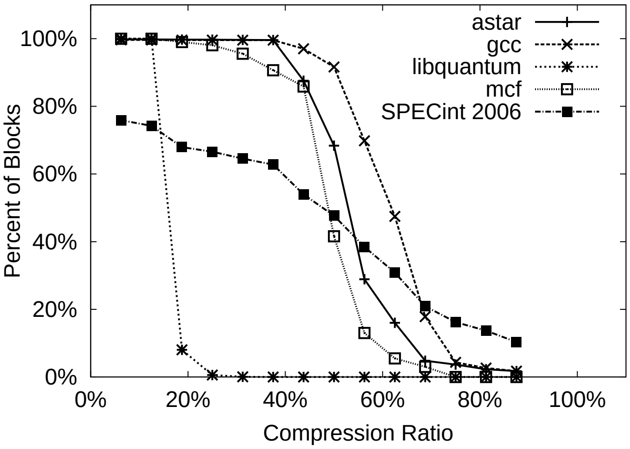
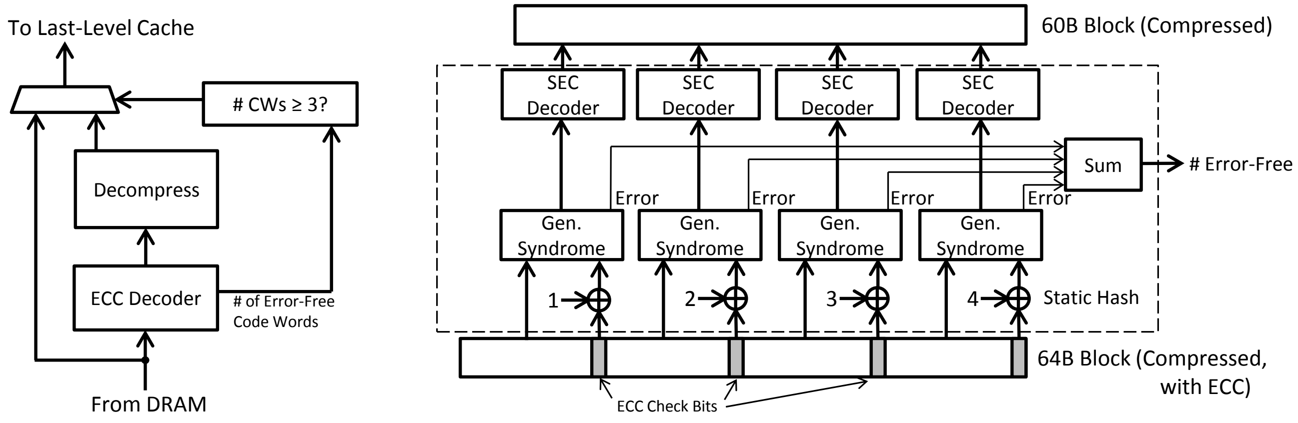


Figure 1: Percent of blocks that can be compressed using FPC given a target compression ratio. More blocks can be compressed if less compression is required.

This approach still achieves a high level of error protection while incurring minimal performance and storage costs, since all ECC is included inline with each block. COP can seamlessly integrate uncompressed blocks alongside compressed blocks in memory, provided that it can tell the difference. Distinguishing between the two is critical and is a key contribution of COP. Without using extra space in DRAM, however, it is not possible to store metadata for all blocks to track which are compressed and which are not. One solution could be to compress each compressible block enough to add a special sequence of bits to the beginning of each to indicate that it is compressed. In addition to requiring more compression, this approach also introduces the (relatively high) possibility that an uncompressed block will be mistaken as compressed if its data happens to contain the special bit sequence.

To determine if a block is compressed or not without incurring extra overhead, COP examines each block after it is read from DRAM and simply checks for valid ECC code words. In this context, we define a valid code word as a 128-bit word which, when passed through a particular (128,120) SECDED decoder, produces a zero syndrome which would traditionally indicate a lack of errors. Unlike the previously-discussed approach with the special bit sequence, the chances of an uncompressed data block containing multiple valid code words are extremely low, and in the very rare cases when this occurs, COP *guarantees* correctness, as we will discuss. After reading a block from DRAM, COP treats it as if it contains four (128,120) code words, and passes all of them through the ECC check logic, as shown in Figure 2. If the block was compressed/protected before being written to memory and no error has occurred, the ECC logic will detect 4 code words with zero errors. If the block was not protected, the entire 64 bytes contains uncompressed application data. In this case, it is unlikely that the ECC decoder will detect a single valid code word and highly unlikely that the block happens to con-



(a) Decoder/decompressor logic.

(b) ECC decoder logic.

Figure 2: ECC decoder/decompressor logic. The ECC decoder counts the number of valid error-free code words seen. If enough code words are seen, the data (minus ECC check bits) is decompressed and sent to the LLC. If not enough code words are seen, the data is passed unmodified to the cache.

tain multiple valid code words. In our implementation, if the memory controller detects 3 or more valid code words in a block, it will treat the block as compressed data. If a soft error has occurred in a protected block, 3 of the 4 code words will remain valid, and the ECC can be used to correct the invalid code word before the block is decompressed. As shown in Figure 2, if fewer than 3 blocks contain valid code words, the block is treated as unprotected data and passed to the last-level cache unmodified.

