CONTENT-AWARE SPIN-TRANSFER TORQUE MAGNETORESISTIVE RANDOM-ACCESS MEMORY (STT-MRAM) CACHE DESIGNS

By

QI ZENG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2017

To my great parents and my dearest wife

ACKNOWLEDGMENTS

When the journey of doctorate study came to end, I wrote down this sentence with all sorts of feelings welled up in my heart. Since the day one in Gainesville, I'm so fortunate to receive countless help, inspiration and encouragement from my family, friends and colleagues. Life would be extremely struggling without those warm hearts around me.

First of all, I would like to express my sincere gratitude to my advisor Prof. Jih-Kwon Peir for the continuous support of my Ph.D. study and related research, for his patience, motivation and inspiration. He has immense knowledge and experience in our research areas, but he never suffocates my premature or incomplete ideas and always encourages me to think out of the box. In life, Dr. Peir and his wife are like my family in United States, and I cannot count how many times they invited me to celebrate holidays or watch gator football games together.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Sartaj Sahni, Prof. Shigang Chen, Prof. Alin Dobra and Prof. Yuguang Fang, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research.

I'm grateful that I had the chance to work with some great colleagues and made some close friends. Yang Chen never refused my request for assistance on debugging large programs and motivated me to write elegant code. Hang Guan helped me with all kinds mathematic problems and his collection of board games did make lots of weekend nights joyful. I would like to also thank my fellow labmates Rakesh Jha, Mingcong Song, Bo Ma and Xi Tao for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in past years.

Lastly, I can never accomplish this without support from my family. My parents cared about my education since I was a kid and provided me with the best resources they can get. I also

want to thank my dearest wife, Min He, who has always been there, through my up and downs. She sacrificed a lot for me through these years, and I will never let her down.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

CONTENT-AWARE SPIN-TRANSFER TORQUE MAGNETORESISTIVE RANDOM-ACCESS MEMORY (STT-MRAM) CACHE DESIGNS

By

Qi Zeng

Chair: Jih-Kwon Peir
Major: Computer Science

As the speed gap between the processor cores and the memory subsystem has been continuously widening in this multi-core era, designers usually use larger caches to hide the memory latency. However, current cache implementations use expensive Static Random-Access Memory (SRAM), which has inherent difficulties in implementing larger caches due to size and power limitations. Spin-Transfer Torque Random-Access Memory (STT-MRAM) is a promising candidate for future cache considering its strong scalability, low read latency, low leakage power and non-volatility. However, STT-MRAM has its own drawbacks such as high write energy and latency, as well as less-than-desirable write endurance, making its usage as caches challenging.

We start STT-MRAM cache design by optimizing the cache replacement policy with content-awareness. Instead of replacing the LRU block under the conventional pseudo-LRU replacement policy, we replace a near-LRU block which is most similar to the requested block. To avoid fetching and comparing all bits of each candidates, we propose a novel content encoding method which represents 64-byte block with a few encoding bits and measures the content similarity by Hamming distance between the encoding bits. Performance evaluation demonstrates that the proposed method is effective with significant reduction in total switch bits, which results in noticeable improvement on write endurance and write energy consumption.

We continue the replacement study on Multi-Level Cell (MLC) STT-MRAM, which records two bits in a single cell to further improve the density for building bigger caches. However, MLC STT−MRAM has two-step transition problem during cache writes incurring extra flips, high energy consumption, and less life time. To alleviate high write energy issue, we apply block-content encoding method and use the encoding bits to select the victim block. After the replacement block is picked, intelligently remapping of 2-bit value to resistance states inside a 2-bit MTJ can further reduce the write energy consumption in comparison to LRU replacement.

Furthermore, inspired by the success in exploiting content similarity, we also proposed a dual-block compression method which uses an entire uncompressed block as dictionary and compressed multiple neighboring blocks in a separate companion block to provide larger dictionary for good compression ratios.

