Space-Oblivious Compression and Wear Leveling for Non-Volatile Main Memories

Haikun Liu, Yuanyuan Ye, Xiaofei Liao, Hai Jin, Yu Zhang, Wenbin Jiang, Bingsheng He[†]
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab/Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Email: {hkliu, yyuanye, xfliao, hjin, zhyu, wenbinjiang}@hust.edu.cn
[†]School of Computing, National University of Singapore
[†]Email: hebs@comp.nus.edu.sg

***Abstract*—Emerging *Non-Volatile Main Memory* (NVMM) technologies generally feature high density, low energy consumption and cost per bit compared to DRAM. However, the limited write endurance of NVMM poses a significant challenge of using NVMM as an substitute for DRAM. This paper proposes a space-oblivious compression and wear-leveling based memory architecture to improve the write endurance of NVMM. As memory blocks of many applications usually contain a large amount of zero bytes and frequent values, we propose a non-uniform compression encoding scheme that integrates *Zero Deduplication* with *Frequent Value Compression* (called ZD-FVC) to reduce bit-writes on NVMM. Moreover, we leverage the memory space available through compression to achieve an intra-block wear leveling policy. It rotates the writing positions of a compressed data block within the data's initial memory space, and thus enhances write endurance by balancing the bit-writes per NVMM cell. ZD-FVC can be integrated into the NVMM module and implemented entirely by hardware, without any intervention of *Operating Systems*. We implement ZD-FVC in Gem5 and NVMain simulators, and evaluate it with several programs from SPEC CPU2006. Experimental results shows that ZD-FVC is much better than several state-of-the-art approaches. Particularly, ZD-FVC can improve data compression ratio by 55% compared to *Frequent Value Compression*. Compared with the *Data Comparison Write*, ZD-FVC is able to significantly improve the lifetime of NVMM by 3.3X on average, and also reduces NVMM write latency by 31% and energy consumption by 19% on average.**

***Index Terms*—Non-Volatile Memory (NVM), Memory Compression, Wear-leveling**

I. Introduction

With the contiguously increasing memory footprint of big data applications, today's computing systems require large capacity of main memory for high performance data processing. The traditional *Dynamic Random Access Memory* (DRAM) technologies are facing scalability challenges in terms of memory density and static energy consumption [1]. Emerging *Non-Volatile Main Memory* (NVMM) technologies, such as *Phase Change Memory* (PCM) [2], *Resistive Random Access Memory* (ReRAM) [3], and 3D XPoint [4], offer high capacity, DRAM-like performance, near-zero standby power consumption, and byte addressability [5]. These good features make them to be potential substitutes for DRAM. They are expected to be commercially available in the near future.

Despite the promising advantages of NVMM against DRAM, NVMM generally features higher write latency and energy consumption, and much lower write endurance [6]. Among these drawbacks, the most important challenge is that a NVMM cell can only sustain 10^8 to 10^{10} write operations before it wears out, while DRAM cells are able to sustain about 10^{15} write operations [2], [7], [8]. To improve NVMMs' lifetime, a common approach is to reduce the bit-writes on NVMM cells. Another approach is to adopt wear leveling techniques [8], [9] to distribute write operations across NVMM cells uniformly.

There have been many studies on bit-write reduction of NVMM, such as DRAM caching [7], [10]–[12], write rationing [13], [14], *Data Comparison Write* (DCW) [15], *Flip-N-Write* (FNW) [16], and cache/memory compression [8], [17]–[19]. DRAM caching and write rationing schemes usually require OS-level support or special hardware extensions. For example, DRAM caching schemes often rely on hardware/software cooperated page access counting (costly) to migrate hot pages to the DRAM buffer. These schemes often result in additional software/hardware penalties, limiting the potential optimization on latency, energy efficiency, and endurance of NVMMs. The FNW-based schemes reduce the number of bit-writes by conditionally flipping the bits to be written if the proportion of updated bits is larger than 50%. However, FNW fails to exploit data redundancy to further reduce bit-writes on NVMM.

Memory compression technologies [20]–[22] have been studied for decades. Most proposals aim to save the space of main memory. A typical commercial example is IBMs MXT architecture [20], which is a hardware implementation of dictionary-based compression algorithm– LZ77 [23]. Almost all memory compression technologies face a challenging problem of addressing compressed memory blocks. To access a compressed memory block, IBMs MXT requires an additional memory access to retrieve the address of compressed data. Ekman and Stenstrom [21] propose to store the address metadata of a compressed memory block associated with each CPU TLB entry. *Linearly Compressed Pages* (LCP) [24], [25] simplify the addressing of compressed memory blocks by conditionally compressing cache lines to a fixed size.

Those mechanisms either increase memory access latency, or complicate architectural and OS-level memory management.

In this paper, we propose a *space-oblivious* NVMM compression architecture that is specifically designed for enhancing NVMM write endurance. As NVMM is able to offer large capacity of main memory, its lifetime is a more critical factor for commercial use. *Unlike the previous memory compression mechanisms that focus on saving memory space, our space-oblivious NVMM compression architecture is particularly designed for bit-write reduction and wear leveling.* The proposed architecture, which is integrated into the NVMM module, leverages compression and decompression engines to write/read 64-byte data blocks, which are typically a cache line size. This allows CPU to seamlessly communicate with the NVMM module. The major contributions of this paper are as follows.

- It has been observed that many applications' memory content often contains a large portion of zero bytes [18], [21], [26] and frequent values [27]–[33]. We propose a new compression encoding scheme that integrates *Zero Deduplication with Frequent Value Compression* (called ZD-FVC) to reduce bit-writes on NVMM. Particularly, because the zero blocks account for a significant proportion of frequent values, ZD-FVC elaborately encodes the zero and non-zero data blocks with 1-bit and 4-bit tags, respectively. Like the Huffman coding, this non-uniform encoding scheme significantly reduces the size of compression metadata and improves memory compression ratio.
- We propose an intra-block wear leveling scheme by leveraging the space available through compression. It rotates the writing positions of a compressed data block within the data's initial memory space, and thus further enhances NVMM write endurance by balancing the bit-writes on memory cells. Our fine-grained wear leveling scheme is orthogonal and complementary to previous page-level and row-level wear leveling schemes for NVMM lifetime extension.
- We propose a simple-yet-efficient NVMM architecture that enables compression and wear leveling entirely by hardware, invisible to CPU and OSes. We achieve this goal by carefully in-place storing the compressed data and metadata, and by utilizing the reserved bits of *error-correcting code* (ECC) to encode the tags for compression and wear leveling.

We implement our space-oblivious and lightweight memory compression architecture in Gem5 [34] and NVMain [35] simulators, and evaluate it with a wide range of benchmarks from SPEC CPU2006. The experimental results show that ZD-FVC is better than several state-of-the-art approaches, such as *Data Comparison Write* (DCW) [15], *Flip-N-Write* (FNW) [16], *Frequent Value Compression* (FVC) [27], *Frequent Pattern Compression* (FPC) [36], and *Base-Delta-Immediate Compression* (BDI) [37]. Compared to the classic FVC, ZD-FVC can improve data compression ratio by 55%. Compared to DCW, our approach is able to reduce bit-writes on NVMM by 15%, and significantly improves the lifetime of NVMM by 3.3X on average. Correspondingly, the NVMM access latency and energy consumption are also reduced by 31% and 19%, respectively.

The rest of this paper is organized as follows. Section II introduces the background and related work. Section III motivates our ZD-FVC based NVMM architecture. Section IV presents the design details of ZD-FVC. Section V describes the experimental methodology and results. We conclude in Section VI.

II. Background and Related Work

First, we introduce the existing typical memory compression algorithms. Second, we introduce the related work of enhancing NVMM write endurance by data compression and bit manipulation.

A. Cache and Memory Compression Algorithms

Several studies have shown that cache and memory data are highly redundant. Compression technologies can reduce data redundancy, and thus increase available cache/memory capacity, reduce read/write latency and power consumption, and improve write endurance [19], [26], [38]–[43]. However, these technologies lead to compression/decompression overhead during write/read operations. Thus, the compression algorithms should be simple and efficient. Here, we introduce several typical cache/memory compression algorithms.

Zero-Based Compression. *Dynamic Zero Compression* (DZC) [26] is proposed to encode a zero-valued byte with one bit, and thus reduces energy consumption when accessing the compressed zero bytes. *Zero-Content Augmented* (ZCA) cache [18] utilizes an address tag and validity bits to represent null cache lines. These compression schemes all utilize addition tag bits to represent zero blocks. Generally, when a memory block is partitioned into smaller size of sub-blocks, there are more opportunities to identify the zero sub-blocks for compression. However, the metadata storage overhead for encoding these zero sub-blocks also increases. Our zero deduplication scheme makes a tradeoff between data compression ratios and the metadata size to select an optimal data granularity for compression. Moreover, we in-place store compression metadata (a bitmap) with the compressed data, do not cause addition storage overhead to locate the zero and non-zero sub-blocks.

