SimiEncode: A Similarity-based Encoding Scheme to Improve Performance and Lifetime of Non-Volatile Main Memory

Suzhen Wu‡, Jiapeng Wu‡, Zhirong Shen‡, Zhihao Zhang‡, Zuocheng Wang‡, Bo Mao‡✉
‡School of Informatics at Xiamen University, Xiamen, Fujian, China
✉Corresponding author: Bo Mao (maobo@xmu.edu.cn)

***Abstract*—Non-Volatile Memories (NVMs) have shown tremendous potential to be the next generation of main memory, yet they are still seriously hampered by the high write latency and limited endurance. In this paper, we first unveil via real-world benchmark analysis that the words within the same cache line showcase a high degree of similarity. We therefore present SimiEncode, a low-overhead and effective Similarity-based Encoding approach. SimiEncode relieves writes to NVMs by (1) generating a mask word with minimized differences to the words within a cache line, (2) encoding each word with the associated mask word by simple XOR operations, and (3) writing a single tag bit to indicate the resulting zero word after encoding. Our prototype implementation of SimiEncode and extensive evaluations driven by 15 state-of-the-art benchmarks demonstrate that, compared with existing approaches, SimiEncode significantly prolongs the lifetime and improves the performance. Importantly, SimiEncode is orthogonal to and can be easily incorporated into existing bit flipping optimizations.**

I. Introduction

Recently, the emerging Non-Volatile Memories (NVMs), such as Phase-Change Memory (PCM) and Spin-Transfer Torque RAM (STT-RAM), have received a surge of interest and been promising candidates of next-generation memory systems [4], [23]. Though they have the advantages of low leakage, high density, and high scalability, NVMs are still concerned by the inherent manufactural defects, in terms of high write latency and limited endurance, raising challenges when making attempts to deploy them in practice. For instance, a PCM cell can only endure about 10^8 writes (as opposed to 10^{16} writes in DRAM) and its write latency is 20X slower than that of DRAM (see Section II-A).

To address the problems, extensive efforts have been conducted to relieve the writes to NVMs via *bit flipping* or *data compression*. Specifically, the bit flipping [6], [9], [10], [15], [22] transforms an incoming word into another one that has fewer changed bits compared with the original word being stored in NVMs, hence reducing the bit writes to NVMs. Another method, the data compression [1], [8], [17], [18], [21] seeks to represent a word using a much shorter one with the aid of some data patterns maintained (e.g., the frequent patterns in Frequent Pattern Compression [1] and the base words in Base Delta Immediate [18]). However, most of existing studies either calls for additional reads [6], [9], [10], [15], [22] or is sensitive to the selected data patterns [1], [8], [17], [18], [21], hence are inefficient and cannot dynamically adapt to the workload changes.

On the other hand, our preliminary workload analysis reveals that the data of the regions within the same cache line exhibit a high degree of similarity. In detail, we conduct in-depth analysis using real-world benchmarks selected from PARSEC [3] and SPEC CPU 2017 [5], showing that the data similarity is *significant* (i.e., the similar regions of the same cache line take up a large proportion) and *prevalent* (i.e., all the evaluated benchmarks showcase the property of data similarity.

Based on the observation, we carve out a new path to prolong the lifetime and improve the performance of NVMs by exploiting *data similarity*. We therefore present SimiEncode, a new Similarity-based Encoding approach that strives to prolong the lifetime and improve the performance of NVMs. SimiEncode first partitions a cache line into many fixed-size *words*. It then carefully generates a *mask word* for each cache line, ensuring that the resulting mask word has minimized differences from the words within the same cache line. SimiEncode then encodes each word with the corresponding mask word through XOR operations and introduces one tag bit to indicate whether the word is encoded into a *zero word* (i.e., the word with all zeros). Once a zero word is identified, SimiEncode simply writes the corresponding tag bit, instead of the whole word, so as to reduce the bit writes to NVMs. Moreover, with the premier objective of reducing more bit writes, SimiEncode dynamically seeks the appropriate encoding granularity. In addition, SimiEncode is also orthogonal to prior studies (e.g., the bit flipping techniques [6], [9], [10], [15], [22]) and therefore can be easily incorporated into the existing optimizations to further prolong the lifetime and improve the performance of NVMs.

Our major contributions can be summarized as follows.

- We unveil via analyzing real-world benchmarks that data similarity is of prevalence and significance in practice.
- We design SimiEncode, a low-overhead encoding approach that leverages the word similarity to lessen the bit writes. SimiEncode also dynamically seeks the appropriate encoding granularity to minimize the bit writes to NVMs (Section III-A). In addition, when the large encoding granularity is used, SimiEncode further breaks the word into sub-words, to further reduce the bit writes.
- We implement the prototype of SimiEncode and conduct extensive performance evaluations with real-world and state-of-the-art benchmarks, showing that SimiEncode can significantly prolong the lifetime, reduce the read and write latencies, and save the energy consumption.

0x40404141	0x41414142	0x42424343	0x43434343	0x43434343	0x43434343	0x43434343	0x43434343

Fig. 1. An example of the first 32 bytes of a cache line where the last five words are the same, which is selected from the `x264` benchmark in SPEC CPU 2017.

II. Background and Motivations

A. Write problem in NVMs

NVMs are advantageous in architecting computer systems with the promising properties, but they still suffer severe manufactural shortcomings, such as expensive write energy, high write latency, and limited write endurance. For example, previous studies show that the write latency of PCM cells is about 20X slower than that of DRAM [7], and a PCM cell can merely sustain around 10^8 writes before getting stuck [29]. Besides, the write latency of STT-RAM is about 2X slower than that of DRAM [11].