Although it is very unlikely, it is possible that a block of data could happen to contain 3 or more code words, which would confuse the decoder. We will refer to these blocks with 3 or more valid code words as aliases because they would appear to the decoder to be compressed blocks even though they are not. For functional correctness, we must guarantee that it is impossible for an uncompressed block to be erroneously treated as a protected/compressed block because it is an alias. This can be accomplished very simply if we never write aliasing blocks to DRAM and instead keep all such blocks in the last-level cache. Figure 3 illustrates the possible types of blocks in COP. Most blocks will be compressible, and will be stored to DRAM in compressed format along with ECC parity bits. As shown, a subset of blocks are incompressible, and a subset of blocks can also be considered aliases. Compressible blocks that are aliases in their uncompressed form are not a concern, because they will be compressed in DRAM. As shown, however, the extremely rare blocks that are both incompressible blocks and aliases are not allowed in DRAM and must reside in the LLC to ensure that the decoder works correctly. Note that we do not need to keep incompressible blocks containing only 2 valid code words in the LLC, even though a soft error could theoretically create a 3rd valid code word. Although this scenario would cause the block to be misinterpreted as compressed, the error is guaranteed to result in data corruption in any case since the block was unprotected.

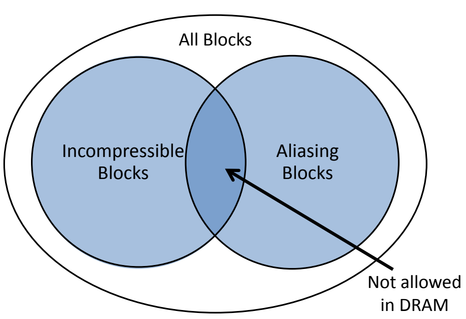


Figure 3: Illustration of blocks allowed in DRAM with COP (not to scale). If a block's data contains 3 or more code words, it is considered an alias. For the majority of blocks that are compressible, aliasing is not a concern. Incompressible aliases cannot be written to memory because the decoder will misinterpret them.

To give the reader an idea of the probability of a block being an alias, consider the (128,120) code mentioned before. Because there are 120 data bits, this code allows for 2^{120} valid code words, while there are 2^{128} possible values that can be represented by the 128 bits when including the parity bits. Given a random 128-bit value, there is then a 0.39% chance that it will be a valid code word. Given a 512-bit block containing 4 random 128-bit values, there is a 0.00002% chance of the block containing 3 or more valid code words. Since the majority of blocks are compressible and we must only retain incompressible aliases in the LLC, very few blocks, if any, will fall into this category. To remember that a block in the LLC is an incompressible alias, COP requires an extra bit for each LLC block. Upon a writeback to memory, the compressor/ECC encoder logic checks for incompressible aliases and rejects writebacks of these blocks, requiring them to be kept the LLC with the "alias" bit set. Alternatively, a check could be added to writebacks to the LLC to proactively set this bit.

Although the previous example discusses the chances of aliasing for a completely random data, application data is not usually random. For some applications, blocks may contain the same word repeated multiple times. In this case, if the repeated data happens to be a valid code word, the block will contain multiple valid code words, significantly changing the odds discussed above. To avoid this scenario, we introduce a static hash that is XORed into each compressed block when it is written by the encoder and before it is processed by the decoder. By using a different hash for each 128-bit segment as shown in Figure 2, we ensure that repeated values will not skew the odds.

We do not include a diagram of the ECC encoder/compressor because it is the opposite of the decoder/decompressor shown in Figure 2, with the stages in reverse order. In the encoder, the compression logic first compresses the 64-byte block down to 60 bytes. The SECDED logic then computes 4 bytes of parity bits for the compressed data, after which the static hash is applied. The compressed/protected block can be written to DRAM at this point. If the compressor cannot compress the block, the 64 bytes of data are written to DRAM as-is, and no hashing is applied.

The observant reader will have noticed that although we employ 4 SECDED codes when protecting a compressed block, the decoder implementation described limits us to correcting only one bit error per block or detecting a double error within one 128-bit code word. If two errors occur and corrupt different SECDED code words, there will be only two valid code words remaining, resulting in data corruption when the decoder erroneously passes the compressed block to the processor without decompressing it. To extend correction to this scenario, the code word threshold could be reduced from 3 to 2, although the number of aliases would increase by orders of magnitude. In a different COP implementation using 8 bytes of ECC metadata per block, COP could divide each compressed block into 8 (64,56) SECDED words, allowing the decoder to still require a high valid code word count (e.g. 5 out of 8) but enabling single-bit correction in multiple code words. As previously discussed, however, requiring more compression reduces the overall number of blocks that can be compressed/protected.

It is possible, though extremely unlikely, for too many incompressible aliases to map to the same set in a set-associative LLC, causing an overflow condition for that set. There are various approaches to handling this corner case. One approach is to disable compression for all of memory, falling back on a technique like Virtualized ECC and allocating space for ECC in memory [22]. Another solution is to spill the addresses (and/or data) of the incompressible aliases to a small region of DRAM (likely less than a page in size). Spilling of the set could then be accomplished by adding an overflow flag bit per LLC set and reserving one of the tag fields to use as a pointer to the overflow blocks. Any misses to an overflowed set must then follow this pointer to search the overflowed blocks for a hit before performing conventional miss handling. The overflow blocks can be arranged as a linked list, allowing an arbitrary number of collisions to be handled, albeit with additional latency. Since this overflow scenario is exceedingly rare, this slow mechanism is only needed for correctness and does not create a performance bottleneck.

3.2. Compression Schemes

Traditionally, the goal of cache or memory compression schemes is to achieve the highest possible compression ratio. High compression allows more data to be cached or most efficiently uses memory bandwidth. In COP, however, the purpose of compression is to make just enough room for ECC check bits. To avoid incurring additional memory accesses to retrieve the ECC bits, it is most desirable to store them inline with each 64-byte block, and there is little additional benefit from high compression ratios beyond what is required for ECC.