***Frequent Value Compression* (FVC) [27].** The basic idea of FVC is to encode frequent values in cache/memory with short codes. As a number of values (such as 0, -1, 1, and 2) are frequently accessed in cache/memory, FVC uses 3 bits (such as 000, 001, 010, and 011) to identify these frequent values. If data is not a frequent value, FVC uses a 3-bit code '111' to identify it and stores the raw data directly in cache or memory. Thus, if only seven frequent values are encoded, FVC needs 3 bits to encode each sub-block. The metadata can cause high storage overhead when the size of sub-block is small. Our ZD-FVC particularly exploits the observation that the zero values account for a significant proportion of the total frequent values, and designs a new encoding scheme to represent the zero values (1 bit) and non-zero values (4 bits), respectively. As a result, ZD-FVC significantly reduces the size of compression metadata and improves the memory compression ratio.

Frequent Pattern Compression **(FPC) [36].** FPC is proposed to compress data that satisfies some specific patterns. They can be encoded with less bits than the raw data. For example, a 64-bit zero value can be represented by a 3-bit code, and a long integer (e.g., 126) can be encoded with a 3-bit prefix and 1 byte of real value.

Base-Delta-Immediate Compression **(BDI) [37].** BDI compresses data by comparing it with a given base value and only storing the delta value. Observing that data within the same cache lines may have only a little difference, BDI represents sub-blocks within a cache line using a base value combined with a delta array. To achieve a higher compression ratio, BDI compresses the same cache line with different combinations of bases and deltas, and chooses the minimum size of compressed data as the output of BDI. Compared with BDI, our scheme is more simple for implementation, introducing less hardware overhead. Moreover, ZD-FVC can reduce the storage overhead of compression tags from 4 bits to 2 bits for each memory block. Thus, ZD-FVC often achieves higher compression ratio than BDI.

B. Bit-Write Reduction through Bit Manipulation

A number of studies have proposed to exploit bit manipulations to reduce bit-writes on NVMM, and thus save energy and extend NVMM's lifetime. Yang *et al.* propose *data comparison write* (DCW) [15], which only writes the changed bits on NVMM cells, and thus reduces bit-writes on NVMM. Cho *et al.* propose *Flip-N-Write* (FNW) [16] to reduce the bit-writes by conditionally flipping the bits when the proportion of changed bits in a word is larger than 50%. These schemes are orthogonal to cache/memory compression methods [26], [27], [36], [37], and thus are complementary to the compression approaches for further reducing the bit-writes on NVMM.

C. Bit-Write Reduction through Memory Compression

In the recent years, a number of compression-based approaches [19], [38]–[43] have been proposed to reduce bit-writes on NVMM, and thus improve write performance, energy efficiency, and endurance of NVMM. Deb *et al.* [40] propose several technologies to enable hardware-based memory compression, such as compression metadata management, data compressibility prediction, and permuted mapping scheme for compressed data blocks. Palangappa *et al.* [19], [41] propose compression-expansion (CompEx) coding, which integrates FPC/BDI with expansion coding to reduce write latency and energy consumption, and to improve write endurance of MLC/TLC NVMM. Xu *et al.* [42] propose to make a tradeoff between data compression ratio and additional storage overhead of compression metadata, and select an optimal encoding scheme to maximize the bit-write reduction. Guo *et al.* propose *dynamic frequent pattern compression* (DFPC) [43], which samples and analyzes the distribution of data values to be written, and discovers frequent patterns for compression during the execution of applications. Dgien *et al.* [38] propose to utilize FPC and DCW to reduce bit-writes on NVMM, and further exploits wear leveling to enhance the endurance of NVMM. Li *et al.* [39] propose an in-place compression architecture using BDI algorithm to reduce energy consumption of NVMM. García *et al.* [8] compose on-chip cache replacement, memory compression, and wear leveling techniques together to improve the lifetime of NVMM.

Overall, these studies all explore the existing cache/memory compression algorithms and their extensions to address the energy, latency, and endurance problems of NVMM. Our proposal integrates zero-deduplication compression with FVC elaborately through a novel memory encoding design. This simple-yet-efficient NVMM compression architecture can particularly exploit the relatively high proportion of zero bytes in memory blocks, and thus significantly reduces the compression metadata of FVC to improve memory compression ratio.

III. Motivation and Design

In this section, we conduct a set of experiments to analyze the memory content of programs in SPEC CPU2006, and then present data distribution of zero and other frequent values. The observations motivate our NVMM compression architecture.

A. Zero-valued Memory Blocks

Many studies have observed that memory and cache usually contain a large fraction of zero blocks [18], [21], [22], [26], [44]. The fraction of zero-valued data may exceed more than 50% of applications' total memory traffic. Our experimental results also validate this observation. Figure 1 shows the fraction of memory only containing zero bytes when the memory is partitioned into blocks of 1B, 2B, 4B, 8B, 16B, 32B, and 64B for several benchmarks in SPEC CPU2006. There are 55% and 51% zero-valued memory on average when the data sizes are 1B and 2B, respectively. Even 15% of 64B sized blocks are all zeros. This observation is also validated by other benchmarks such as Media-Bench, TPC-C, and SPECjbb 2000 [26], [36]. For several applications, such as gcc, milc, povray, and astar, the fraction of zero blocks increases significantly when the block size becomes smaller, implying that a small block size has a high potential to improve memory compressibility.

B. Optimal Block Size for Compression

For a 64-byte memory block, one can use a bitmap to identify whether a partitioned sub-block is zero, and then sequentially store non-zero blocks following the bitmap. In this paper, we call this zero-based compression mechanism as *Zero Deduplication* (ZD). For example, we can use 64 bits to encode and locate the zero bytes if the memory block is partitioned into 1B sub-blocks, or use 32 bits to encode 2B sub-blocks. Although using small sub-blocks potentially improves the compression ratio, the bitmap size increases

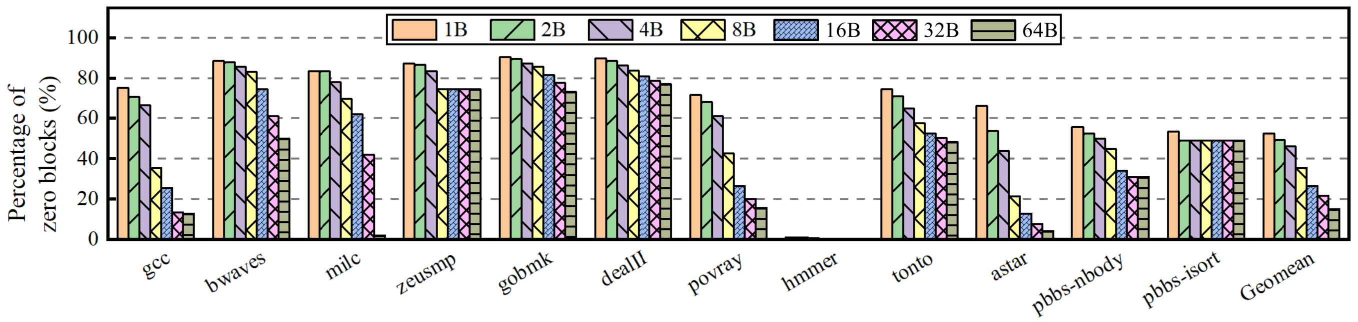


Fig. 1: The fraction of zero blocks in applications' memory traffic when the data is partitioned into blocks at the granularity of 1B, 2B, 4B, 8B, 16B, 32B, and 64B, respectively

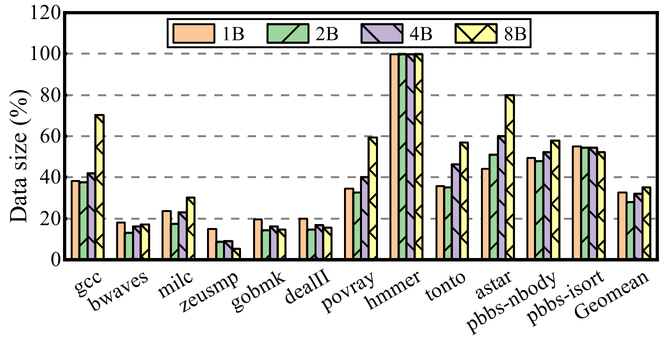


Fig. 2: The data size after using zero-based compression when the data block is at the granularity of 1B, 2B, 4B, and 8B, respectively

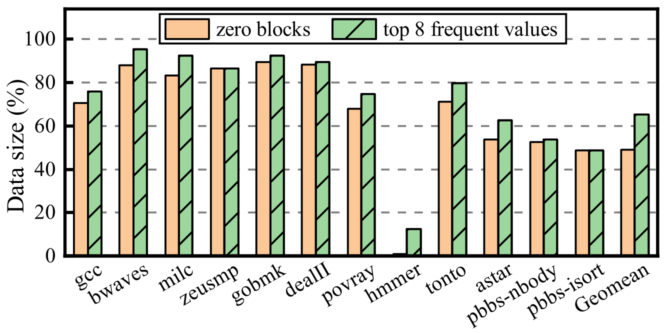


Fig. 3: The fraction of zero blocks and the top 8 frequent values in application's memory when the block size is 2B

accordingly. To determine the optimal block size for compression, we should make a tradeoff between the data compression ratio and the storage overhead of compression metadata. We conducted a set of experiments to study the optimal block size for compression. As the proportion of zero content is less than 30% when the block size is larger than 8B, Figure 2 only shows the proportions of data reduced by compression when the memory block is partitioned into 1B, 2B, 4B, and 8B sub-blocks, respectively. We find that the data size after compression is reduced by about 70% on average when the block size is set as 2B. As a result, in our prototype system, we compress memory blocks at the granularity of 2 bytes.