The write energy and endurance of PCM further differ in the writing binary bits. For instance, PCM represents information through two interchangeable states of chalcogenide alloy, namely crystalline state (SET) and amorphous state (RESET). The SET operation (i.e., writing the bit '1') calls for more power consumption, since the operated PCM cell should be heated above 300°C but below 600°C over a period of time; while the RESET operation (i.e., writing the bit '0') is more detrimental to the PCM cells, since the operated PCM cell should be heated above 600°C hastily during the RESET operation [23].

The write asymmetric also differs in different NVMs. For example, writing '0' in a PCM cell is more detrimental to endurance than writing '1' [27], while writing '1' in a STT-RAM cell leads to a higher bit error rate than writing '0' [28]. Thus, to prolong the lifetime and improve the performance of NVMs, a straightforward approach is reducing the bit writes, which is used by existing studies.

B. Workload characteristic

Knowing the workload characteristics is important for storage design. Previous studies [1], [18] have pointed out that applications may generate redundant data patterns, including zeros (commonly used to initialize data) and repeated values (e.g., adjacent pixels that have the same color). This phenomenon drives us to investigate the *data similarity* within the same cache line. We select 10 benchmarks from SPEC CPU 2017 [5] and another 5 benchmarks from PARSEC [3] for analysis. Figure 1 shows an example of the first 32 bytes of a cache line selected from the `x264` benchmark in SPEC CPU 2017, in which the 32 bytes are divided into eight words (4 bytes per word) and the last five words are the same.

We then formalize the analysis. Suppose that the size of a cache line is 64 bytes. We divide a cache line into 16 equal-sized words (4 bytes per word), namely $\{W_1, W_2, \cdots, W_{16}\}$. Given two words W_i and W_j, we can simply use $W_i \oplus W_j$ to learn their similarity, where $\oplus$ denotes the exclusive-or (XOR) operation. We say the two words W_i and W_j are more similar once the resulting $W_i \oplus W_j$ contains more zero bits.

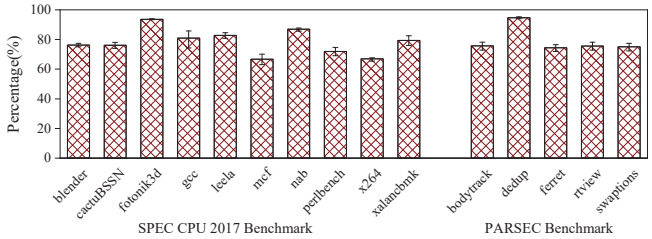

Fig. 2. Similarity analysis of the real-world benchmarks.

Hence, to characterize the data similarity existed in a cache line, we define the *similarity degree* of the i-th word (denoted by s_i, where $1 \le i \le 16$) to the other 15 words within the same cache line, which can be calculated through Equation (1):

$$s_i = \frac{\sum_{1 \le j \ne i \le 16} \theta(W_i \oplus W_j)}{\sum_{1 \le j \ne i \le 16} l}, \tag{1}$$

where θ is a function to count the number of resulting zero bits in a word, and l represents the word size. Finally, the average word similarity of a cache line can be simply calculated through Equation (2):

$$\bar{s} = \frac{\sum_{1 \le i \le 16} s_i}{16}. \tag{2}$$

Figure 2 presents the resulting word similarity of the 15 benchmarks. Given a benchmark, we obtain the word similarity for each cache line, and calculate the average word similarity as well as the maximum and minimum values across all the cache lines. We observe that the words within the same cache line exhibit a high degree of data similarity across all the 15 benchmarks. Specifically, the similarity degrees range from 66.7% (`mcf` benchmark) to 94.7% (`dedup` benchmark) and finally reach 77.2% on average, as shown in Figure 2. Moreover, the word similarity within cache lines of each benchmark is rather stable, which inspires us to design a new approach based on the identified word similarity for eliminating the bit writes to NVMs.

C. Motivation

The emergence of NVM devices such as 3D-Xpoint leads to significant changes to the storage hierarchy in personal devices, cloud, and high-performance platforms. But writes still significantly affect the energy, latency and endurance of NVMs. The existing studies use bit flipping by comparing the new and old cache lines, or data compression to reduce the bit writes to some extent [6], [8], [18], [22], [24]. However, most of these schemes ignore the significant word similarities within cache lines, which can be utilized to further reduce the bit writes to NVMs. Only BDI [18] utilizes the word similarity when the beginning parts of the words are the same. In a word, the word similarity within a cache line is still not fully exploited to reduce the bit writes.

Motivated by the above observations and the importance to address the write issues in NVMs, we present SimiEncode, an encoding approach to explore word similarity for prolonging

Algorithm 1 Encoding procedure of SimiEncode

Input: k (number of words of a cache line), l (size of a word)
Output: M (the mask word), Coded words $\{W'_1, W'_2, \cdots, W'_k\}$

1: Divide a cache line into k words $\{W_1, W_2, \cdots, W_k\}$
2: Initialize U where $U[j] = 0$ for $1 \leq j \leq l$
3: // Examine the bit distribution of the k words
4: **for** $1 \leq i \leq k$ **do**
5: **for** $1 \leq j \leq l$ **do**
6: $U[j] = U[j] + W_i[j]$
7: **end for**
8: **end for**
9: // Determine the mask word
10: **for** $1 \leq j \leq l$ **do**
11: **if** $U[j] > \frac{k}{2}$ **then**
12: $M[j] = 1$
13: **else**
14: $M[j] = 0$
15: **end if**
16: **end for**
17: // Encode the k words
18: **for** $1 \leq i \leq k$ **do**
19: $W'_i = W_i \oplus M$
20: **end for**

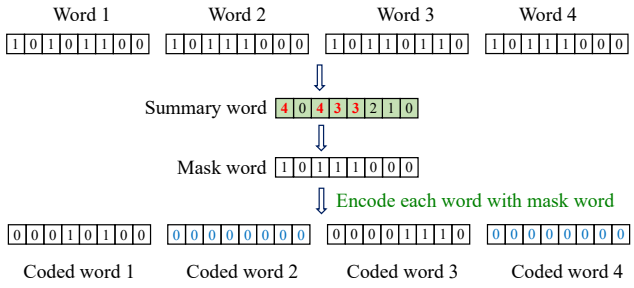

Fig. 3. An encoding example of SimiEncode.