Because COP does not target high compression ratios, it can use simpler and less aggressive compression algorithms. A key insight is that many compression approaches that are capable of providing high compression ratios are ineffective at compressing blocks with more limited compressibility. This can occur when the overhead of the compression metadata required outweighs the space reduction achieved. For instance, frequent pattern compression (FPC) requires a 3-bit prefix per 32-bit word, thus incurring a cost of 48 bits of metadata per block [1]. To free 4 bytes (32 bits), we must recoup the cost of the metadata and extract a total of 80 bits of redundancy from the block. By using a compression algorithm requiring less metadata, blocks with limited compressibility can be protected.

The remainder of this section proposes compression schemes and optimizations for use with COP. For each scheme, we increase the target compression ratio by 2 bits (freeing 34 bits overall) to allow COP to combine compression schemes for maximum benefit. In the combined approach, COP uses two bits of every compressed block to indicate which compression scheme was used.

3.2.1. MSB compression This approach is inspired by the (more-complex) base delta immediate (BDI) algorithm. BDI can compress a cache line containing values that are similar in magnitude [15]. It works by storing each data block as a base value and a set of deltas. To decompress the block and retrieve the original values, each delta is simply added to the base. Because fewer bits are required to store the deltas, the block can be significantly compressed. For instance, if the base is a 4-byte word and each delta is one byte, then a 64-byte block can be compressed into only 19 bytes, compressing the data 70%. As previously mentioned, this is much higher than the compression ratio required by COP. For COP, we could extend the dynamic range of the BDI algorithm by increasing the number of bits per delta. Adopting a variant of BDI with

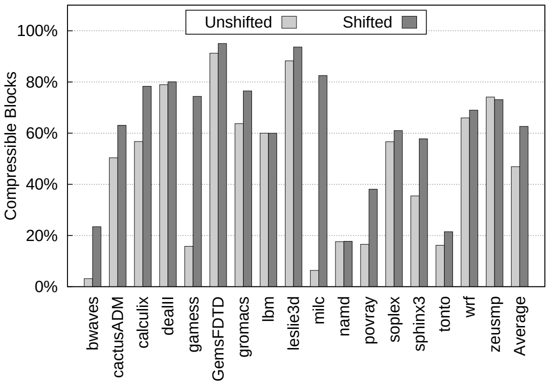


Figure 4: Compressibility improvement using MSB compression when the comparison is shifted by 1 bit. To be considered compressible, 4 bytes must be freed in a block.

wider deltas would decrease the compression ratio but allow more blocks to be compressed.

During BDI decompression, we observe that the most significant bits (MSBs) of the base value typically remain unchanged after the deltas are added. For instance, if a delta is one byte while the base is four bytes, a long carry chain is required to change the MSBs of the result when the delta is added to the base. As a simplification of BDI, we can simply check for redundancy in the form of matching MSBs across each value in the block. In COP's MSB compression, if we observe that across the 8 8-byte words in a block, 5 of the MSBs are always the same, we can remove the redundant bits from 7 of the words. This compression frees 35 bits, making room for 32 bits of ECC and 2 bits to indicate the compression scheme. To free more than 4 bytes per data block, we can simply increase the number of MSBs compared. We use 8 bytes as the comparison stride to allow blocks of 64-bit values to be compressed, and observe that the algorithm remains effective for 32-bit values, since in this case, half of the values in the block will be omitted from the comparison. This MSB compression requires less logic than BDI, since no addition is required.

We also observe that traditional BDI compression is not very effective for blocks containing floating point values, since the significands of floating point values are left-normalized. The bit comparison used by MSB compression, however, overlaps the floating-point exponents, allowing it to compress floating point values with similar exponents. The most significant bit in standard floating point representation is the sign bit, however. If we include the sign bit in the MSB comparison, blocks containing values with different signs cannot be compressed. To optimize overall compressibility by including this case, we slightly modify the previously discussed approach by shifting the 5-bit comparison over by one bit, such that we ignore the most significant bit. This optimization can significantly benefit the compressibility of floating point applications. To demonstrate this benefit, Figure 4 shows the compressibility of SPECFP 2006 benchmarks using MSB compression with both unshifted and shifted comparisons. As shown, by shifting the MSB comparison by 1 bit, compressibility improves by 15% for these applications.

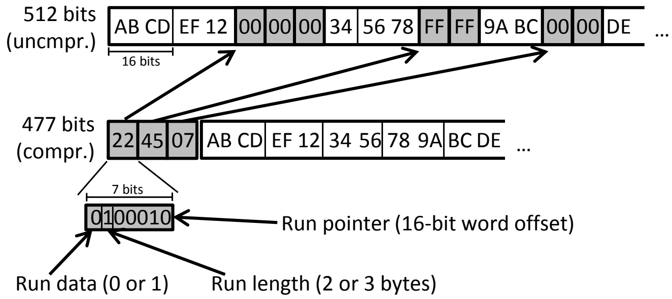


Figure 5: Example of compression using run length encoding. Each run is encoded using 7 bits of metadata.

3.2.2. Frequent pattern compression Frequent pattern compression (FPC) uses a 3-bit prefix for every 32-bit word to encode common patterns such as repeated bytes or sign extended values [1]. Because FPC is a well-known algorithm in the realm of cache compression, we evaluated its effectiveness in conjunction with COP. Our results indicate, however, that while FPC is best-suited for achieving high compression ratios it is less effective for less-compressible blocks due to the fixed metadata overhead. Interestingly, we find that in the context of COP, a simplified run length encoding can extract the redundancy in the same sign-extended values as FPC with less metadata overhead, allowing more blocks to be compressed.

3.2.3. Run length encoding Run length encoding extracts redundancy in the form of runs of binary 1s or 0s [10]. The benefit of this approach over a compression algorithm such as FPC is that we require only a small number of short runs and metadata is required only to encode each run. In our implementation, we require 7 bits of metadata for each run, where runs can be either 2 or 3 bytes long. Therefore, accounting for the metadata, we need only 2 3-byte runs, 4 2-byte runs, or a combination or 2-byte and 3-byte runs in order to free 4 bytes for ECC. In the case with 2 3-byte runs, for instance, we remove the 6 bytes of redundancy while adding two 7-bit pointers, freeing 34 bits overall. This leaves us two bits two indicate the compression type when combining compression schemes.