C. Frequent Value Compression

It has been observed that many applications' memory contain a considerable amount of frequent values [27]–[31], [45]. We can encode these frequently-used values to further compress the non-zero data. We count the top frequent values appeared in the memory traffic of SPEC CPU2006. Figure 3 shows that the top 8 frequent values account for about 70% of total memory traffic on average. The experimental results are similar to the observations of previous work [27]–[31], [45]. Since FVC decodes zero and non-zero values with the same sized tags, it may lead to high storage overhead of compression metadata. In our experiments, the top 8 frequent values are 0, 1, 2, 4, 3, -1, 5, and 8. However, the zero values account for a majority of frequent values. This motivates us to design a non-uniform encoding scheme to mitigate the storage overhead of compression metadata for encoding zero values. Because the zero values are encoded with 1 bit '0' at the Zero Deduplication stage, we use only 3 bits ('000–110') to encode the other frequent values, and use the remaining code '111' to represent the uncompressed data.

D. Differential Write

A write operation often leads to a number of bit flips, which refer to a switch of bit values either from 0 to 1 or vice-versa. Reducing bit flips can enhance the NVMM's lifetime and reduce write energy consumption. Although memory compression is an effective mechanism to reduce data to be written, it does not necessarily reduce the number of bit flips if the compression ratio is low. Because the compression may change the position of incompressible data blocks, and thus cause even more bit flips than the *data-comparison write* (DCW) scheme [15]. In such case, we exploit the differential write mechanism to further reduce the bit flips. When the compressed data is written to the NVMM, we compare the value of new data (possibly compressed) with the value of existing data in the NVMM, and only update the changed bits.

E. Wear Leveling

The above memory compression schemes often can significantly reduce the data size of partitioned memory blocks. Unlike the previous compression approaches for space saving, our NVMM compression architecture is space-oblivious in the sense that the saved memory space is only exploited for wear leveling. We can rotate the starting address of the

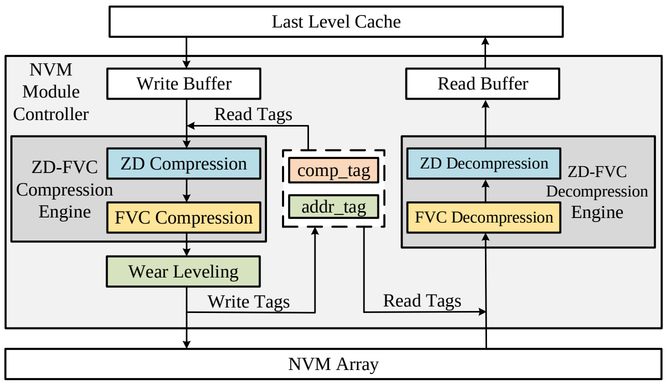


Fig. 4: An overview of our NVMM compression architecture

compressed data within the data's initial memory space, and thus balance bit writes on NVMM cells to improve NVMM write endurance. For example, if a 64-byte memory block can be compressed into 16 bytes, the compressed data can be alternately placed on four positions within the block's initial memory space.

In summary, the above observations motivate us to develop a space-oblivious compression for wear leveling in the following rationales. First, the memory layout for compression and wear leveling is based on fixed blocks. Through experimental profiling, we choose the optimal block size to be 2 bytes for the best compression ratio. Second, we elaborately integrate zero-based compression with frequent value compression to mitigate the storage overhead of compression metadata, and further leverage the partial write scheme to reduce bit-writes on NVMM. Finally, we exploit the memory space saved by compression to achieve fine-grained wear leveling, and thus extend the lifetime of NVMM.

IV. Memory Compression and Wear Leveling

Figure 4 illustrates the proposed space-oblivious NVMM compression architecture, which is integrated into the NVMM module. The ZD-FVC compression and decompression engines implement our two memory compression mechanisms: *Zero Deduplication* (ZD) and *Frequent Value Compression* (FVC). When the compressed data is written to the NVMM array, we implement a wear leveling policy by exploiting the memory space saved by compression. We encode 2-bit compression tags (*comp_tag*) and 2-bit wear leveling tags (*addr_tag*) in the field typically reserved for *error-correcting code* (ECC). The *comp_tag* represents whether and how a data block is compressed. The *addr_tag* represents the location (starting address) of the compressed data block within the data's initial memory space.

When a cache line is evicted from the on-chip last level cache, the data is first compressed by the ZD-FVC compression engine, and then stored in the NVMM array with the wear leveling scheme. Meanwhile, the *comp_tag* and *addr_tag* are encoded accordingly for future read operations. Upon a cache miss, the ZD-FVC decompression engine reads the data (possibly compressed) from the NVMM array according to the *addr_tag*, and decompresses the data according to the *comp_tag*. At last, the decompressed data is sent to the CPU cache.

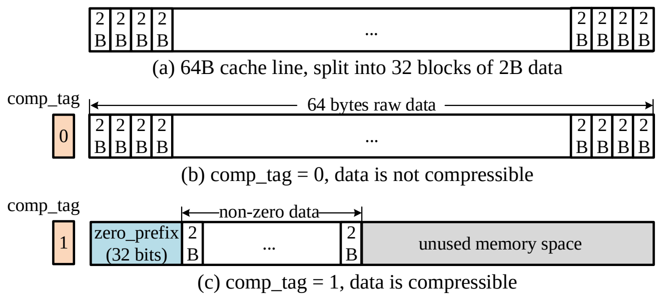

2 B
...
(a) 64B cache line, split into 32 blocks of 2B data
comp_tag
64 bytes raw data
0
(b) comp_tag = 0, data is not compressible
comp_tag
non-zero data
1
zero_prefix (32 bits)
unused memory space
(c) comp_tag = 1, data is compressible

Fig. 5: The data organization after using Zero Deduplication

In the following, we will describe the incremental implementation of our compression and wear leveling schemes step by step. To better understand ZD-FVC, we decouple the ZD-FVC algorithm into two stages: ZD and FVC.

A. Zero Deduplication

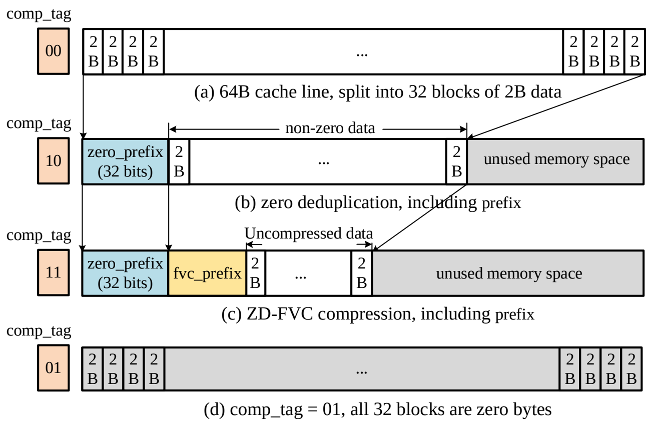
In this section, we introduce our *zero deduplication* (ZD) scheme. Figure 5 shows the data organization of a 64-byte cache line after using ZD. As suggested in Section III, we divide the cache line into 32 sub-blocks, and use 1 bit to represent each 2B sub-block. Totally, we need a 32-bit bitmap (called *zero_prefix*) to identify the zero-valued sub-blocks.

At first, all 2B sub-blocks are compared with zero in parallel. If they are equal to zero, the corresponding bit in the *zero_prefix* is set to 0. Otherwise, the corresponding bit is set to 1. If the number of zero bits in the *zero_prefix* is not larger than 2, implying the number of zero blocks in the cache line is too small, this 64B data block is not compressible with ZD because the *zero_prefix* spends 4 bytes. In this case, we set the *comp_tag* to 0 and store the 64B raw data directly in the NVMM, as shown in Figure 5(b).

If the number of non-zero blocks is larger than 2, the 64B data block is compressible with ZD. We first set the *comp_tag* to 1, and then store the 32-bits *zero_prefix* in the first 4B of the memory block, followed by non-zero sub-blocks sequentially, as shown in Figure 5(c). The position of zero and non-zero sub-blocks can be inferred by the 32-bit *zero_prefix*. As a result, we only need to store the non-zero sub-blocks in ZD. When the memory block is decompressed, we can simply decode the 32-bit *zero_prefix* to locate the non-zero sub-blocks because they are aligned on a 2-byte boundary.

