DFPC: A Dynamic Frequent Pattern Compression Scheme in NVM-based Main Memory

Yuncheng Guo, Yu Hua*, Pengfei Zuo
Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, China
*Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

***Abstract*—Non-volatile memory technologies (NVMs) are promising candidates as the next-generation main memory due to high scalability and low energy consumption. However, the performance bottlenecks, such as high write latency and low cell endurance, still exist in NVMs. To address these problems, frequent pattern compression schemes have been widely used, which however suffer from the lack of flexibility and adaptability. In order to overcome these shortcomings, we propose a well-adaptive NVM write scheme, called Dynamic Frequent Pattern Compression (DFPC), to significantly reduce the amount of write units and extend the lifetime. Instead of only using static frequent patterns in existing FPC schemes, which are pre-defined and not always efficient for all applications, the idea behind DFPC is to exploit the characteristics of data distribution in execution to obtain dynamic patterns, which often appear in the real-world applications. To further improve the compression ratio, we exploit the value locality in a cache line to extend the granularity of dynamic patterns. Hence DFPC can encode the contents of cache lines with more kinds of frequent data patterns. We implement DFPC in GEM5 with NVMain and execute 8 applications from SPEC CPU2006 to evaluate our scheme. Experimental results demonstrate the efficacy and efficiency of DFPC.**

I. Introduction

With the rapid growth of massive data to be processed, there is increasing demand to deploy large main memories [1]. However, the traditional DRAM technology as main memory is facing significant challenges in the cell scalability and power leakage. Non-volatile memories (NVMs), such as phase change memory (PCM) and resistive random access memory (ReRAM), have the potential to build future memory systems due to their salient features of lower standby power consumption and better scalability than DRAM [2]. However, NVMs suffer from a number of shortcomings in performance compared to DRAM [3]. In practice, we need to carefully handle the problems of write latency and limited lifetime in NVMs[4], [5].

In order to improve the write performance and endurance of NVMs, existing schemes mainly consider to reduce the number of write units. For example, Flip-N-Write (FNW) [6] compares the old and new data to reduce the number of bits changed by at least half. The efficiency of FNW depends on the difference between the old and new data in each cache line, which fails to exploit the data redundancy among different cache lines. In order to further reduce the data to be written, Frequent Pattern Compression (sFPC) [7] reduces the number of the written bits via compressing each 32-bit word. Specifically, sFPC maintains a pattern table in the memory controller and the table contains multiple common word patterns, such as the full-zero word pattern. If matching any of the patterns in the table, each word in the cache line is encoded into a compressed format, thus reducing the number of written bits. The word patterns in the pattern table are pre-set and cannot be modified, called $static\ patterns$. Dgien et al. [8] propose FPC that combines sFPC and FNW, and show that FPC can reduce on average $2\times$ more bit-writes than FNW. However, the FPC cannot work for the applications in which the word patterns with high frequency of appearance are not matched with the static patterns.

In order to improve the flexibility and adaptability of data patterns, we analyze the distribution of data values in each word and the word patterns with high frequency of appearance in different applications. We observe that different applications have their own high-frequent word patterns that may not be included in the pattern table. We call these high-frequent patterns that are different from application to application $dynamic\ patterns$. Based on the observation, we propose a well-adaptive NVM write scheme, called Dynamic Frequent Pattern Compression (DFPC). The idea behind DFPC is to extend the pattern table to include the dynamic patterns besides the static patterns. The dynamic patterns are obtained via sampling and analyzing the characteristic of bit value distribution in the cache lines to be written to NVM. We reduce the overhead of sampling by utilizing the match operations in the compression scheme. Moreover, DFPC exploits the value locality in a cache line to extend the granularity of dynamic patterns, from 32 bits to 64 bytes.

II. Backgrounds and Motivations

A. NVM Properties and Limitations

Unlike traditional charge-based memories such as DRAM and SRAM, emerging non-volatile memories store data by using resistive memories, which have higher density and scalability. Hence, NVMs have the potential to be widely used in the main memory.