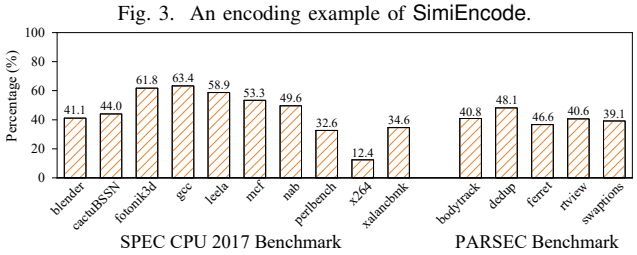

Fig. 4. Percentage of zero words in real-world benchmarks.

the lifetime and improving the performance of NVMs. The *main idea* behind SimiEncode is to look for a *mask word* for each cache line that has the smallest Hamming distance from the words in the cache line, thus generating as many zero words as possible. SimiEncode then uses tag bits to represent the generated zero words and writes the tag bits to NVMs (as opposed to flushing the zero words), so as to reduce the bit writes.

III. Design of SimiEncode

In this section, we first outline the encoding workflow in SimiEncode to exploit the word similarity. Then we present the write and read workflows of SimiEncode, followed by descriptions of the encoding and decoding logic. The applicability of SimiEncode is discussed at the end of this section.

A. Encoding workflow

The first step in the encoding workflow is to find the mask word with the minimum differences among the k words within the same cache line, and then apply the encoding method on the words within the cache line.

Mask word: In order to generate as many zero words as possible to reduce bit writes, generating the mask word requires careful consideration. The simplest method is picking a word from the cache line (e.g., the first word) to be a mask word which is used in the BDI scheme [18], but our evaluation results indicate that the selected word is not sufficiently similar to the other words. The other method is traversing all words and calculating the similarity of each word to the other remaining words. The word with the highest similarity among all words will be selected as the mask word, incurring high time overhead and $O(n^2)$ time complexity.

In this paper, we use Hamming distance to generate a mask word such that the Hamming distance between the mask word and the words in a cache line is the smallest, thus the mask word is the most similar to the words overall. We consider various methods to calculate similarity, but not all of them are appropriate, such as Cosine distance, Pearson distance and Jaccard distance, which are relatively complicated and not effective. When the data is represented in binary, Manhattan distance and Euclidean distance are equivalent to Hamming distance. If the goal is solely to generate the most zero words, it seems that the word with the most occurrences should be selected as the mask word. However, when the word can be divided into sub-words (see Sub-word in III-B), this method is ineffective. Therefore, in SimiEncode, we use Hamming distance to generate mask word.

Encoding procedure: Algorithm 1 elaborates the detailed procedure of encoding a cache line in SimiEncode. SimiEncode first partitions a cache line into k equal-sized words with the word size of l (Line 1, suppose that the size of a cache line is divisible by k). It then records the bit distribution of the k words by directly adding them and generates a *summary word* U (Lines 4-8). SimiEncode then scans each bit of U to establish the mask word M: if the value '1' at a position is dominated (i.e., more than $\frac{k}{2}$), the value of the mask word at this position is set as '1'; otherwise, it is set as '0' (Lines 10-16). SimiEncode finally encodes each word by XORing it with the mask word and resembles the resulting coded words into a new coded cache line (Lines 18-20). We can readily deduce that the computation complexity of Algorithm 1 is $O(kl) = O(1)$, where kl is the size of a cache line.

Figure 3 depicts an encoding example of SimiEncode. It finally generates two zero words (i.e., Coded word 2 and Coded word 4). We further select 15 benchmarks from SPEC CPU 2017 [5] and PARSEC [3] to validate the effectiveness of Algorithm 1. Figure 4 indicates that SimiEncode can encode 43.8% of the words of a cache line into zero words on average when the word size is 4 bytes.

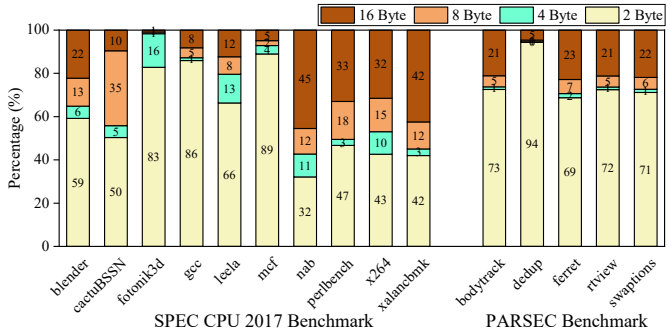

Fig. 5. The proportion of zero words generated by different encoding granularities.

TABLE I
ENCODING GRANULARITY AND THE ADDITIONAL BIT WRITES.