B. Zero Deduplication Integrated with Frequent Value Compression

As presented in Section III-C, applications' memory often contains a considerable proportion of frequent values. We utilize *frequent value compression* (FVC) [27] to improve the compression ratio. We use 3 bits to encode the top 8 frequent values except zero, and thus further reduce the amount of data to be written. We integrate zero deduplication with frequent value compression together to implement a simple-yet-efficient compression mechanism, as shown in Figure 6.



comp_tag
00
2 B
...
(a) 64B cache line, split into 32 blocks of 2B data
10
zero_prefix (32 bits)
non-zero data
unused memory space
(b) zero deduplication, including prefix
11
fvc_prefix
Uncompressed data
(c) ZD-FVC compression, including prefix
01
(d) comp_tag = 01, all 32 blocks are zero bytes

Fig. 6: The data organization after using Zero Deduplication and Frequent Value Compression

To identify whether and how a memory block is compressed, we extend the *comp_tag* to 2 bits, which represent four cases, as described in Table I. (1) '00' represents the 64 bytes cache line is compressed by neither ZD nor FVC. (2) '01' represents the 32 sub-blocks are all zero values, as shown in Figure 6(d). (3) '10' represents that the data is only compressed by ZD, as shown in Figure 6(b). (4) '11' represents that the data has been compressed by both ZD and FVC, as shown in Figure 6(c). We use aligned FVC codes (*fvc_prefix*) to represent these frequent values.

Because all-zero 64-byte memory blocks account for about 15% of applications' memory traffic on average (see Figure 1), we particularly use the compression tag '01' to represent these null blocks. The all-zero memory blocks can be easily identified by testing the 32-bit *zero_prefix*. As a result, the space for storing the 32-bit *zero_prefix* and zero-valued sub-blocks are entirely saved. This further reduces bit-writes on the NVMM. We note that this simple-yet-efficient optimization may require additional logic circuits for other compression algorithms such as FVC, FPC, and BDI.

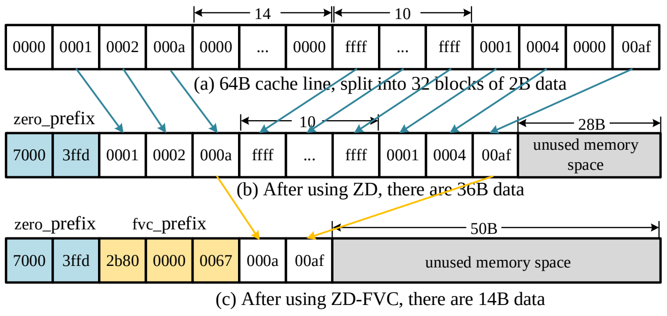
TABLE I: Codes and Descriptions of Comp_tag

Codes	Descriptions
00	uncompressed data
01	the whole 64-byte data is all zeros
10	compressed only by ZD
11	compressed by both ZD and FVC (ZD-FVC)

TABLE II: Frequent Value Encoding

Frequent Values (2B)	-1	1	2	3	4	5	8	Other
Encoding (3 bits)	000	001	010	011	100	101	110	111

We encode the top 7 frequently-used non-zero values with 3 bits, as shown in Table II. We note that the zero sub-blocks have been encoded by the *zero_prefix* at the zero deduplication stage, we can use the code '111' to identify other incompressible sub-blocks. For a frequent value, we only store its *fvc* code in the *fvc_prefix*. For an uncompressed value, we need to store the code '111' and its raw data after the *fvc_prefix*. As a result, we can encode all non-zero blocks and store their codes sequentially in the *fvc_prefix*. The uncompressed sub-blocks are stored sequentially after the *fvc_prefix*, as shown in Figure 6(c).



14
10
0000 0001 0002 000a 0000 ... 0000 ffff ... ffff 0001 0004 0000 00af
(a) 64B cache line, split into 32 blocks of 2B data
zero_prefix
7000 3ffd 0001 0002 000a ffff ... ffff 0001 0004 00af
28B
unused memory space
(b) After using ZD, there are 36B data
fvc_prefix
7000 3ffd 2b80 0000 0067 000a 00af
50B
unused memory space
(c) After using ZD-FVC, there are 14B data

Fig. 7: An example for illustrating ZD-FVC

If the memory space used by the *fvc_prefix* and the uncompressed data is smaller than the total size of non-zero data, we apply FVC to further compress the frequent values. The *comp_tag* is set to '11', and the *fvc_prefix* is stored after the *zero_prefix*, followed by the uncompressed raw data. If FVC can not reduce the size of non-zero blocks, the compressed data using ZD is directly stored in the NVMM array, and the *comp_tag* is set to '10' accordingly.

Here, we analyze the storage overhead of compression encoding in ZD-FVC, i.e., the total size of *zero_prefix* and *fvc_prefix*. For each zero sub-block, ZD-FVC only uses one bit to identify it. For each non-zero sub-block, ZD and FVC use 1 bit and 3 bits in the *zero_prefix* and *fvc_prefix*, respectively. Totally, ZD-FVC uses 4 bits to represent a non-zero block, and 1 bit to represent a zero block.

We can analytically compare ZD-FVC with the classic FVC. Assume the number of zero and non-zero 2B sub-blocks within a 64B memory block are a and b, respectively, and c blocks are not compressible with FVC. $a+4b$ bits are used to encode the zero and non-zero sub-blocks in ZD-FVC, while 96 bits $((64/2) \times 3)$ are required to encode the sub-blocks in FVC. We can derive that ZD-FVC has higher compression ratio than FVC if the condition in Equation 1 is satisfied.

$$a + 4b + 2 \times 8c < 96 + 2 \times 8c \quad (1)$$

where $a + b = 32$. We can deduce that ZD-FVC is better than FVC if the number of zero sub-blocks within a 64B memory block is larger than 11, i.e., the proportion of zero sub-blocks should exceed 34%. This requirement is satisfied by many applications, as shown in Figure 3.

In the following, we use an example to illustrate data compression with ZD-FVC, as shown in Figure 7. Each data block presents a hexadecimal value. In the first stage, ZD removes the zero-valued 2B sub-blocks, and uses only 1 bit to identify each zero block in the *zero_prefix*. For each non-zero sub-block, ZD sets the corresponding bit in the *zero_prefix* and places the raw data after the *zero_prefix*. For example, the first four sub-blocks ('0x0000', '0x0001', '0x0002', '0x000a') in the Figure 7(a) are represented by bits '0', '1', '1', and

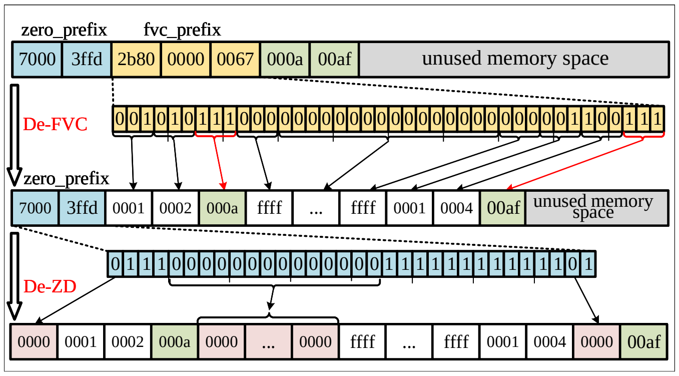


Fig. 8: An illustration of ZD-FVC decompression

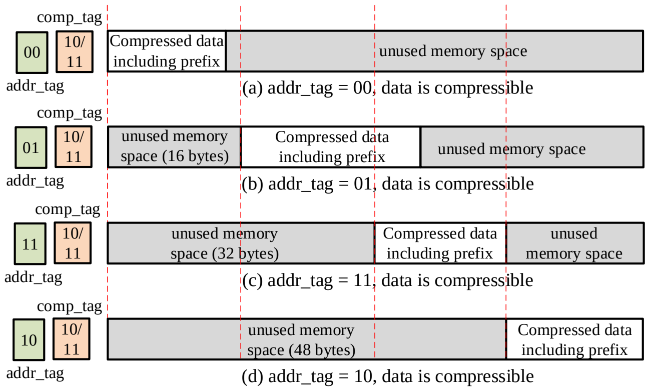

comp_tag
00
10/ 11
addr_tag
Compressed data including prefix
unused memory space
(a) addr_tag = 00, data is compressible
comp_tag
01
10/ 11
addr_tag
unused memory space (16 bytes)
Compressed data including prefix
unused memory space
(b) addr_tag = 01, data is compressible
comp_tag
11
10/ 11
addr_tag
unused memory space (32 bytes)
Compressed data including prefix
unused memory space
(c) addr_tag = 11, data is compressible
comp_tag
10
10/ 11
addr_tag
unused memory space (48 bytes)
Compressed data including prefix
(d) addr_tag = 10, data is compressible

Fig. 9: An illustration of our wear leveling scheme

'1', respectively. They correspond to '0111' ('0x7') in the *zero_prefix*, as shown in Figure 7(b). After using ZD, the 64-byte data is reduced to 36 bytes, including 4-bytes *zero_prefix* and 32-bytes non-zero data. The remaining 28 bytes memory space is saved.