Since all NVMs store information by changing the physical states, the write operation consumes longer time and energy than the read operation, which leads to the asymmetry of read and write. Moreover, the write operation wears out the NVM cell especially in high frequency, which results in the limited

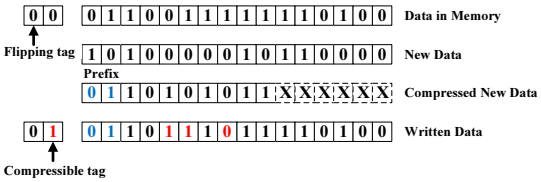


Fig. 1. An illustration of a 16-bit write operation in FPC

endurance of NVMs. Therefore, NVM-based systems need to reduce the number of write-bits in the write operation.

B. FNW and FPC

Flip-N-Write (FNW) [6] compares the new write-bits with existing bits to reduce the number of bit-flips. FNW reduces the number of bits changed more than half by using only one tag bit per compared unit.

Compression is an attractive approach to increase effective memory capacity of a large-size memory. Compression increases space usage, and hence improves performance and energy efficiency. Alameldeen et al. propose Frequent Pattern Compression (sFPC) [7] to compress write words based on a set of static patterns recorded in a pattern table.

Dgien et al. propose FPC by combining sFPC with FNW [8]. This method uses two tag bits per word and reduces the size of data before being written into the data array. Figure 1 presents an example of a write operation. We take 16-bit write unit as an example. The pattern table uses 2-bit prefix with 4 patterns, and each pattern contains 4 bits to represent the 4-bit characters' status (compressible or incompressible). In Figure 1, since the 2nd and 4th characters of the new data are zero-characters, the Compression Engine matches the new data with the pattern "X0X0" and encodes the prefix of this pattern "01" with the incompressible characters. Then the compressed data with 10 bits will be written in data array based on FNW. Finally, this write operation only writes 3 bits with FPC (sFPC combined with FNW).

C. Motivations

Existing schemes [7], [8] generally exploit static patterns for compression, which are obtained by analyzing the statistical characteristics of real-world applications. In general, the frequency of selected patterns directly affects the number of compressible cache lines. However, the high-frequent patterns in the applications are not always matched with the static patterns. In order to improve the flexibility of the patterns used in conventional compression schemes to adapt to different applications, we study the characteristics of the data generated by applications. We observe that most integers used in applications are compressed and stored in a 32-bit form, in which only 4, 8, or 16 bits have been really used. The low utilization results in the wide distribution of zero (in 4-bit character) [9]. Since read and write operations access one cache line (64 Bytes) at a time, the zero-character distribution of the cache line can fully exhibit the data distribution of applications. To verify this point, we conduct the simulation of examining write accesses. We use SPEC CPU2006 benchmark suite for mimicking the commercial workloads. The results reveal that

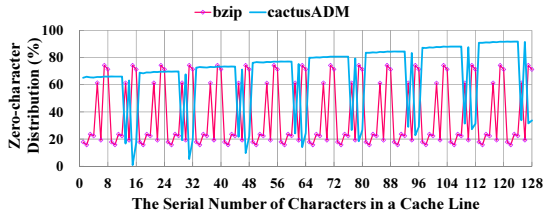


Fig. 2. The distribution of zero-characters

the potential patterns of the zero-character distribution. We observe that the characters in some specific serial numbers, where the zero-character appears frequently, form a part of a periodic cycle. As shown in Figure 2, each 8-character data corresponds to a periodic cycle in $bzip$. In each periodic cycle, the 4th, 7th, and 8th characters have high frequency of '0'. In $cactusADM$, the specific serial numbers are different with those in $bzip$ since the data patterns have changed. If we consider zero-character as the compressible unit, the higher points which represent the higher frequent occurrence of zero-character will be regarded as the compressible part of data pattern. The data patterns of $\frac{64\times 8bit}{32bit} = 16$ words in the cache line can be extracted based on the distribution at runtime.

III. System Design and Implementation

A. Dynamic Pattern

Based on our observation, we first define the static pattern and the dynamic pattern. The static pattern is interpreted as the pattern pre-defined that fails to be modified at runtime. The dynamic pattern is interpreted as the pattern obtained by sampling and analyzing the characteristic of bit value distribution in the incoming data at runtime. Figure 3 presents an example showing the pattern extracted from the distribution in $bzip$. In this example, the dynamic pattern, "XXX0XX00" (the incompressible 4-bit character is expressed as 'X'), can be obviously obtained by observing the zero-character distribution. This pattern occupies 36% of the words, more than other static patterns, which reveals the potential high frequency of the dynamic patterns. We can add the dynamic patterns to the pattern table to enhance the compressibility of the compression scheme.