Prefix	Encoding granularity (bytes)	Number of additional bits
00	2	$18+\frac{h}{2}$
01	4	$34+\frac{h}{4}$
10	8	$66+\frac{h}{8}$
11	16	$130+\frac{h}{16}$

Encoding granularity: Matching the encoding granularity of the encoding scheme to the size of the data elements in the cache line has a considerable effect on the effectiveness of the encoding scheme. For example, the cache line with all integer numbers is better to choose a different encoding granularity from the one with all floating-point numbers. If the encoding granularity is not matched with the size of the underlying data elements, it will fail to produce abundant zero words and thus reducing the effectiveness of the encoding scheme.

Specifically, if the encoding granularity is smaller than the data element in the cache line, SimiEncode fails to produce zero words, potentially resulting in more writes as SimiEncode requires to write additional mask word and tag bits. Conversely, if the encoding granularity is larger than the underlying data element, SimiEncode suffers two drawbacks: fewer number of zero words after encoding and larger size of the mask word to be written.

In order to facilitate the finding of the most appropriate encoding granularity, SimiEncode provides four options of the encoding granularity: 2-byte, 4-byte, 8-byte, and 16-byte. Figure 5 shows the distribution of the different encoding granularities that generates the proportion of zero words in SimiEncode. SimiEncode traverses all the four encoding granularities and selects the one that generates the largest proportion of zero words. SimiEncode also employs a prefix with two bits to indicate the encoding granularity finally selected. Table I shows the prefix, the corresponding encoding granularity, and the size of the additional information (i.e., prefix, mask word, and tag), where h denotes the size (in unit of byte) of a cache line.

B. Write and read workflow

Write procedure: Figure 6 illustrates the write procedure in SimiEncode. SimiEncode first generates a mask word based on the k words within the same cache line (Step ①, where $k = 8$). It then encodes the cache line by XORing each word with

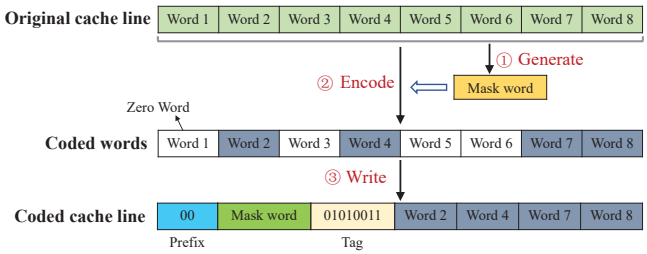

Fig. 6. Write procedure of SimiEncode.

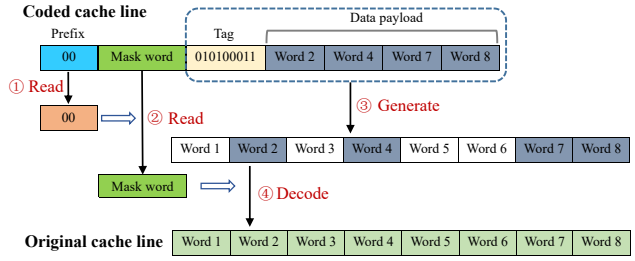

Fig. 7. Read procedure of SimiEncode

the mask word (Step ②). To further reduce the bit writes after generating zero words, SimiEncode maintains an additional *tag* along with the coded cache line. The tag is composed of k bits, where the i-th bit is set as '0' once the i-th coded word is identified as a zero word (where $1 \leq i \leq k$). For example, as Word 1 in Figure 6 is a zero word, the first bit of the tag is set as '0', while the second bit of the tag is marked as '1' since Word 2 is a non-zero word. When writing a coded cache line, SimiEncode simply writes the prefix, the mask word of this cache line, the corresponding tag, and the non-zero coded words to NVMs (Step ③).

Read procedure: Figure 7 illustrates the read procedure in SimiEncode. When reading a cache line, SimiEncode first identifies the encoding granularity (i.e., the word size) based on the prefix values according to Table I (Step ①). Suppose that the encoding granularity is l. It then extracts the mask word (Step ②) and recovers each coded word by scanning each tag bit: if the i-th tag bit is '0' (where $1 \leq i \leq k$), SimiEncode is aware that the i-th coded word is a zero word and generates a l-bit zero word directly; otherwise, SimiEncode reads the subsequent l bits from the data payload to resemble the i-th coded word (Step ③). For example, after identifying that the first bit of the tag in Figure 7 is '0', SimiEncode generates a zero word as Word 1; SimiEncode then reads the second bit of the tag which is '1', and retrieves the first l bits from the payload to serve as Word 2. SimiEncode finally recovers the original words by XORing each coded word with the mask word (Step ④).

Cutting down more tag bits: SimiEncode further reduces more bit writes through cutting down the additional information introduced by data encoding. We find that the resulting *zero cache line* after encoding (i.e., the cache line with all zero bits) occupies a considerable proportion of the coded cache lines. Figure 8 shows that the percentage of the zero cache lines encoded by SimiEncode reaches 23.8% on average for

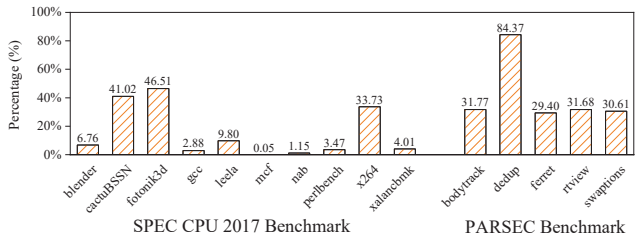

Fig. 8. Percentage of zero cache lines encoded by SimiEncode for real-world benchmarks.