In the second stage, we use FVC to compress the non-zero data. For example, the first four non-zero data ('0x0001', '0x0002', '0x000a', '0xffff') are encoded with codes '001', '010', '111', and '000' respectively. Putting these codes together, we get a bit string '001010111000', i.e., '0x2b8'. Thus, the first three characters of *fvc_prefix* is '2b8', as shown in Figure 7(c). After FVC is performed, the uncompressed sub-blocks are stored behind the *fvc_prefix*, such as '0x000a' and '0x00af'. Finally, the 64-byte data is compressed to 14 bytes, including 4-byte *zero_prefix*, 6-byte *fvc_prefix*, and 4-byte uncompressed data. As a result, ZD-FVC reduces bit-writes of 50-byte data.

Note that the described two stages are only illustrated for better understanding the ZD-FVC algorithm. Actually, ZD can be integrated with FVC in the hardware implementation. ZD-FVC only compares the data block with the zero and frequent values once, and the *zero_prefix* and *fvc_prefix* are generated concurrently. ZD-FVC does not require additional SRAM to store the non-zero data intermediately during the zero deduplication stage. It only needs a few 16-bit comparator and multiplexers for compression/decompression, incurring very limited hardware overhead.

C. Decompression of ZD-FVC

We illustrate the decompression of ZD-FVC using an example, as shown in Figure 8. At first, we should read the *comp_tag* to determine whether the memory block is uncompressed or zero-valued. If the *comp_tag* is '00', the data is directly fetched to CPU. If the *comp_tag* is '01', the ZD-FVC decompression engine generates a null memory block. If the *comp_tag* is '10', implying the FVC is not applied, we directly use De-ZD to decompress the data, as illustrated in Figure 8. The bits in the *zero_prefix* is checked in parallel to find out the 2B zero-valued sub-blocks and non-zero sub-blocks, which are fetched into the read buffer and sent to CPU. If the *comp_tag* is '11', we use De-FVC to decode the non-zero frequent values, as illustrated in Figure 8. From the *fvc_prefix*, the decompression engine can infer the number of sub-blocks compressed by FVC, and use the aligned 3-bit codes to decode the frequent values and uncompressed sub-blocks in place. In the following, the decompression of non-zero blocks is the same as the progress of De-ZD.

D. Wear Leveling

To exploit the memory space saved by ZD-FVC, we implement a wear leveling scheme to extend the lifetime of NVMM. As the compression ratio of ZD-FVC is as high as 4 on average in our experiments, we divide the 64-byte memory block into four sections evenly. We also use 2-bit *addr_tag* to locate the starting address of compressed data. When the *addr_tag* is '00', '01', '11', and '10', the compressed data is stored from the first, 17th, 33th, and 49th byte of the memory block, as shown in Figure 9(a), (b), (c), (d), respectively. When the data is not compressed, i.e, *comp_tag* = '00', the raw data are stored at the beginning of the memory block. When the 64-byte data is all zeros, i.e, *comp_tag* ='01', ZD-FVC does not store anything in the memory block.

In order to rotate the position for storing the compressed data, we design an address rotating algorithm, as described in Algorithm 1. The current data address (*addr_tag*) is determined by the value of *comp_tag*, the previous *addr_tag*, and the size of compressed data. The basic principle of our algorithm is to rotate the value of *addr_tag* among '00', '01', '11', and '10' iteratively. We set the progression order of the *addr_tag* as "00-01-11-10" to mitigate write wear of ECC bits. However, when the 64-byte data is uncompressed, the *addr_tag* must be set to '00' to guarantee that the 64-byte uncompressed data can be stored in place (line 2–4). When we get the starting address (line 5), we have to check whether the remaining memory space can hold the compressed data (line 7–20). If not, we roll back to prior position and check again, until we find a proper position to store the data. In this way, we can evenly distribute the number of bit-writes on NVMM cells, and thus extend the lifetime of NVMM.

Theoretically, our wear leveling algorithm supports any size (less than 64 bytes) of data rotating within a memory block. Thus, it can be adopted by many compression algorithms. However, compression algorithms with high compress ratios can exploit more space for wear leveling. Moreover, our hardware-based wear leveling scheme only use a 2-bit *addr_tag* to locate the starting address of data, without any involvement of OS or more complicated hardware-based address remapping mechanisms.

Algorithm 1 Address Rotating

```
1: function ROTATION($comp\_tag, addr\_tag, size$)
2:     if $comp\_tag$ = '00' then
3:         $addr\_tag \leftarrow$ '00';
4:         return $addr\_tag$;
5:     $addr\_tag \leftarrow addr\_tag + 1$;    //rotate to the next position
6:     while $True$ do
7:         if $addr\_tag$ = '00' then
8:             return $addr\_tag$;
9:         if $addr\_tag$ = '01' then
10:            if $size <= 48$ bytes then
11:                return $addr\_tag$;
12:            $addr\_tag \leftarrow addr\_tag - 1$; //roll back to the prior position
13:        if $addr\_tag$ = '11' then
14:            if $size <= 32$ bytes then
15:                return $addr\_tag$;
16:            $addr\_tag \leftarrow addr\_tag - 1$; //roll back to the prior position
17:        if $addr\_tag$ = '10' then
18:            if $size <= 16$ bytes then
19:                return $addr\_tag$;
20:            $addr\_tag \leftarrow addr\_tag - 1$; //roll back to the prior position
```

E. Discussion

Reversed space in ECC codes. For 64-byte cachelines or memory blocks, modern computer architectures often use additional 8 bytes to store the ECC codes. There are many ways to construct the ECC codes. For example, if we use Hamming codes that are widely-used in DRAM memories [46], a code of s bits can protect t bits information, where $2^{(s-1)} \geq s + t$. When we use a 8-bit code, we can correct single error and detect double errors for a data region as large as 120 bits. Thus, we only need $5 \times 8 = 40$ bits to protect a 512-bit memory block, and the remaining 24 bits can be used to store the compression and wear-leveling tags. On the other hand, a 60-bit BCH code can protect a data field up to 1023 bits from up to 6 errors [47]. This implies that a 512-bit memory block can be also protected by a 60-bit BCH code, and the remaining 4 spare bits can be used for storing the compression tags and wear-leveling tags. Overall, we can use some spare bits within ECC codes to encode the compression states, without compromising the capability of ECC. This avoids the additional storage overhead introduced by the compression and wear-leveling tags.

Write wear due to ECC bits. The ECC bits have a side effect on write endurance because they may be updated upon each NVMM write. For write-intensive applications, the ECC coding would wear out faster than the NVMM cells to be protected. As we use some reversed ECC bits to store the *addr_tag* and *comp_tag* bits, they also lead to write wear on the ECC bits. Recently, the Floating-ECC [48] and sliding-mode ECC architectures [49] have been proposed to circulate the writes of ECC bits across a cache line. We can seamlessly integrate these techniques with ZD-FVC to mitigate the write wear due to ECC bits.

Integrate with coarse-grained wear leveling schemes. Our fine-grained wear leveling scheme is only conducted within memory blocks at the 64-byte cacheline granularity. However, there are also "harmful" NVMM access patterns that write the same memory blocks frequently. In this case, our scheme is sub-optimal because it only targets at intra-block wear leveling by placing the compressed data in different sub-regions of a block. Our scheme is orthogonal and complementary to other coarse-grained wear leveling schemes, such as row level [50] and page level [7]. We can adopt these schemes seamlessly to spread frequent write operations uniformly across the entire NVMM capacity through a mapping table. On the other hand, there also have been some studies on extending NVMM lifetime by retiring the cells with permanent failures [51]–[54]. For example, *Error-Correcting Pointers* (ECP) [54] permanently encodes the locations of failed cells into a table and assigns other cells to replace them. These schemes are also complementary to our work to further improve the write endurance of NVMM.

V. Evaluation

In this section, we first present the experimental methodology for evaluating our NVMM compression architecture, and then study the effectiveness of ZD-FVC in terms of memory compression ratio, reduction of bit-writes, access latency, energy consumption, and lifetime of NVMM.

A. Experimental Methodology

Experimental setting. We simulate our NVMM compression architecture with Gem5 [34] and NVMain [35] simulators. The gem5 is a widely-used full-system architectural simulator for computer architecture research. NVMain is a cycle-accurate architectural NVM simulator. The system configuration is described in Table III.