B. Extended Pattern

Dynamic patterns can improve the adaptation of compression in various applications. However, the scalability of data patterns in compression is low due to the fixed format of data patterns. In our observation, the requests always access one 64-Byte cache line at a time, but FPC only compresses words with 32-bit form. When a write request with all zero bits accesses the data array, it will be divided into 16 words for compression, which causes $3 \times 16 = 48$ write bits before being written. For further reducing the number of bit-writes with high efficiency, we extend the size of dynamic patterns. We improve the traditional extract algorithm to detect more complex patterns, such as BDI patterns, which exploit the value locality of the cache line [10], [11].

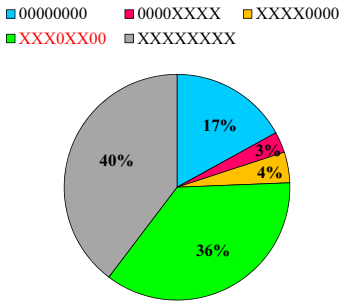


Fig. 3. The distribution of data patterns in *bzip*. The first three patterns are static patterns. "XXX0XX00" is a dynamic pattern extracted from the distribution. The pattern with all 'X' means the incompressible word.

C. Dynamic Frequent Pattern Compression

By using the extended dynamic patterns, we propose a well-adaptive NVM write scheme. The idea is to extract the dynamic patterns from the zero-character distribution and use both static patterns and dynamic patterns to compress more data. The scheme is implemented in Memory Controller. For the compression with common patterns, the scheme divides the 64-Byte cache line into 16 32-bit words. The content of a word is checked to determine if matching any of the patterns with high frequency. Each data pattern contains 8 bits to represent 8 4-bit characters' status (compressible or incompressible). For the compression with the extended patterns, the scheme can select appropriate matching algorithms. For supporting compression in NVMs, the cache line is modified to include the tag bits. Each word needs 2 tag bits as the compressible tag and flipping tag, and hence 32 additional tag bits are added into a 64-Byte cache line. Figure 4 shows the architecture of the NVM-based main memory system with DFPC. We add a component in the architecture of compression scheme, called Sampling Engine, which consists of a group of counters for sampling the zero-character distribution of the application and extracting the dynamic patterns. After obtaining the dynamic patterns, the Sampling Engine adds the patterns to the dynamic pattern table. For write accesses, the scheme consists of three stages, i.e., Sampling stage, Analysis stage and Dual-Pattern Compression stage.

1) Sampling Stage: The dynamic patterns require sufficient number of words to be obtained by the extraction algorithm. To facilitate compression operations in the Sampling stage, a few static patterns are pre-defined in a pattern table. In our experiments, we use 4 static patterns and 3-bit prefix to maintain 8 data patterns in total. So the number of appropriate dynamic patterns should be no more than 4. During a write access, if a word is compressible and matched with a pattern, it is encoded along with a prefix. The compressible tag is set to be enabled. Before being written through the cache line of NVMs, the bits of each word are compared with existing bits. If more than half of the bits are different, the whole word will flip, and the flipping tag is set to be enabled. For sampling, we design a Sampling Engine to record the most frequent patterns. The Sampling Engine uses two groups of counters. One group is for counting the number of zero-characters in the cache line. The number of counters is set as 128. The other group is for counting the reduced bits of the extended patterns (All-zero cache lines are not included). To reduce additional overhead of sampling, we leverage a match operation, which reads the value of each character, and supports the sampling operation in the meantime. Hence the sampling information in the Sampling Engine comes from the Compression Engine rather than the Write Buffer.