The 7-bit metadata to encode each run is structured as shown in Figure 5. The first bit indicates if it is a run of 0s or a run of 1s. The next bit indicates whether the run is 2 bytes long or 3 bytes long. Finally, the next 5 bits point to the 16-bit word offset in the block where the run begins. To compress a block, metadata for each run is placed at the start of the block, and the runs described by the metadata are omitted from the rest of the block. Only the minimum number of runs must be encoded to create space for ECC. Thus, the number of runs encoded per block can vary depending on the length and number or runs present. When decompressing a block, the number of runs (and metadata chunks) can be determined by examining the

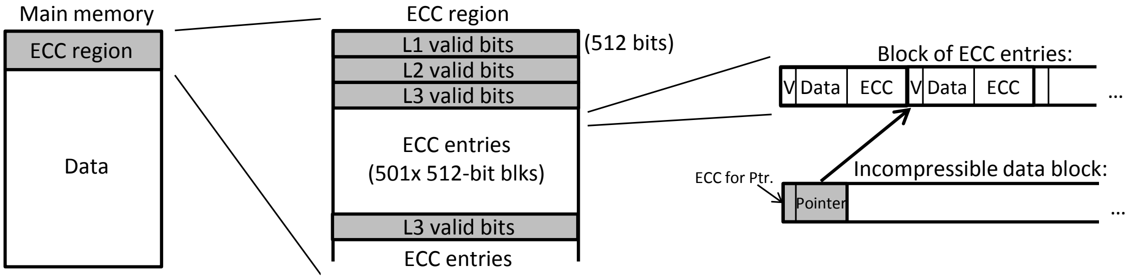


Figure 6: ECC region organization for COP-ER. A small portion of main memory is allocated for ECC. Each uncompressed data block is truncated to include a pointer to an ECC entry containing the displaced data and parity bits for the block.

metadata chunks and counting the number of bytes freed by each one. Once enough bytes have been freed to store the ECC (e.g. 4 bytes), we can determine where the metadata stops and the actual data begins, since no more runs would be needed.

3.2.4. Text compression Many widely-used applications are responsible for processing large amounts of text. Latin characters that fall within the range of ASCII encoding are very common. The ASCII standard is a 7-bit encoding that defines 128 possible characters [2]. Historically, one byte was transmitted per character, with the 8th bit being used for parity. In more modern storage of ASCII characters, each character is stored in a byte with an extra zero as the most significant bit. If an entire memory block contains ASCII characters, the MSB of all bytes will be zero, and can be omitted to compress the block. This approach works well in the context of COP, since it is capable of providing only low compression ratios. For instance, we could theoretically free 62 bits in a 64-byte line (a compression ratio of 12%), assuming 2 bits to indicate the compression type.

The more modern Unicode formats enable additional characters, but maintain backwards compatibility with ASCII. For instance, UTF-8 is a variable length encoding that mixes ASCII symbols with longer symbols. UTF-16, on the other hand, uses a fixed 2-byte representation per character. To convert ASCII characters to UTF-16, one byte of zero padding is simply added. Therefore, even when considering Unicode text, if a data block contains only ASCII characters (which is even more likely for UTF-16, as a block contains half the number of characters), it can be compressed using this approach. Even for languages with non-latin characters, characters that fall within the ASCII range are still commonly used, as is the case for HTML.

3.3. Protecting incompressible blocks

As described thus far, COP is able to protect compressible blocks, which comprise a high percentage of application data. For certain applications or hardware implementations, it may be desirable to protect all of memory against bit flips, including incompressible data. Prior proposals (e.g. Virtualized ECC) can protect non-ECC DIMMs by allocating space for ECC check bits in memory [22]. As previously discussed, there are two downsides to this approach. First, extra memory accesses are required to retrieve the check bits associated with the data, which can degrade performance. Second, making room for ECC check bits in main memory substantially reduces the usable memory space.

In cases where it is imperative to protect all data from errors, a hybrid version of COP can allocate a small ECC region in memory. We refer to this version of COP as COP-ER. In a naïve implementation, the same storage overhead as Virtualized ECC is required, since incompressible blocks are not always adjacent, so ECC space could be reserved for all blocks to facilitate addressing of the ECC region. In this manifestation, the benefit of the combined approach is in performance, since most of the time the check bits can be retrieved with the compressed data, and the ECC region need not be accessed.

Since the ECC region is only used to store check bits for incompressible blocks, an optimized implementation can significantly reduce its storage overhead such that the available memory storage is minimally impacted. Through appropriate engineering of the ECC region, we can use the space as efficiently as possible, tightly packing ECC data. When ECC entries are not present for all blocks, a simple offset computation is no longer effective for finding a block's parity bits in the ECC region. To solve the problem of efficiently locating ECC metadata following a read to an incompressible block, we suggest displacing a portion of each incompressible block's data and inserting a pointer to point directly to an entry in the ECC region. Each entry in the ECC region is then comprised of a valid bit, the displaced data, and the ECC check bits needed to protect the whole block.