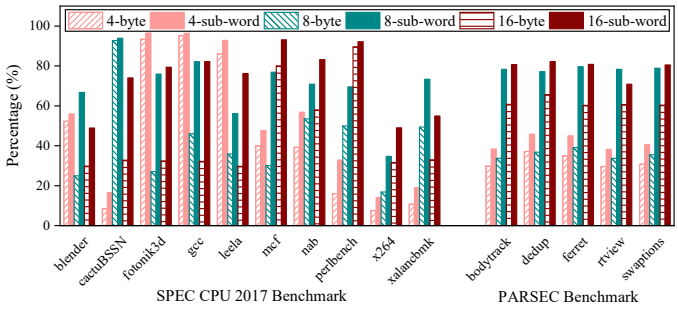

Fig. 9. Percentage of generated zero words for different encoding granularities and sub-words.

15 benchmarks. In this case, SimiEncode introduces one tag bit for a zero cache line to indicate whether a coded cache line is a zero cache line after encoding. If it is, SimiEncode directly writes the tag bit of the zero cache line, the prefix, and the mask word to NVMs. Therefore, compared to the original design of SimiEncode that needs to record numerous tag bits, this optimized method can dramatically reduce the bit writes for zero cache lines, hence further prolonging the lifetime and improving the performance of NVMs.

Sub-word: SimiEncode now identifies and eliminates zero bits at the word granularity. This approach, however, may have poor effect when the word size increases, even when the words in the cache line are highly similar. Apparently, it is inefficient to identify zero words with the same size as the encoding granularity. To effectively exploit the word similarity in the cache line, SimiEncode further divides a coded word into a number of fixed-size sub-words and examines the zero bits at the sub-word granularity. If the sub-word size is too large, the number of identified zero words will be low. Conversely, if the sub-word size is too small, more tag bits are needed. In SimiEncode, the sub-word size is set to 2 bytes, the same as the minimum encoding granularity employed by SimiEncode. Figure 9 shows the percentage of generated zero words using 4, 8, and 16 bytes as the encoding granularity and the percentage of generated zero words by further breaking the coded word into sub-words. The number of zero words is increased by 8.4%, 32.5%, and 24.8%, respectively, when using 4, 8, and 16 bytes as the coding granularity with 2-byte sub-words.

C. Encoding and decoding logic

In this section, we elaborate the design of encoding and decoding logic. Figure 10 shows the encoding logic consists of 5 separate encoder units, including 4 different granularity

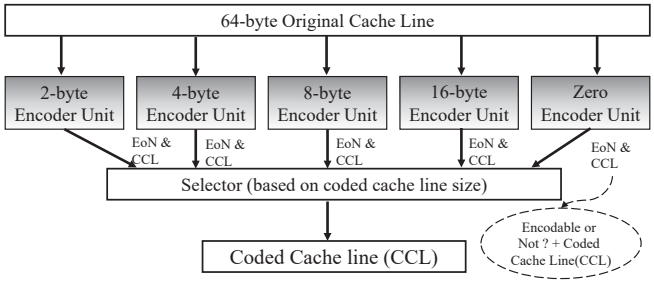

Fig. 10. The encoding logic consists of 5 separate encoder units.

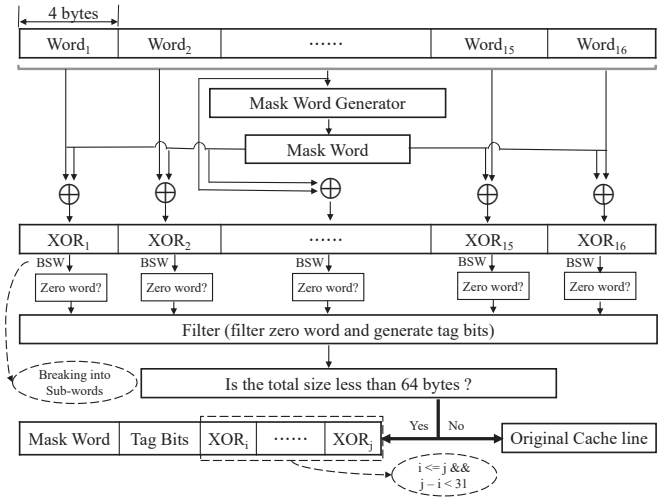

Fig. 11. Implementation of a 4-byte granularity encoder unit to encode a 64-byte cache line.

encoder units and a zero encoder unit. Each encoder unit takes a cache line as inputs and outputs after determining whether the cache line can be coded by this unit. If the cache line can be coded, the unit outputs the coded cache line. Otherwise, the original cache line is the output. Therefore, we need an additional bit to indicate whether the cache line is raw or coded. All encoder units can be executed in parallel. The Selector logic is used to determine whether the 5 encoder units can be successfully coded for the cache line. If multiple encoder units are available, the Selector selects the coded cache line with the smallest size.

Figure 11 illustrates the implementation of a 4-byte granularity encoder unit to encode a 64-byte cache line. The encoder divides the cache line into 16 4-byte words ($Word_1$, $Word_2$, ... , $Word_{16}$). The Mask Word Generator generates a mask word based on these 16 words, and then encodes each word with the mask word by XOR operations. The resulting values (XOR_1, XOR_2, ..., XOR_{16}) are further divided into 2 2-byte sub-words. The Filter logic sequentially assigns a tag bit to each sub-word to indicate whether the sub-word is a zero word or not. If a sub-word is a zero word, the Filter assigns a '0' tag bit and filters out the sub-word. Otherwise, the Filter assigns a '1' tag bit. If the total size of the coded cache line is less than 64 bytes, the coded cache line can be stored as a mask word, tag bits and a set of non-zero sub-words. Otherwise, the coder unit fails to code.

TABLE II
SYSTEM CONFIGURATIONS.