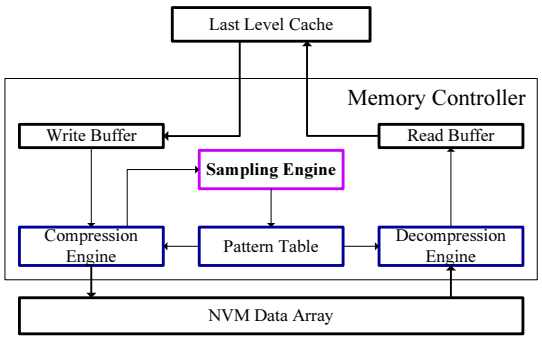


Fig. 4. The DFPC system

2) Analysis Stage: When the sampling amount reaches the sampling granularity N, the dynamic patterns are obtained by analyzing the zero-character distribution. The distribution may be quite complex in some application. So we design a simple and effective pattern extracting algorithm with high efficiency. In the Analysis stage, the counters are analyzed by the extracting algorithm (as shown in Algorithm 1). First, the counters are checked for obtaining the Upper Bound (UB) and the Lower Bound (LB). The value of the threshold T is calculated by using the equation in line 5 of Algorithm 1.

The threshold factor (TF) and the sampling granularity N are pre-defined via experimental verification in Section IV-A.

The dynamic patterns can be extracted by comparing the count values of the counters with the threshold. When the count value is larger than or equal to the threshold, the pattern

Algorithm 1 Algorithm for extracting dynamic patterns.

Require:
The array of count values in counters, C;
The number of counters, n;
The threshold factor, TF;
Ensure:
The array of dynamic patterns, P;
1: **for** each $i \in [1, n]$ **do**
2: $UB = MAX(UB, C[i])$;
3: $LB = MIN(LB, C[i])$;
4: **end for**
5: $T = LB + (UB - LB) \times TF$
6: **for** each $i \in [1, n]$ **do**
7: **if** $C[i] \geq T$ **then**
8: $C[i] = 0$;
9: **else**
10: $C[i] = 1$;
11: **end if**
12: **end for**
13: extracting $\frac{n}{8}$ dynamic patterns from C and adding them to P
14: eliminating existed and repeated patterns
15: **return** P.

code is set as '0', which means this character is compressible. Otherwise, the pattern code is set as '1' ('X'). After checking all counters, the algorithm extracts 16 patterns. However, some of the extracted patterns are repeated or useless, like "XXXXXXXX". The repeated patterns and incompressible pattern should be removed. In order to make full use of the limited number of prefixes, DFPC selects top-4 patterns based on the compression ratios of these extracted patterns and extended patterns. The compression ratio is determined by the pattern's compressibility and frequency of occurrence. Finally, the remaining 4 dynamic patterns are added to the pattern table. The pattern table won't be modified later.

3) Dual-Pattern Compression Stage: In the Dual-Pattern Compression stage, the Compression Engine can obtain two kinds of data patterns from the dynamic pattern table and the static pattern table, respectively. The content of a word is compressed based on the static and dynamic patterns concurrently. The Compression Engine then compares the numbers of compressible bits in two data patterns, and chooses the pattern with more compressible bits as the matching result to execute the compression operation.

During the read accesses, the state of a compressible tag bit determines if the data are compressed. The state of flipping tag bit presents the flipped data. After parsing these tags, the Decompression Engine can decoded the word rapidly. If the data are flipped, the Decompression Engine will first flip the data. Then if the data is compressed, the Decompression Engine will read the first 3 bits to obtain the prefix, which establishes a match between the compressed word and the matching pattern, and find the matching pattern. Finally, the Decompression Engine decodes the compressed data and is filled with zero-characters based on the pattern.

D. Temporal and Spatial Overheads

The implementation of DFPC consists of compression and pattern extraction. The time overhead of compression is about several cycles on average in Memory Controller [11]. In Section IV-B, we define the access time as 3 cycles in compression and 2 cycles in decompression. For pattern extraction in the Analysis Stage, Algorithm 1 traverses the counters with constant-scale time complexity. Since most applications usually run quite a period of time after warming up, the time consumption of pattern extraction can be negligible. After pattern extraction, the static patterns are still retained in the pattern table. Thus the old data compressed by using static patterns can be decompressed via its prefix during the read accesses.

As we mentioned, the cache line is modified to provide the tag bits. FNW needs one flipping tag bit per word. For supporting compressions, like FPC and DFPC, each word needs another tag bit to mark the word compressible or not. In order to carry out the sampling, DFPC needs a set of counters, which only use $128(counters) \times 8\ Byte(64-bit\ int) = 1KB$ and incur no extra space overhead at run time.