Figure 6 shows the structure of COP-ER's optimized ECC region. As shown, the ECC region occupies a portion of the memory space and can grow dynamically as needed. To allow the region to be resized, the operating system can avoid allocating the nearby pages until memory is near capacity. As shown, the bulk of the ECC region is comprised of blocks containing ECC entries. If 28-bit pointers are used in incompressible blocks to point to an ECC region block/entry offset, an additional 6 parity bits are required to correct any bit errors in the pointer, displacing a total of 34 bits from each incompressible block. Each ECC entry, therefore, must contain 34 bits of

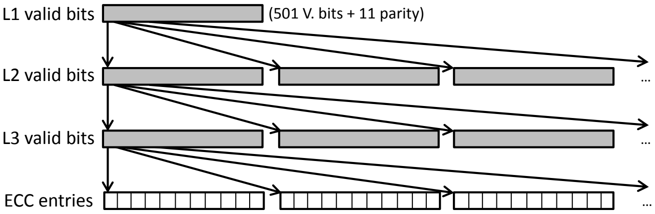


Figure 7: Valid bits form a high-radix tree to speed the location of free entries in the COP-ER approach.

data plus 11 bits of parity for the incompressible block plus a valid bit for a total of 46 bits per entry, allowing 11 entries per 64-byte block.

On a read to an incompressible block, the decoder first detects that the block is not in compressed format (see Figure 2). Next, a read is performed to access the ECC region block containing the entry indicated by the embedded pointer. Note that ECC region blocks can be cached to improve performance. The missing data and parity bits are retrieved, and the block is checked for errors. We add an additional bit alongside each L3 cache block to indicate if the block was uncompressed when originally read from DRAM. On an LLC replacement, if the victim line is clean, it can be silently invalidated and overwritten with the new block. If the victim is dirty and the "was uncompressed" bit is set, we know that an ECC entry already exists for the block. In this case, the pointer to the ECC entry is read from memory. If the block is now compressible, the original ECC entry is invalidated and the block is written in compressed format. If the block is still incompressible, the existing ECC entry can be reused and updated with the new data/parity bits. If the "was uncompressed" bit is *not* set and the dirty data being written back is incompressible, a new ECC entry must be allocated.

To find a free spot for a new ECC entry, the valid bit hierarchy shown in Figure 7 is used. These bits allow free ECC entries to be efficiently found and filled without a (time-consuming) exhaustive search. In a worst-case scenario, a large main memory storing a lot of incompressible data could require millions of ECC entries, so COP-ER needs an efficient way of maintaining and searching the free list. Each L3 valid bit block contains 501 valid bits in addition to 11 bits of parity to protect the valid bits. Each valid bit corresponds to one block of ECC entries, as shown. When all 11 ECC entries in an ECC region block are valid, its L3 valid bit is set. L3 in this context refers to the level in the valid bit hierarchy, not a cache level. The memory controller stores a pointer to the most recently used block of L3 valid bits. To allocate a new ECC entry, it looks for a 0 valid bit, indicating a block of ECC entries with a free entry. If all 501 valid bits are set, the 3-level valid bit hierarchy is walked. Each block of L2 valid bits corresponds to a block of L3 valid bits, and a similar arrangement applies for the L1 bits. When the last ECC entry in a block is allocated or an entry is freed in a full block, the tree structure is updated appropriately. By filling free entries this manner, COP-ER limits the size of the ECC region in case the data compressibility changes or memory is deallocated.

Table 1: Simulator configuration

Category	Configuration
OoO Core	3.2 GHz Issue: 4-wide Window size: 128
Caches	L1 Instr: 32 KB/4-way, 4 cycles L1 Data: 32 KB/8-way, 4 cycles L2: 256 KB/8-way, 9 cycles L3: 4 MB/16-way, 34 cycles
Memory	Bus speed: 1600MHz Bus width: 64 Total capacity: 8GB Channels: 2 DIMMs per channel: 1 Ranks per DIMM: 2 Chips per rank: 8

Table 2: Memory-intensive benchmarks

SPECint 2006	SPECpf 2006	PARSEC
astar	bwaves	canneal
bzip2	cactusADM	fluidanimate
gcc	GemsFDTD	streamcluster
mcf	lbm	x264
omnetpp	milc	
perlbench	soplex	
sjeng	wrf	
xalancbmk	zeusmp	

Another benefit of COP-ER is that it can virtually eliminate the incompressible alias problem. Recall that COP-ER displaces some data in incompressible blocks to store an ECC region pointer. This provides some control over the bits stored in DRAM that will be analyzed by the memory controller. If the bits used to form the pointer are selected such that they overlap with all four code words as seen by the decoder, ECC entry allocation can be adjusted so that the block is no longer an alias when incorporating the pointer.

4. Evaluation

To evaluate COP, we used an approach inspired by the interval simulation methodology [8]. This approach divides execution into intervals between long-latency miss events, which will have the largest performance impact and overshadow lower-latency accesses. This high level of abstraction allows us to efficiently simulate many instructions and large input sets. To accurately model memory system behavior, our simulator relies on DRAMSim2 [17]. For our evaluation, we used the SPEC2006 benchmarks with reference inputs as well as the PARSEC suite with native inputs [4]. A section of 1 billion instructions was run from each SPEC benchmark, chosen

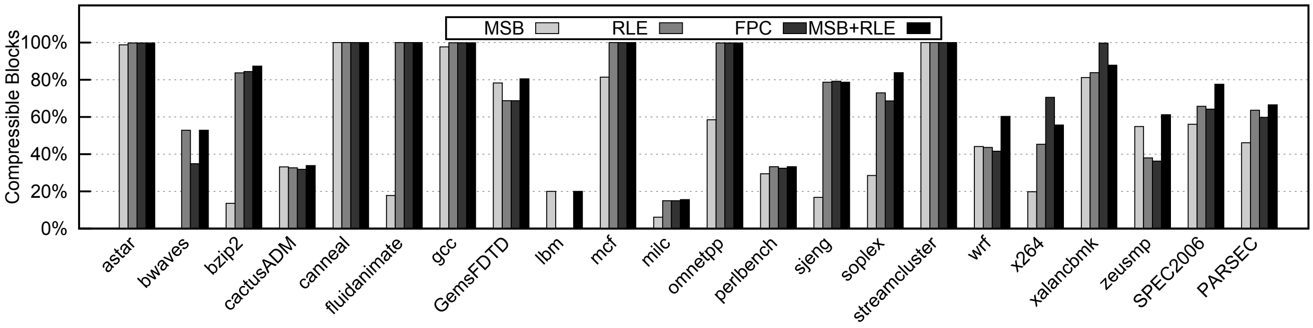


Figure 8: Compressibility when freeing 8 bytes per 64-byte block.