TABLE III: System Configuration

CPU	out-of-order, 2 GHz, 8 cores
L1 cache	32 KB separated icache and dcache, 2 cycles
L2 cache	1 MB, 20 cycles
L3 cache	16 MB, 50 cycles
PCM	Capacity: 4 GB Controller: FRFCFS scheduler Bus Frequency: 400 MHz Timing (tCAS-tRCD-tRP-tRAS): 5-22-60-17 (cycles) Energy: 81.2 PJ/bit for read, 1684.8 PJ/bit for write

Benchmarks. We use twelve programs from SPEC CPU 2006 benchmark [55] and *Problem Based Benchmark Suite* (PBBS) [56] in our experiments. These applications' memory access traffic show different distributions of zero blocks and frequent values, as shown in Figure 1 and 3. We run each benchmark for 10 million instructions. The total memory traffic and read-to-write ratios are shown in Table IV.

Comparisons. We compare ZD and ZD-FVC with several state-of-the-art bit manipulation and memory compression

TABLE IV: Memory Traffic and Read-to-Write Ratios

Benchmarks	Total Memory Traffic (MB)	Read/Write
gcc	218.87	5.14
bwaves	810.98	1.19
milc	550.71	6.33
zeusmp	613.57	4.40
gobmk	113.69	1.15
dealII	7.37	1.04
povray	4.08	1.33
hmmer	141.80	1.01
tonto	3.45	1.21
astar	189.66	1.71
PBBS-nbody	356.34	1.64
PBBS-isort	347.97	1.01

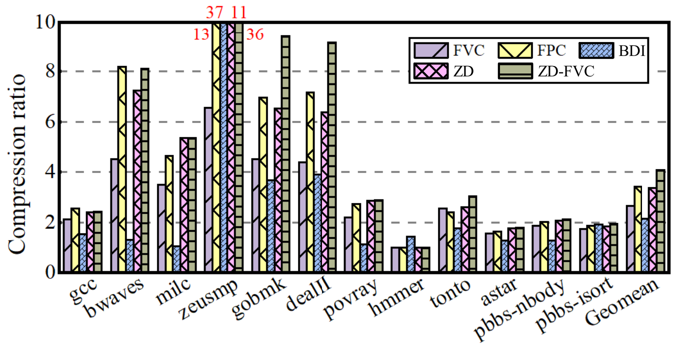


Fig. 10: Data compression ratios of different schemes

schemes, such as *Data Comparison Write* (DCW) [15], *Flip-N-Write* (FNW) [16], *Frequent Value Compression* (FVC) [27], *Frequent Pattern Compression* (FPC) [36], and *Base-Delta-Immediate Compression* (BDI) [37]. DCW is a typical differential write scheme that writes only modified bits in the NVMM array. FNW flips the data bits when the proportion of bits to be changed is larger than 50%. FVC compresses data by encoding the frequent values. FPC compresses data with some given patterns. BDI compresses data using a base value combining with an array of deltas. It chooses the minimum size of data compressed in parallel as its output. ZD and ZD-FVC all compress 2-byte sub-blocks in a cache line.

B. Memory Compression Ratio

Figure 10 shows data compression ratios of FVC, FPC, BDI, ZD and ZD-FVC, respectively. FVC, FPC, BDI, ZD, and ZD-FVC reduce the data size to be written on NVMM by 61.3%, 69.9%, 53.7%, 70.0%, and 75.0% on average, respectively. Correspondingly, the average compression ratios of FVC, FPC, BDI, ZD, and ZD-FVC are 2.6, 3.3, 2.2, 3.3, and 4.0, respectively. Compared with FVC, ZD-FVC improves the compression ratio by 53%. Particularly, the data compression ratio is significantly improved for bwaves, zeusmp, gobmk, and dealII.

The improvement of compression ratio is mainly attributed to two special designs. First, we particularly use the compression tag '01' to represent the all-zero memory blocks, which account for about 15% of applications' memory access traffic on average (see Figure 1). This mechanism significantly reduces the storage overhead of compression encoding. Second, we observe that the proportion of 2B zero sub-blocks is even larger than 70% of all frequent values in applications' memory (see Figure 3), and design a new encoding scheme to represent the zero sub-blocks and non-zero sub-blocks with 1 bits and 4 bits, respectively. This scheme further reduces the storage overhead of compression encoding.

C. Reduction of Bit-writes

Figure 11 shows the reduction of bit-writes for each application using DCW, FNW, FVC, FPC, BDI, ZD, and ZD-FVC, respectively, all normalized to DCW. Due to the high compression ratio and a differential write mechanism, ZD-FVC can reduce the bit-writes by 14.9%, 10.4%, 19.1%, and 7.5% on average compared with DCW, FNW, FVC, and FPC, respectively. For FVC, although the data size after compression is reduced, surprisingly, it causes even more bit-writes than DCW. This phenomenon is much clear for bwaves, gobmk, povray, and omnetpp, which contain a large proportion of zero-valued sub-blocks. Because compression usually changes the writing position of incompressible sub-blocks after compression, and thus can not exploit the differential write mechanism to write only changed bits. For example, if the difference between the new data and the old data is only one bit, DCW only writes the modified one bit. In contrast, compression may cause a significant difference between the compressed data and the old data (possibly compressed), resulting in a lot of bit-writes.

D. NVMM Access Latency

Memory compression/decompression usually lead to addition latencies when data is written to or read from NVMM. However, memory compression also offers opportunities to reduce the memory traffic for reading/writing the compressed data, and thus reduce the access latency of NVMM. The additional compression/decompression latencies have been discussed in the work [15], [16], [27], [36], [37]. To study whether and how ZD-FVC can reduce the NVMM read/write latencies, we properly model the compression/decompression latencies for different compression schemes based on the implementation complexity of hardware [37], as shown in Table V. For a write operation, we cautiously set the compression latency of ZD-FVC as 8 memory cycles, i.e., the sum of compression latencies of ZD and FVC. In practice, ZD can be integrated into FVC, incurring less compression latency. For a read operation, since the data may be compressed or not compressed, we model the additional read latencies with different values for the two different cases. When data (possibly compressed) is read from the NVMM array, we need one additional cycle to check whether the compression tag is '00'. If the data is not compressed, ZD-FVC decompression engine directly sends the data to CPU. These compression/decompression latencies are modeled as an addition to NVMM write and read latencies, respectively.

Figure 12 shows the normalized NVMM access latency of each benchmark with DCW, FNW, FVC, FPC, BDI, ZD, and ZD-FVC, respectively, all relative to DCW. The results

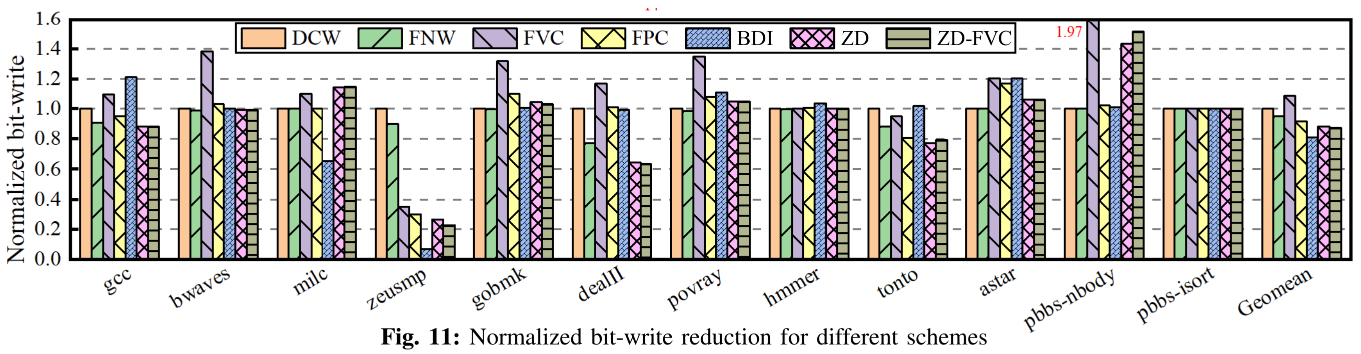


Fig. 11: Normalized bit-write reduction for different schemes

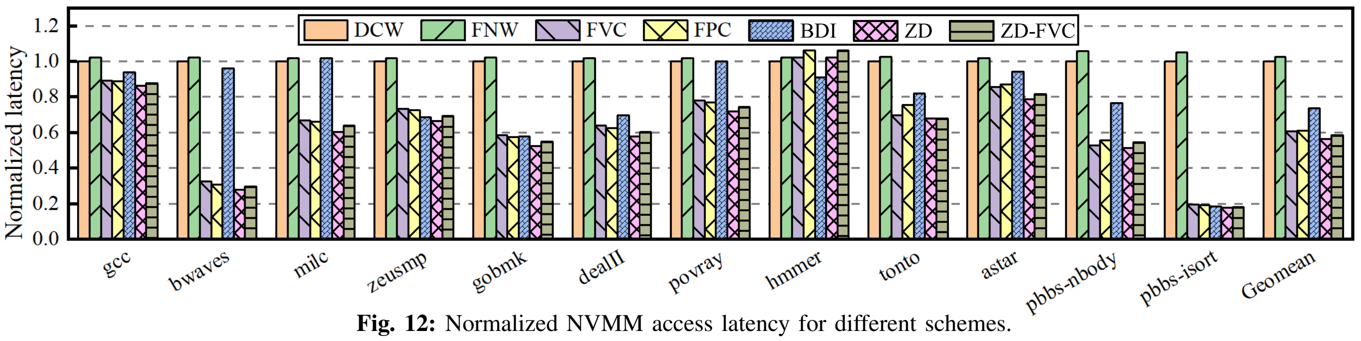


Fig. 12: Normalized NVMM access latency for different schemes.