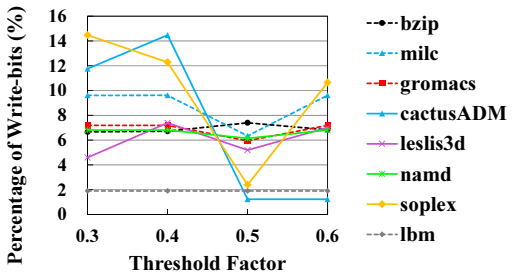


Fig. 5. Write-bits with different Threshold Factors TF

IV. Performance Evaluation

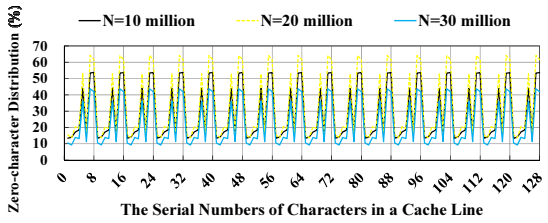
A. Experimental Configurations

We present the configuration and the experimental environment. We implemented DFPC in the GEM5 simulator [12] with NVMain [13] to evaluate our design. The configurations of the simulation are shown in Table I. In our experiments, all caches have 64B cache line size. The 8GB PCM main memory controller has individual read and write queues and uses FRFCFS scheduling algorithm of NVMain. The static patterns are selected from the zero-extended patterns in [8], [14]. During trace generation, the applications are first run through 10 million instructions to warm up the system [11]. The applications then run 2 billion instructions to record enough accesses. We evaluate and compare the performance of our DFPC to three state-of-the-art methods, Flip-N-Write (FNW) [6], Base-Delta-Immediate Compression with FNW (BDI), and Frequent Pattern Compression with FNW (FPC) [8].

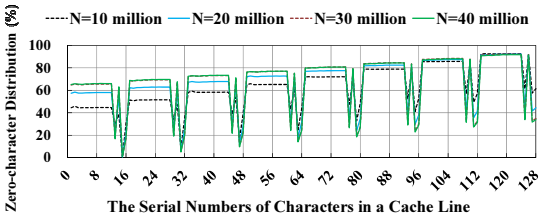
1) Threshold Factor: Before performing the evaluation of DFPC, we first analyze the impact of the parameters in the design. The threshold factor TF is an important parameter of the extraction algorithm used in DFPC. With a high threshold factor, the dynamic patterns extracted from the distribution can compress more words due to the less number of specific serial numbers. The dynamic patterns extracted by pre-setting lower threshold factor have more compressible characters in a word. We consider that if we preset the threshold factor within a rational range, there will be a small effect on the performance of write-bits reduction. To verify this point, we implement DFPC and count the amount of written bits in 8 applications with the threshold factor from 0.3 to 0.6. Statistical results are shown in Figure 5. We observe that the reductions of write-bits perform well from 0.4 to 0.6. Hence, we choose 0.5 as the default threshold factor.

TABLE I
Experiment Configurations

Processor and Cache	
CPU	4 cores x86-64 processor, 2 GHz
Private L1/L2 caches	32 KB/ 2048 KB
Memory Using PCM-based Memory	
Capacity	8 GB, 1 channel, 1 ranks, 8 banks
Read latency	75 ns
Set latency	37.5 ns for each word of 32 bits
Reset latency	12.5 ns for each word of 32 bits
Parameters of DFPC	
Compression latency	1.5 ns [11]
Decompression latency	1 ns [11]
Sample granularity N	5 million write accesses
Threshold factor TF	0.5



(a) $bzip$



(b) $cactusADM$

Fig. 6. Zero-character distribution with different Sample Granularities N

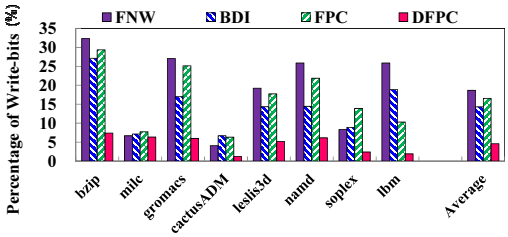


Fig. 7. The Percentage of Write-bits

2) Sample Granularity: To improve the representation of the dynamic patterns, we need to select appropriate sample granularity N. We use DFPC to collect the zero-character distribution every 10 million write accesses. One of the statistical results is shown in Figure 6. We observe that there are few differences among the statistics. The cycle of distribution remains unchanged in all statistical results, while the number of compressible characters in a word fluctuates within a narrow range. The dynamic patterns will be more representative when expanding the sample granularity, while the dynamic patterns will be extracted fairly later and perhaps cause performance degradation.