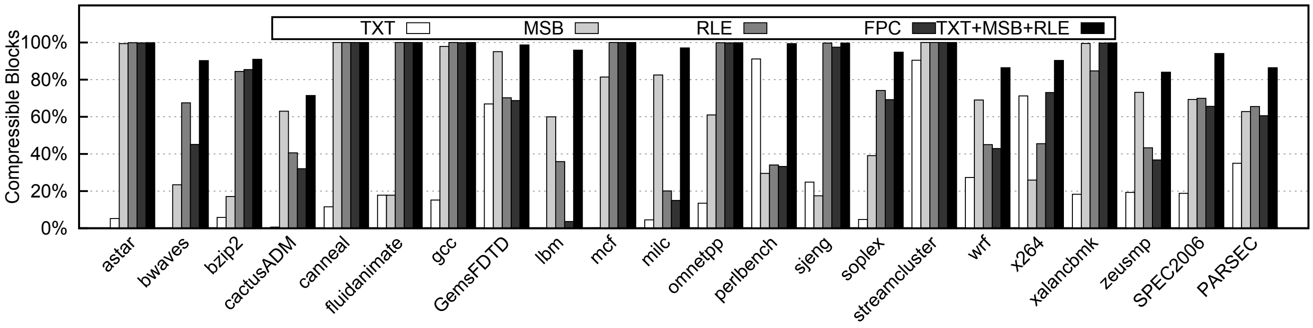


Figure 9: Compressibility when freeing 4 bytes per 64-byte block.

using the SimPoint tool [16]. The PARSEC workloads were run in 4-threaded mode for 4 billion instructions in the parallel region of interest.

We used Sniper, a Pin-based simulator to capture a trace of references to the L3 cache along with the data contents of each referenced cache block for compressibility analysis [5, 12]. These references were divided into epochs, each containing independent (overlappable) requests. The perfect-L3 IPC was also recorded, allowing our simulator to compute the performance impact of L3 cache misses. We modeled a memory system with a 4MB L3 cache backed by a dual-channel 8GB main memory. Table 1 shows the details of the simulated system. Because we are concerned with improving the resiliency of main memory, our results highlight the memory-intensive benchmarks shown in Table 2. This subset includes 20 benchmarks from the SPEC and PARSEC suites. We also show averages for each suite in the result figures.

To evaluate the effectiveness of the compression schemes considered, we simulated each benchmark while noting the compressibility of each DRAM block accessed. We counted the total number of DRAM accesses as well as the number of accesses to compressible blocks. Two compression ratios were evaluated for this experiment. Figures 8 and 9 show the results for freeing 8 or 4 bytes, respectively. As shown, COP can compress significantly more blocks if only 4 bytes must be freed. As shown in Figure 9, text compression (TXT) is particularly effective for certain benchmarks such as perlbench. MSB compression, on the other hand, is very effective overall and able to compress approximately 70% of blocks on average. Run length encoding (RLE) is similarly effective overall, though some benchmarks favor one of either MSB or RLE. We also evaluated the effectiveness of frequent pattern compression (FPC) for comparison. Because RLE generally outperforms FPC and has a simpler hardware implementation, we do not include FPC in the combined compression algorithm. The combined algorithm includes the best of all of the schemes, incorporating TXT, MSB, and RLE using two bits to select one of the three. As shown, the combined approach is highly effective and able to compress 94% of blocks on average.

We also modeled the reliability benefits of COP. When evaluating the resilience of DRAM, a number failure modes can be considered, including both hard errors and soft errors. A study by Sridharan et al. showed that 49.7% of failures in the field (both hard and soft errors) were single-bit errors [19]. Another 2.5% of failures were multi-bit failures in the same word, and 12.7% were multi-bit failures in the same row. Just like a conventional SECDED (ECC DIMM) approach, COP is unable to correct multi-bit failures in the same word. Multi-bit row failures are also likely the result of failing peripheral circuitry that neither SECDED nor COP can repair. Other failure types (e.g. single-column) will generally corrupt only one bit per block, which can be corrected by SECDED or COP. Because of the similar correction capabilities (and limitations) of standard SECDED and COP and due to the complexity of accurately modeling all possible failure modes, we used a single-bit failure model. Note that this approach does model double-bit errors, which are modeled as separate single events.

For our analysis, we used a methodology inspired by the PARMA reliability model [21]. To compute the soft error rate for L2 caches, PARMA introduces the idea of a “vulnerability