TABLE V: Compression/Decompression Latencies Added to NVMM Writes and Reads

Schemes	Write (cycles)	Read-1[a] (cycles)	Read-2[b] (cycles)
DCW	2	0	0
FNW	4	1	2
FVC	4	1	5
FPC	8	1	5
BDI	8	1	2
ZD	4	1	5
ZD-FVC	8	1	7

[a]Data is not compressed. [b]Data is compressed.

are accumulated read/write latencies observed by the NVMM module. For FNW, the NVMM access latency is higher than DCW because FNW needs to perform bit-flipping when the fraction of bit flips is more than 50%. FVC, FPC, BDI, ZD, and ZD-FVC introduce additional compression and decompression latencies, however, they can be offset by the reduced memory access latencies due to memory compression. Thus, they all present lower memory access latencies compared to DCW and FNW. Although the compression and decompression latencies of ZD-FVC are higher than other schemes in our setting, ZD-FVC can still achieve lower NVMM access latency compared with other schemes because its compression ratio is much larger.

E. Energy Consumption

Although memory compression can reduce write energy consumption through bit-write reduction, data compression/decompression consume non-trivial energy. The energy consumption of compressing and decompressing a 64B memory block is modeled as 1.2 PJ and 2.1 PJ [57], respectively. Figure 13 shows the normalized energy consumption of NVMM for different applications with DCW, FNW, FVC, FPC, BDI, ZD, and ZD-FVC. All results are relative to DCW. ZD-FVC is able to reduce NVMM energy consumption by 19.1%, 14.9%, 19.6%, and 7.7% compared with DCW, FNW, FVC, and FPC, respectively. Because the energy consumption of NVMM write operations is almost 20 times higher than that of NVMM reads, the reduction of energy consumption is roughly consistent with the bit-write reduction. As ZD-FVC achieves a higher degree of bit-write reduction, it leads to much lower energy consumption compared to other schemes.

F. Lifetime

In the following, we analyze the impact of memory compression and wear leveling schemes on the lifetime of NVMM cells that is only confined to the memory region used by the benchmarks. Generally, NVMM lifetime is proportional to the available capacity, and is inversely proportional to the number of bit-writes. Memory compression can reduce the size of data to be written, and thus increase the available NVMM capacity to some extent. Assume the capacity of NVMM used by applications is C, the memory compression ratio is R, and the number of bit-writes on NVMM is N. The available capacity after compression becomes $C \times R$. Similar to previous works [7], [42], [50], we use Equation 2 to estimate the lifetime of NVMM cells used by the evaluated applications.

$$lifetime = \frac{C \times R}{N} \tag{2}$$

Figure 14 shows the lifetime extension for each benchmark with DCW, FNW, FVC, FPC, BDI, ZD, and ZD-FVC, respectively, all normalized to DCW. ZD-FVC can significantly improve the lifetime of NVMM by 3.3X, 3.1X, 74.2%, 18.4%, and 85.9% compared with DCW, FNW, FVC, FPC, and BDI, respectively. Moreover, ZD-FVC shows a 63.6X improvement

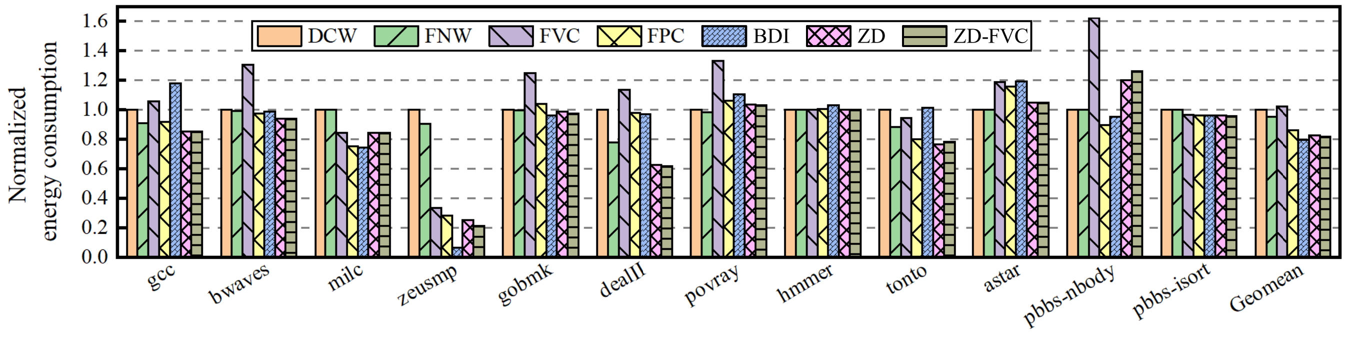


Fig. 13: Normalized energy consumption of NVMM for different schemes

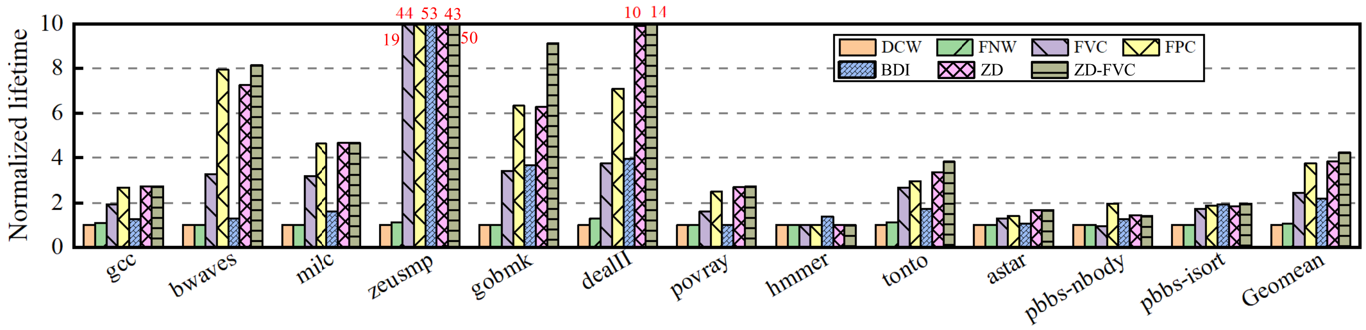


Fig. 14: Normalized NVMM lifetime for different schemes

of NVMM lifetime compared to a scheme in which the raw data is completely overwritten. Our approach is particularly effective for workloads whose memory content contains a large fraction of zero blocks, such as bwaves, zeusmp, gobmk, and dealII. The enhanced lifetime is mainly attributed to the high memory compression ratio and reduced bit-writes.

VI. Conclusion

In this paper, we propose a space-oblivious compression and wear-leveling based memory architecture to improve the write endurance of NVMM. We observe that memory blocks in many applications usually contain a large amount of zero bytes and frequent values. Thus, we propose a new memory compression scheme called ZD-FVC, which integrates zero deduplication with frequent value compression to reduce the data traffic written to NVMM. Moreover, we also leverage the available memory space through compression to implement an intra-block wear leveling scheme. It rotates the writes of compressed data blocks within the data's initial memory space, and thus enhances NVMM endurance by balancing the bit-writes per cell. We evaluate ZD-FVC with a wide range of benchmarks from SPEC CPU2006. Experimental results show that ZD-FVC performs better in compression ratio, latency, energy efficiency, and lifetime than several state-of-the-art approaches, such as DCW, FNW, FVC, FPC, and BDI.

Acknowledgments

This work is supported jointly by National Key Research and Development Program of China under grant No.2017YFB1001603, and National Natural Science Foundation of China (NSFC) under grants No.61672251, 61732010, 61825202, 61929103.

References

[1] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz, “Rethinking DRAM power modes for energy proportionality,” in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE, 2012, pp. 131–142.

[2] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, “Phase-change technology and the future of main memory,” *IEEE Micro*, vol. 30, no. 1, pp. 143–143, Jan. 2010.

[3] M. Chang, J. Wu, T. Chien, Y. Liu, T. Yang, W. Shen, Y. King, C. Lin, K. Lin, Y. Chih, S. Natarajan, and J. Chang, “19.4 embedded 1Mb ReRAM in 28nm CMOS with 0.27-to-1V read using swing-sample-and-couple sense amplifier and self-boost-write-termination scheme,” in *Proceedings of the 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC' 14)*. IEEE, 2014, pp. 332–333.