Processor and Cache	
CPU	4 cores x86-64 processes, 3.2 GHz
L1 I/D cache	private, 32 KB/core, 8-way, 64 bytes per cache line
L2 cache	private, 256 KB/core, 8-way, 64 bytes per cache line
L3 Cache	shared, 6 MB, 12-way, 64 bytes per cache line
PCM-based Memory	
Capacity	4 GB, 1 channel, 1 rank, 2 banks
Memory controller	FCFRFS
Read Latency	100 ns
Set latency	37.5 ns for each 32-bit word
Reset latency	12.5 ns for each 32-bit word

The decoding logic is simplistic. For a coded cache line, all the XORed words are first restored according to the tag bits. Then, these XORed words are decoded by executing XOR operations with the mask word to restore the original cache line and returned to the upper layer.

D. Discussion

SimiEncode is a general design and can be applied for different NVMs with the diversity of write asymmetric. For example, the characteristics of STT-RAM indicate that writing the bit '1' induces a higher bit error rate than writing the bit '0'. Therefore, by producing more zero bits in a cache line, SimiEncode can improve the reliability of STT-RAM. On the other hand, the endurance of PCM is more vulnerable to writing the bit '0'. Hence, we can simply flip the bits of the cache line encoded by SimiEncode, making the bit '1' predominate in the cache line and hence yielding in more endurable writes for PCM.

IV. PERFORMANCE EVALUATION

A. Evaluation Setup

We deploy SimiEncode atop of PCM to evaluate its lifetime, performance and energy. Specifically, we implement SimiEncode on the GEM5 Simulator [14] with NVMain [19]. Table II shows the major system configurations. To demonstrate the flexibility of SimiEncode, we select 10 benchmarks from SPEC CPU 2017 [5] and another 5 benchmarks from PARSEC [3]. We conduct two categories of evaluations with different objectives: performance-oriented evaluations and complementarity-oriented evaluations.

Performance-oriented evaluations: These evaluations are to show the advantages gained by SimiEncode, in terms of lifetime improvement, read/write latency reduction, and energy consumption saving. We select the following four related studies for comparison.

- **Baseline:** Baseline system directly writes the arrival data to NVMs without any operation.
- **BDI** [18]**:** BDI exploits small tail differences among the data (i.e., only the last few bytes of data in the same cache line are distinct) and uses an implicit zero base, an explicit non-zero base, and an array of differences to represent the cache line, so as to reduce the data size.
- **DFPC** [8]**:** DFPC compresses the cache line with the aid of static and dynamic data patterns. Static data patterns are predefined, while dynamic data patterns are dynamically extracted based on the data distribution characteristics of the application during execution.
- **ZD-FVC** [13]**:** ZD-FVC encodes zero and non-zero words with 1-bit and 4-bit tags, respectively. 1 bit is used to indicate whether the word is a zero word or not. For non-zero words, FVC [25] is used to further encode with 3-bit tags.

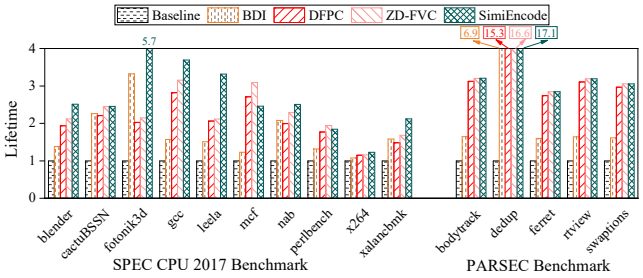

Fig. 12. The lifetime results (the larger the better).

Complementarity-oriented evaluations: These evaluations are to demonstrate that SimiEncode is orthogonal to and can be easily incorporated into existing bit flipping schemes to get further improvements. We mainly consider the following three bit flipping schemes.

- **DCW** [24]**:** It first reads the old data and compares it with the new written data, and only writes the different bits to reduce unnecessary bit writes.
- **FNW** [6]**:** FNW adaptively stores the original state or the flipped state selectively, based on which state can reduce more bit writes. In this experiment, we assign a tag bit for each 16 data bits to indicate whether the data bits are flipped or not.
- **READ** [22]**:** A variant of FNW that adjusts the encoding granularity based on the number of dirty words.
- **SimiEncode + DCW/FNW/READ:** We first use SimiEncode to encode the cache line, and then execute DCW/FNW/READ to further reduce the bit writes.

B. Evaluation Results and Analysis

We evaluate the overall performance from the following aspects: lifetime, write latency, read latency, energy consumption, and complementarity. For clear presentation, we normalize the experimental results to those of the Baseline.

Lifetime: We first assess the lifetime improvement gained by SimiEncode. We assume that the wear leveling approaches [13] are well performed to balance the bit writes, such that the lifetime of the PCM cells is directly determined by the number of bit writes. Figure 12 presents the lifetime results driven by different benchmarks, showing that by eliminating the writes for zero words and zero lines, SimiEncode dramatically prolongs the PCM lifetime by 382.0%, 177.1%, 66.2% and 42.0% on average, compared with the Baseline, BDI, DFPC and ZD-FVC, respectively. Both SimiEncode and ZD-FVC reduce the zero-word writing with the assistance of tag bits, but SimiEncode utilizes the word similarity much more effectively and reduces 20% more zero words than ZD-FVC.

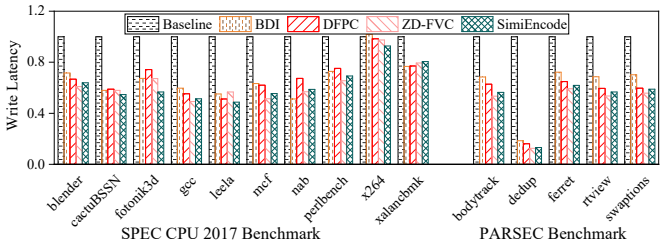

Fig. 13. Write latency results (the lower the better).