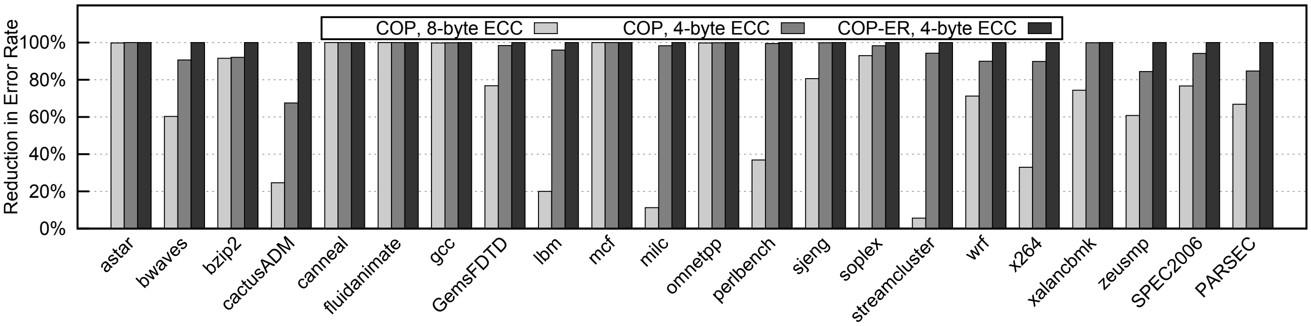


Figure 10: Error rate reduction with COP. The 4-byte ECC version requires less compression, allowing protection of more blocks.

clock" that counts the number of cycles that each data block is vulnerable in the cache before it is read. Using a raw error rate per bit, the analytical model can then compute the overall error rate contribution of each block. We adapt this approach for use in DRAM, and track the amount of time that each data block is vulnerable in DRAM before it is read into the L3. Because applications have different sized footprints and L3 miss rates, we computed a unique error rate for each benchmark. We based our evaluation on a raw soft error rate of 5000 FIT/Mbit, as in [11].

Figure 10 shows the computed error rate reduction for each benchmark using COP. We show results for COP with 8 bytes or 4 bytes of ECC per compressed block. In all cases, one byte of parity bits is used per code word, with the 8-byte version incorporating 8 (64,56) SECDED codes per block, and the 4-byte version using 4 (128,120) code words. As shown, the 4-byte version provides better reliability, with a 93% reduction in the error rate on average. The error rate reduction provided by COP-ER is also shown, and is nearly 100% in all cases, since COP-ER can correct all single-bit errors.

Using the same approach, we also compared the reliability of COP-ER and standard SECDED with an ECC DIMM. In this comparison, the only uncorrectable errors are multi-bit errors. Because COP-ER uses wider codes than standard (72,64) SECDED, results show that COP-ER's error rate is 6x that of an ECC DIMM approach. Compared to the unprotected case, however, both of these schemes provide high error coverage. Note that if we modeled some single events to cause (uncorrectable) single-word multi-bit errors, COP-ER would appear more reliable overall. If the probability of a single-word multi-bit error is proportional to the number of vulnerable bits, COP-ER benefits from using fewer bits than an ECC DIMM approach.

Table 3 shows the percent of blocks that are incompressible and contain code words across all benchmarks. To help the reader interpret the measured result, the equivalent number of blocks in a fully-used 8GB main memory is also shown. Remember that COP must keep incompressible aliases in the LLC and cannot write such blocks to DRAM. Only blocks with 3 and 4 code words are considered aliases, although the table shows other cases for completeness. Out of all of the applications studied, we observed only a single incompressible data block containing 3 code words and saw no blocks with 4 code words.

Table 3: Code words in incompressible data blocks

Number of code words	Percent of blocks	Equiv. 8GB mem. blocks
1	1.4%	1879048
2	0.005%	6710
3	0.000002%	3
4	0.0%	0

We also modeled the performance of COP and compared it to an approach that allocates space in memory to store ECC for all blocks, similar to Virtualized ECC [22]. The baseline implementation in this work (ECC Reg.) differs from the Virtualized ECC proposal in two ways. First, our baseline does not require ECC address translation since only a single ECC region is allocated, allowing a simple offset calculation to locate a block's ECC metadata. Second, our ECC region baseline uses a wider error-correcting code that can protect an entire block with a single code word. A wide (523,512) code is used to ensure a fair comparison with COP, since COP also uses wide codes. Since each 64-byte data block requires at least 11 bits for error protection, the contiguous ECC region is allocated with a 2-byte entry per data block to facilitate addressing. To improve the performance of this approach as well as for COP-ER, ECC metadata is cached in the L3. For all performance simulations, we simulated a 4-core version of the system in Table 1 with private L2 caches and a shared L3. For SPEC2006, we ran a copy of the of the same application on each core, while for PARSEC we ran the 4-threaded version of each program. All simulations therefore run 4 billion instructions in total. For all COP configurations, we assumed an additional decode/decompress latency of 4 cycles.

Figure 11 shows the performance results comparing COP and COP-ER to the baseline ECC region approach. For COP, performance is slightly degraded as a result of the increased memory latency due to decompression. The performance of COP-ER, which protects all data, is slightly worse than COP due to the extra memory accesses to retrieve check bits for incompressible blocks. Because COP-ER places less pressure on the memory system, it degrades performance much less

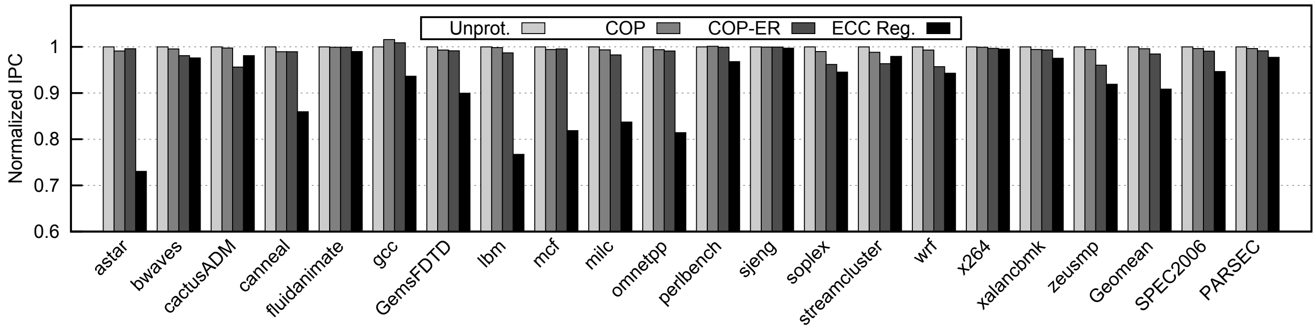


Figure 11: Performance of COP compared to an ECC region approach like Virtualized ECC. COP slightly increases the memory access latency, but puts less pressure on the memory system.

than the baseline. For the applications shown, COP-ER performs about 8% better than the ECC region baseline while providing the same error coverage.