B. Evaluation Results and Analysis

1) Write-bit Reduction: DFPC compresses the contents of words, based on the static and dynamic patterns, to reduce the number of the bits to be written. The write-bit reduction reduces the energy consumption and improves the endurance of PCM. As shown in Figure 7, by using FNW, FPC and BDI can reduce more than 70% written bits. With dynamic patterns, DFPC outperforms FNW, FPC and BDI, and decreases 75%, 72% and 68% written bits on average.

2) Write Latency: Compression schemes can significantly reduce the number of written bits before executing comparison in a write operation. Hence the number of writing words can also be reduced before being written. The write accesses in compressions can be completed more quickly compared with FNW. Experimental results of write latency are shown in Figure 8. In general, DFPC outperforms FNW, FPC and BDI, and decreases 74%, 43% and 61% write latency on average.

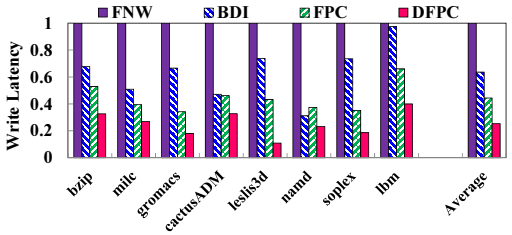


Fig. 8. Write Latency (normalized to FNW)

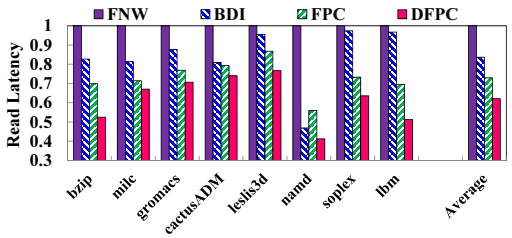


Fig. 9. Read Latency (normalized to FNW)

3) Read Latency: Read latency is important to the PCM-based main memory performance. Compression schemes reduce the write latency, and hence improve the efficiency of the queue of operations and reduce the waiting time of read operations, which is an important fraction of read latency. Figure 9 shows the read latency reductions of DFPC, BDI, and FPC compared with FNW. By using compressions, one read phase can gain more data with the decreasing size of the compressed word, and the experimental results verify that the performance of write-bits reduction impacts on read latency. Due to the improvement in write latency, the experimental results of DFPC in read latency outperform BDI and FPC. DFPC obtains 38% more read latency reduction compared with FNW.

4) IPC and Energy Improvement: IPC is important to the entire system performance. Minimizing the number of write-bits can reduce the response time and improve the access speed. IPC is influenced by a variety of factors, including the latency of the accesses. The results of IPC improvement are illustrated in Figure 10. DFPC performs well in most applications, especially in $bzip$ and lbm due to the high compressibility of dynamic patterns and high frequency of write accesses. DFPC obtains 23% speedup on average, compared with FNW.

Energy consumption is an important concern in the PCM-based main memory, especially in MLC and TLC PCMs. In write accesses, the PCM cell requires high current. Handling the high levels of power is a crucial challenge for the PCM devices and the entire system, especially during the high frequency of write accesses. High energy consumption causes the heat problem, which decreases the endurance and possibly destroys the device, and increases the budget for cooling. Hence the improvement on energy can bring significant benefits to both the budget saving and environment. As shown in Figure 11, DFPC alleviates the energy consumption and extends the lifetime. On average, DFPC saves 41% energy compared with FNW.