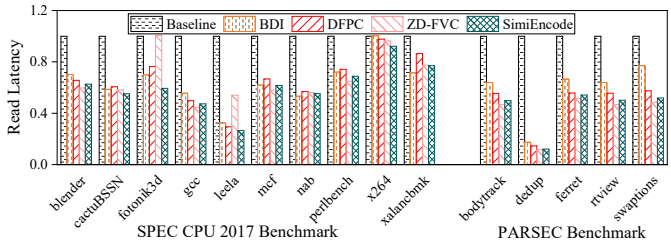

Fig. 14. Read latency results (the lower the better).

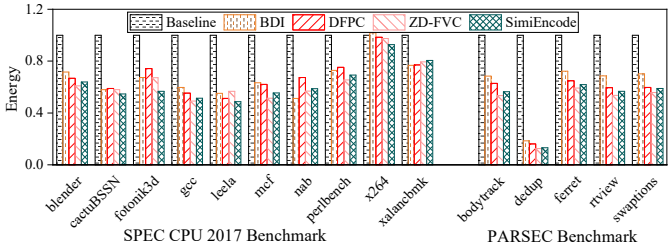

Fig. 15. Energy consumption results (the lower the better).

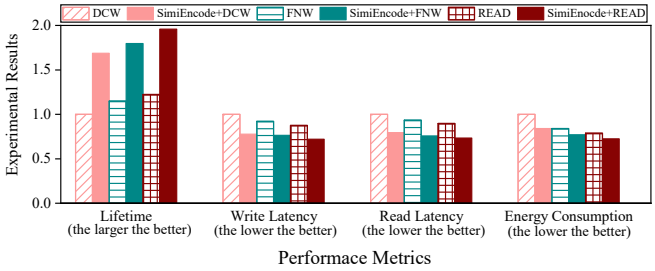

Fig. 16. Complementarity results on top of DCW, FNW and READ schemes.

Although ZD-FVC further compresses non-zero words, each non-zero word requires 3 additional tag bits, which offsets the lifetime improvement achieved.

Write latency: We then evaluate how much write latency can be reduced by SimiEncode. Figure 13 shows the write latencies of different approaches, indicating that SimiEncode effectively reduces the write latencies by 41.4%, 6.4%, 4.7% and 0.1% on average, compared with Baseline, BDI, DFPC and ZD-FVC, respectively. Although BDI, DFPC, and SimiEncode all strive to eliminate unnecessary bit writes to NVMs, SimiEncode can reduce more bit writes by only using a tag bit to represent a whole zero word. The reason is the same as that of the lifetime improvement by reducing more bit writes. However, since these schemes only reduce the number of bit writes, they still need to perform the write operations in the cache line. Thus, the write latency reduction is limited to some extent, compared with the lifetime improvements.

Read latency: Figure 14 shows the read latencies of different approaches. We can see that SimiEncode reduces the read latencies by 45.1%, 7.4%, 5.3% and 0.8% on average, compared with Baseline, BDI, DFPC and ZD-FVC, respectively. The read latency is mainly composed of two parts: the time of reading data from NVMs to the cache, and the waiting latency in the queue which accounts for a majority part [8]. SimiEncode reduces the write latency and hence accelerates the dispatching speed of the read requests in the waiting queue. In PCM, when a bank is busy handling the ongoing write request, the subsequent read requests should wait in the queue until the write completes [2], [12], [20], [26]. Thus, by significantly reducing the write latency, SimiEncode finally reduces the waiting time of the read requests and hence reducing the overall read latency. Though SimiEncode performs additional steps before reads, the induced extra latency overhead is actually negligible.

Energy consumption: Energy consumption is mainly introduced by the collective execution of read and write requests, and most of them is contributed by the write requests. The fundamental reason is that the PCM cell needs a high current to change the state to accomplish the write requests (Section II-A). Figure 15 shows the energy consumption results of different approaches. By significantly eliminating the bit writes, SimiEncode reduces the energy consumption by 37.1%, 5.2%, -2.9%, 19.0% on average respectively, compared wtih Baseline, BDI, DFPC, and ZD-FVC.

Complementarity: We finally demonstrate that SimiEncode can further prolong the lifetime and improve the performance by being deployed atop of existing bit flipping schemes, such as DCW [24], FNW [6] and READ [22]. Figure 16 shows the complementarity results. We can see that compared with the original bit flipping schemes, the SimiEncode-based schemes further prolongs the lifetime by 69.0%, reduces the write latency by 17.8%, reduces the read latency by 18.2%, and saves the energy consumption by 9.7% on average. These bit flipping schemes only exploit the similarity between the old and the new cache line to reduce the bit writes. By integrating our proposed SimiEncode scheme, the resulting SimiEncode-based schemes can also exploit the data similarity among the words within a cache line, which further reduces the bit writes, thus improving the performance and prolong the lifetime of PCM devices.

V. Related Work

In view of high write latency and limited endurance that still trouble NVMs, a lot of studies have conducted to relieve the writes to NVMs by opportunistically reduce the changed bits in the writes. These studies can be categorized into the following two aspects: bit flipping and data compression.

Bit flipping: Bit flipping schemes [6], [9], [10], [15], [22] transform the newly written word to another equal-length coded word, with the objective of reducing the bit writes to

NVMs. They opportunistically write either the newly written word or the flipped value of it after inspecting the bit difference between the new and the original words. FNW [6] ensures that the number of changed bits finally loaded to NVMs is always no more than half of the original word. As opposed to the one-to-one mapping in bit flipping, Coset coding [9] maps a word to a set of coded words and selects to write the one with fewest changed bits compared to the original word, so as to minimize the number of bits that must be written to NVMs. However, bit flipping schemes need additional reads of original words to check the bit difference between the new and original words.