[4] Micron Technology, https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[5] C. J. Xue, G. Sun, Y. Zhang, J. J. Yang, Y. Chen, and H. Li, “Emerging non-volatile memories: opportunities and challenges,” in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS' 11)*. IEEE, 2011, pp. 325–334.

[6] G. Dhiman, R. Ayoub, and T. Rosing, “PDRAM: A hybrid PRAM and DRAM main memory system,” in *Proceedings of the 46th Annual Design Automation Conference (DAC' 09)*. ACM, 2009, pp. 664–469.

[7] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable High Performance Main Memory System Using Phase-change Memory Technology,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA' 09)*. ACM, 2009, pp. 24–33.

[8] A. A. García, R. de Jong, W. Wang, and S. Diestelhorst, “Composing lifetime enhancing techniques for non-volatile main memories,” in *Proceedings of the International Symposium on Memory Systems (MEMSYS' 17)*. ACM, 2017, pp. 363–373.

[9] S. Mittal, J. S. Vetter, and D. Li, "Writesmoothing: Improving lifetime of non-volatile caches using intra-set wear-leveling," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI (GLSVLSI' 14)*. ACM, 2014, pp. 139–144.

[10] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing (ICS' 11)*. ACM, 2011, pp. 85–95.

[11] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting program semantics to place data in hybrid memory," in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT' 15)*. IEEE, 2015, pp. 163–173.

[12] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, and R. Guo, "Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures," in *Proceedings of the International Conference on Supercomputing (ICS' 17)*. ACM, 2017, pp. 26:1–26:10.

[13] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong, "Mellow Writes: Extending Lifetime in Resistive Memories Through Selective Slow Write Backs," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA' 16)*. IEEE, 2016, pp. 519–531.

[14] S. Akram, J. B. Sartor, K. S. McKinley, and L. Eeckhout, "Write-rationing garbage collection for hybrid memories," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 18)*. ACM, 2018, pp. 62–77.

[15] B. Yang, J. Lee, J. Kim, J. Cho, S. Lee, and B. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'07)*. IEEE, 2007, pp. 3014–3017.

[16] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO' 42)*. ACM, 2009, pp. 347–357.

[17] P. M. Palangappa and K. Mohanram, "Castle: Compression architecture for secure low latency, low energy, high endurance NVMs," in *Proceedings of the 55th Annual Design Automation Conference (DAC' 18)*. ACM, 2018, pp. 87:1–87:6.

[18] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd International Conference on Supercomputing (ICS' 09)*. ACM, 2009, pp. 46–55.

[19] P. M. Palangappa and K. Mohanram, "Compex++: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVMs," *ACM Transactions on Architecture and Code Optimazation*, vol. 14, no. 1, pp. 10:1–10:30, Apr. 2017.

[20] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 271–285, Mar. 2001.

[21] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA' 05)*. IEEE, 2005, pp. 74–85.

[22] J. Dusser and A. Seznec, "Decoupled zero-compressed memory," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC' 11)*. ACM, 2011, pp. 77–86.

[23] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[24] 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 *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO' 13)*. ACM, 2013, pp. 172–184.

[25] G. Pekhimenko, T. C. Mowry, and O. Mutlu, "Linearly compressed pages: A main memory compression framework with low complexity and low latency," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques(PACT' 12)*. ACM, 2012, pp. 489–490.

[26] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*, 2000, pp. 214–220.

[27] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-33)*. ACM, 2000, pp. 258–265.

[28] Y. Zhang, J. Yang, and R. Gupta, "Frequent value locality and value-centric data cache design," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, 2000, pp. 150–159.

[29] G. Sun, D. Niu, J. Ouyang, and Y. Xie, "A frequent-value based PRAM memory architecture," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC' 11)*. IEEE, 2011, pp. 211–216.

[30] J. Yang and R. Gupta, "Frequent value locality and its applications," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, pp. 79–105, Nov. 2002.

[31] P. Zhou, B. Zhao, Y. Du, Y. Xu, Y. Zhang, J. Yang, and L. Zhao, "Frequent value compression in packet-based noc architectures," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference (ASP-DAC' 09)*. IEEE, 2009, pp. 13–18.

[32] L. Orosa and R. Azevedo, "Temporal frequent value locality," in *Proceedings of the 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP' 16)*. IEEE, July 2016, pp. 147–152.

[33] M. M. Islam and P. Stenstrom, "Characterizing and exploiting small-value memory instructions," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1640–1655, July 2014.

[34] Gem5, *http://www.gem5.org/Main_Page*.

[35] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, July 2015.

[36] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[37] 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 *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT' 12)*. ACM, 2012, pp. 377–388.

[38] D. B. Dgien, P. M. Palangappa, N. A. Hunter, J. Li, and K. Mohanram, "Compression architecture for bit-write reduction in non-volatile memory technologies," in *Proceedings of the 2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH' 14)*. ACM, 2014, pp. 51–56.

[39] Y. Li, H. Xu, R. Melhem, and A. K. Jones, "Space oblivious compression: Power reduction for non-volatile main memories," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI (GLSVLSI' 15)*. ACM, 2015, pp. 217–220.

[40] A. Deb, P. Faraboschi, A. Shafiee, N. Muralimanohar, R. Balasubramonian, and R. Schreiber, "Enabling technologies for memory compression: Metadata, mapping, and prediction," in *Proceedings of the 34th International Conference on Computer Design (ICCD' 16)*. IEEE, 2016, pp. 17–24.

[41] P. M. Palangappa and K. Mohanram, "Compex: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVM," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA' 16)*. IEEE, 2016, pp. 90–101.

[42] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, and C. Li, "Extending the lifetime of NVMs with compression," in *Proceedings of Design, Automation Test in Europe Conference and Exhibition (DATE' 18)*. IEEE, 2018, pp. 1604–1609.

[43] Y. Guo, Y. Hua, and P. Zuo, "DFPC: A dynamic frequent pattern compression scheme in NVM-based main memory," in *Proceedings of Design, Automation Test in Europe Conference and Exhibition (DATE' 18)*, 2018, pp. 1622–1627.

[44] F. Oboril, F. Hameed, R. Bishnoi, A. Ahari, H. Naeimi, and M. Tahoori, "Normally-off stt-mram cache with zero-byte compression for energy efficient last-level caches," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED' 16)*. ACM, 2016, pp. 236–241.

[45] J. Yang and R. Gupta, "Energy efficient frequent value data cache design," in *Proceeding of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*. IEEE, 2002, pp. 197–207.

[46] K. Furutani, K. Arimoto, H. Miyamoto, T. Kobayashi, K. Yasuda, and K. Mashiko, "A built-in Hamming code ECC circuit for DRAMs," *IEEE Journal of Solid-State Circuits*, vol. 24, no. 1, pp. 50–56, 1989.

[47] S. A. Aly, A. Klappenecker, and P. K. Sarvepalli, "On quantum and classical BCH codes," *IEEE Transactions on Information Theory*, vol. 53, no. 3, pp. 1183–1188, March 2007.

[48] H. Farbeh, H. Kim, S. G. Miremadi, and S. Kim, "Floating-ECC: Dynamic repositioning of error correcting code bits for extending the lifetime of STT-RAM caches," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3661–3675, 2016.

[49] X. Wang, M. Mao, E. Eken, W. Wen, H. Li, and Y. Chen, "Sliding Basket: An adaptive ECC scheme for runtime write failure suppression of STT-RAM cache," in *Proceedings of 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE'16)*. IEEE, 2016, pp. 762–767.

[50] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture(ISCA' 09)*. ACM, 2009, pp. 14–23.

[51] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (MICRO'10)*. IEEE, 2010, pp. 141–152.

[52] M. K. Qureshi, "Pay-as-you-go: Low-overhead hard-error correction for phase change memories," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. IEEE, 2011, pp. 318–328.

[53] D. H. Yoon, N. Muralimanohar, J. Chang, P. Ranganathan, N. P. Jouppi, and M. Erez, "Free-p: Protecting non-volatile memory against both hard and soft errors," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE, 2011, pp. 466–477.

[54] S. Swami, P. M. Palangappa, and K. Mohanram, "ECS: Error-correcting strings for lifetime improvements in nonvolatile memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 4, pp. 1–29, 2017.

[55] SPEC CPU 2006, https://www.spec.org/cpu2006.

[56] J. Shun, G. Blelloch, J. Fineman, P. Gibbons, A. Kyrola, K. Tangwonsan and H. V. Simhadri, "Problem Based Benchmark Suite," Retrieved from http://www.cs.cmu.edu/~pbbs/, 2012.

[57] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in MLC phase change memory," in *Proceedings of the IEEE International Symposium on High-Performance Comp Architecture (HPCA' 12)*. IEEE, 2012, pp. 1–10.