Although COP-ER relies on an ECC region to protect incompressible blocks, it is able to significantly reduce the ECC storage overhead since its ECC region does not need to store ECC for compressible blocks. The amount of ECC storage required by each application was computed for both COP-ER and the baseline. Figure 12 shows the reduction in ECC storage space for each benchmark. In computing the storage overheads for COP-ER, we assume that an ECC entry is needed for any block that is ever incompressible in DRAM during execution (no entries are deallocated). As shown, COP-ER can reduce the space requirements by 80% on average.

5. Conclusion

Reliability has become an increasingly important requirement for main memory, even in cost-conscious designs. This paper describes how to achieve a 93% reduction in soft error rate over a baseline non-ECC DRAM system by first compressing and then adding ECC to each block stored in memory, avoiding the cost and energy overhead of ECC DIMMs and adding only a small amount of logic to the memory controller. The paper explores several low-complexity compression schemes and adapts them to the goal of freeing a minimal amount of space per block (just enough to accommodate the ECC bits), and proposes a very effective hybrid scheme that achieves 94% compressibility over the set of evaluated workloads. Since not all blocks are compressible, a novel zero-overhead coding scheme is used to distinguish compressed and protected blocks from incompressible blocks after they are read from main memory. To optionally extend ECC protection to incompressible blocks, a small region of main memory is used to store ECC for those blocks, along with a few bytes of uncompressed data from the original block, displaced to make room for a pointer that is used to find the matching ECC blocks whenever an incompressible block is read from main memory. The proposed approach can be naturally extended to provide even greater resilience (e.g. chipkill support), but a detailed exploration is left to future work.

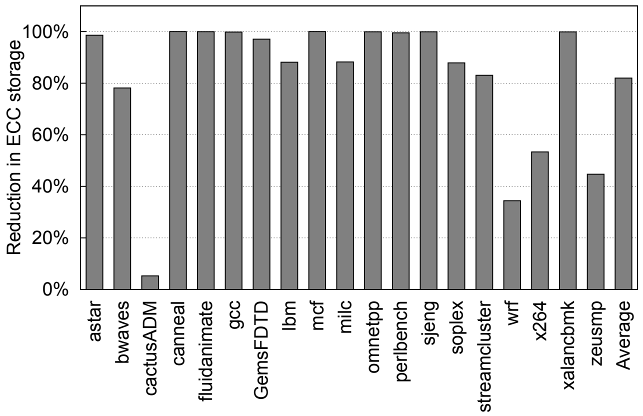


Figure 12: Reduction in ECC region size using COP-ER compared to the ECC region baseline.

Acknowledgment

This work was supported in part by the NSF (grants CCF-1016262, CCF-1116450 and CCF-1318298), and by DARPA (grant HR0011-12-0019). Nam Sung Kim has a financial interest in AMD and Samsung Electronics. Mikko Lipasti has a financial interest in Thalchemy Corp.

References

[1] A. Alameldeen and D. Wood, “Adaptive cache compression for high-performance processors,” in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, June 2004, pp. 212–223.

[2] American National Standard for Information Systems, “Coded character sets - 7-bit American national standard code for information interchange (7-bit ASCII),” March 1986.

[3] R. Baumann, “Soft errors in advanced computer systems,” *Design Test of Computers, IEEE*, vol. 22, no. 3, pp. 258–266, May 2005.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proc. 17th Int. Conf. on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2008, pp. 72–81.

[5] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11. New York, NY, USA: ACM, 2011, pp. 52:1–52:12.

[6] L. Chen, Y. Cao, and Z. Zhang, “Free ecc: An efficient error protection for compressed last-level caches,” in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, Oct 2013, pp. 278–285.

[7] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.

[8] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *Proc. 16th IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, Jan 2010, pp. 1–12.

[9] M. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[10] A. Jas and N. Touba, "Test vector decompression via cyclical scan chains and its application to testing core-based designs," in *International Test Conference*, Oct 1998, pp. 458–464.

[11] S. Li, K. Chen, M.-Y. Hsieh, N. Muralimanohar, C. D. Kersey, J. B. Brockman, A. F. Rodrigues, and N. P. Jouppi, "System implications of memory reliability in exascale computing," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. New York, NY, USA: ACM, 2011, pp. 46:1–46:12.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[13] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proc. 11th Int. Symp. on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 243–247.

[14] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. 46th Annual IEEE/ACM Int. Symp. on Microarchitecture*. New York, NY, USA: ACM, 2013, pp. 172–184.

[15] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques*. New York, NY, USA: ACM, 2012, pp. 377–388.

[16] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, June 2003.

[17] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[18] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *Proc. 20th International Symposium on High Performance Computer Architecture*, Feb 2014, pp. 638–649.

[19] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 76:1–76:11.

[20] H. Stracovsky, M. Espig, V. W. Lee, and D. Kim, "Reliability support in memory systems without error correcting code support," July 2013, US Patent 8,495,464.

[21] J. Suh, M. Manoochehri, M. Annavaram, and M. Dubois, "Soft error benchmarking of l2 caches with PARMA," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 85–96, June 2011.

[22] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proc. Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2010, pp. 397–408.

[23] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *Proc. 41st IEEE/ACM International Symposium on Microarchitecture*, Nov 2008, pp. 210–221.