Data compression: Data compression aims to represent a word with fewer bits. Frequent pattern compression (FPC) [1], [8] maintains a set of patterns and shrinks the word into a corresponding prefix if the word matches the recorded patterns. BDI (also called BΔI) [18] uses a common pattern with an array of differences to represent the original cache line. COE [21] combines data compression with bit flipping to further reduce the number of changed bits. ComEx coding [17] integrates FPC and BDI with the codes using the low energy states. However, these existing data compression schemes cannot sufficiently exploit the word similarity in the same cache line.

By contrast, our proposed SimiEncode partitions a cache line into many fixed-size words. It then carefully generates a mask word which has minimized the differences from the words within the same cache line to exploit the word similarity. Thus, the word similarity within a cache line is fully identified and utilized to reduce the bit writes to NVMs.

VI. Conclusion

This paper presents SimiEncode, an encoding approach that exploits the word similarity to improve the performance and prolong the lifetime of NVMs. The main idea of SimiEncode is to dig the similarity among the words of the same cache line and strive to transform them into zero words/sub-words. By using a single tag bit to represent a whole zero word after transformation, SimiEncode dramatically reduces the bit writes to NVMs. Compared with the state-of-the-art schemes, extensive evaluations with 15 real-world benchmarks show the efficiency of SimiEncode.

Acknowledgement

This work is supported by the National Natural Science Foundation of China under Grant No. U1705261, No. 61972325, No. 61872305, and No. 62072381, CCF-Tencent Open Fund WeBank Special Fund, CCF-Huawei Innovation Research Plan (CCF2021-admin-270-202102), Zhejiang Lab (2021KF0AB01).

References

[1] A. Alameldeen and D. Wood. Frequent Pattern Compression: A Significance-based Compression Scheme for L2 Caches. Technical report, University of Wisconsin-Madison, 2004.

[2] M. Arjomand, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. Boosting access parallelism to pcm-based main memory. In *Proc. of ISCA*, 2016.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT*, 2008.

[4] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao. Emerging NVM: A Survey on Architectural Integration and Research Challenges. *ACM TODAES*, 23(2):1–32, 2017.

[5] J. Bucek, K.-D. Lange, and J. v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Proc. of ICPE*, 2018.

[6] S. Cho and H. Lee. Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance. In *Proc. of MICRO*, 2009.

[7] S. Eilert, M. Leinwander, and G. Crisenza. Phase Change Memory: A New Memory Enables New Memory Usage Models. In *Proc. of IEEE International Memory Workshop*, 2009.

[8] Y. Guo, Y. Hua, and P. Zuo. DFPC: A Dynamic Frequent Pattern Compression Scheme in NVM-based Main Memory. In *Proc. of DATE*, 2018.

[9] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin. Coset Coding to Extend the Lifetime of Memory. In *Proc. of HPCA*, 2013.

[10] M. Jalili and H. Sarbazi-Azad. Captopril: Reducing the Pressure of Bit Flips on Hot Locations in Non-Volatile Main Memories. In *Proc. of DATE*, 2016.

[11] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *Proc. of ISPASS*, 2013.

[12] B. Li, S. Shan, Y. Hu, and X. Li. Partial-set: Write speedup of pcm main memory. In *Proc. of DATE*, 2014.

[13] H. Liu, Y. Ye, X. Liao, et al. Space-oblivious compression and wear leveling for non-volatile main memories. In *Proc. of MSST*, 2020.

[14] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, et al. The Gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[15] R. Maddah, S. M. Seyedzadeh, and R. Melhem. CAFO: Cost Aware Flip Optimization for Asymmetric Memories. In *Proc. of HPCA*, 2015.

[16] P. J. Nair, C. Chou, B. Rajendran, and M. K. Qureshi. Reducing read latency of phase change memory via early read and turbo read. In *Proc. of HPCA*, 2015.

[17] P. M. Palangappa and K. Mohanram. CompEx++: Compression-Expansion Coding for Energy, Latency, and Lifetime Improvements in MLC/TLC NVMs. *ACM TACO*, 14(1):1–30, 2017.

[18] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In *Proc. of PACT*, 2012.

[19] 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*, 14(2):140–143, 2015.

[20] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *Proc. of HPCA*, 2010.

[21] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, and C. Li. Extending the Lifetime of NVMs with Compression. In *Proc. of DATE*, 2018.

[22] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, C. Li, G. Xu, and Y. Chen. Adaptive Granularity Encoding for Energy-efficient Non-Volatile Main Memory. In *Proc. of DAC*, 2019.

[23] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li. Emerging Non-Volatile Memories: Opportunities and Challenges. In *Proc. of CODES+ISSS*, 2011.

[24] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu. A Low Power Phase-Change Random Access Memory Using a Data-Comparison Write Scheme. In *Proc. of ISCAS*, 2007.

[25] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *Proc. of MICRO*, 2000.

[26] J. Yue and Y. Zhu. Making write less blocking for read accesses in phase change memory. In *Proc. of MASCOTS*, 2012.

[27] W. Zhang and T. Li. Characterizing and Mitigating the Impact of Process Variations on Phase Change Based Memory Systems. In *Proc. of MICRO*, 2009.

[28] Y. Zhang, X. Wang, Y. Li, A. K. Jones, and Y. Chen. Asymmetry of MTJ Switching and Its Implication to STT-RAM Designs. In *Proc. of DATE*, 2012.

[29] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory using Phase Change Memory Technology. In *Proc. of ISCA*, 2009.