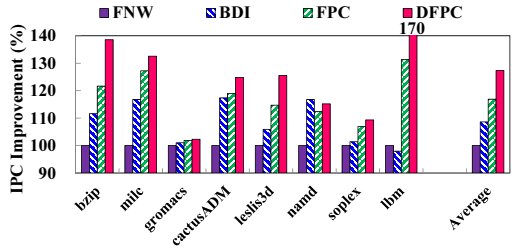


Fig. 10. IPC Improvement (normalized to FNW)

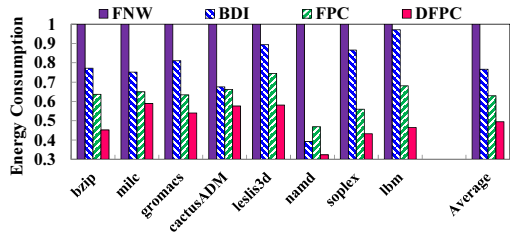


Fig. 11. Energy Consumption (normalized to FNW)

V. Related Work

There are many schemes aiming to reduce the amount of write-bits and improve the write performance. Data Comparison Write (DCW) [15] exploits the non-volatility of NVMs and checks the new write-bits with existing bits to determine the modified bits. DCW can reduce the write power consumption to a half. Flip-N-Write (FNW) [6] compares new data with existing data to reduce the number of bit-flips.

Compressions are also widely used in NVM-based main memory systems. Alameldeen and Wood [7] observe that most data stored in fixed size only use several bits and they propose a significance-base compression scheme, call sFPC. Dgien et al. [8] combine sFPC with FNW, and observe that DCW and FNW can not work as expected with the compressed data. Base-delta-immediate Compression [10] (BDI) exploits the value locality of the words in one cache line. BDI has high compression ratio, but delivers inefficient performance in some application when combined with FNW. Palangappa and Mohanram [11] propose a coding scheme in MLC/TLC NVMs, which encodes the contents of cache lines with static integer patterns and BDI patterns. Hycomp [16] is a hybrid cache compression framework, which selects the appropriate compression based on the feature of the cache lines in Last Level Cache. This general framework has high compression ratio and strong compatibility. Our DFPC can be also implemented within Hycomp.

VI. Conclusion

To improve write performance of NVMs and offer the flexibility of compression schemes, we propose a well-adaptive NVM write scheme, called Dynamic Frequent Pattern Compression (DFPC). DFPC compresses the data contents based on a set of specific patterns to minimize the written data. We observe that some zero characters in the specific localities, where the zero-character appears frequently, form a periodic cycle in the typical data distribution. By considering these localities as the compressible part of a word, we extract dynamic patterns by sampling and analyzing the regularity of bit value distribution in the incoming data at runtime. DFPC also extends the data patterns by exploiting the value locality in a cache line to include more kinds of patterns and achieve higher compression ratio. Experimental results demonstrate that DFPC reduces the amount of write-bits by 75%, write latency by 74%, and read latency by 38% while gaining 1.2× IPC improvements, compared with the state-of-the-art FNW. While we take PCM as an example of NVMs, the proposed scheme should also improve the performance of other NVMs, such as ReRAM and STT-RAM, which have long write latency and low cell endurance.

Acknowledgment

This work is supported by National Key Research and Development Program of China No.2016YFB1000202, National Natural Science Foundation of China (NSFC) No.61772212 and State Key Laboratory of Computer Architecture, No.CARCH201505.

References

[1] P. J. Nair, D. H. Kim, and M. K. Qureshi, "Archshield:architectural framework for assisting dram scaling by tolerating high error rates," in *Proc. ISCA*, 2013.

[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, 2010.

[3] H. Mao, X. Zhang, G. Sun, and J. Shu, "Protect non-volatile memory from wear-out attack based on timing difference of row buffer hit/miss," in *Proc. DATE*, 2017.

[4] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[5] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proc. MSST*, 2017.

[6] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *Proc. MICRO*, 2009.

[7] 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.

[8] 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 *Proc. NANOARCH*, 2014.

[9] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proc. ISCA*, 2005.

[10] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proc. PACT*, 2012.

[11] 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 Optimization (TACO)*, vol. 14, no. 1, p. 10, 2017.

[12] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[13] 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, 2015.

[14] Z. Li, F. Wang, Y. Hua, W. Tong, J. Liu, Y. Chen, and D. Feng, "Exploiting more parallelism from write operations on pcm," in *Proc. DATE*, 2016.

[15] 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. ISCAS*. IEEE, 2007, pp. 3014–3017.

[16] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proc. MICRO*, 2015.