CHAPTER 1
INTRODUCTION

1.1 Magnetic Memory Technologies

Magnetic storage media has been widely used as hard drive instead of on-chip embedded memory due to its storage limitation. Since the first 128Kbits MRAM chip was manufactured in 2003 [1], the development of MRAM draws increasing attention. The second-generation magnetic memory, the spin-transfer torque MRAM (STT-MRAM) provides almost all the desirable features for future cache: fast read speed, high density, near zero leakage power, and nonvolatility and is considered as a viable replacement to SRAM for building large on-die caches.

In an STT-MRAM cell, 1-bit value is recorded as the two resistance states of a magnetic tunneling junction (MTJ). The resistance states of MTJs can be switched to each other by passing through a spin-polarized current [2]. Very recently, *Everspin* shipped sample products of 64MB STT-MRAM [1], which signals the bright start of the commercialization era after many years of joint effort from both academia and industry [3][4].

Recently, multiple-level cell (MLC), which stores multiple bits per MTJ cell, has also been developed to further increase data density [5][6][7][8]. The MLC technology has achieved great success in commercial NAND flash [9]. In MLC STT-MRAM, each MTJ cell can store 2-bit information − 00, 01, 10, and 11, represented by 4 MTJ resistance states in our study.

1.2 Structure of MTJ

Conventional STT-MRAM cell stores one bit information using two resistance states of a magnetic tunneling junction (MTJ) device, is known as single-level cell (SLC) STT-MRAM as shown in Figure 1-1 (a). The MTJ of SLC STT-MRAM consists of two ferromagnetic layers and one tunnel barrier layer (MgO). The reference ferromagnetic layer has a fixed magnetic direction

(MD), and the free ferromagnetic layer can change its magnetic direction via a spin-transfer torque. When the two layers have different magnetic directions, the MTJ resistance is high, indicating a '1' state, while the MTJ resistance is low when the two layers have the same direction, indicating a '0' state.

Multiple-level cell (MLC) can store multiple bits per cell with two proposed types of MLC MTJs [7], parallel MLC MTJs and serial MLC MTJs. Because parallel MTJs have relatively higher Tunneling Magneto Resistance (TMR) ratio, smaller switching current and better reliability than serial MTJs, we focus on the structure and working mechanism of parallel MTJs. Figure 1-1(b) shows the structure of a parallel MLC MTJ. The free layer of the MLC MTJ has two magnetic domains of which the MDs can be changed separately. The MD of one domain (soft domain) can be switched by a small current while that of the other domain (hard domain) can be switched by only a large current.

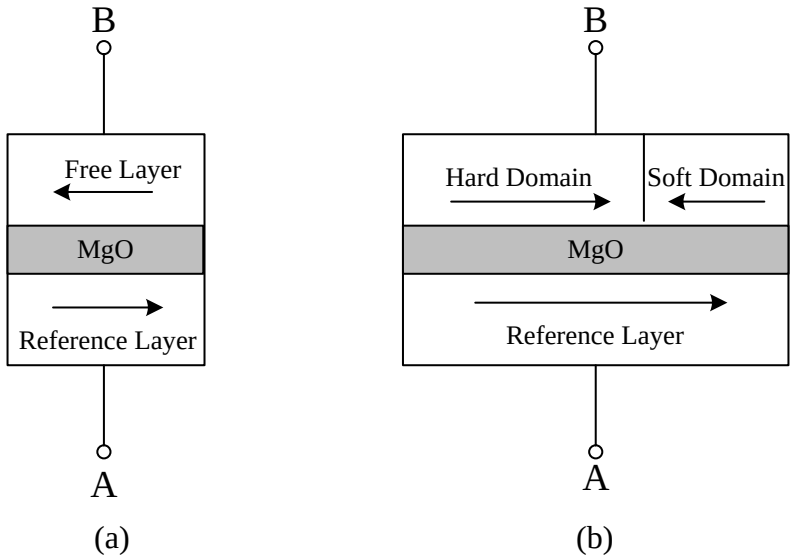


Figure 1-1. MTJ structure. (a) SLC MTJ. (b) Parallel MLC MTJ

The four resistance states are defined by the four combinations of relative magnetization directions of the two domains as shown in Figure 1-2. Specifically, the most significant bit

(MSB) and the least significant bit (LSB) can be defined by the magnetization directions of the hard and soft domains, respectively.

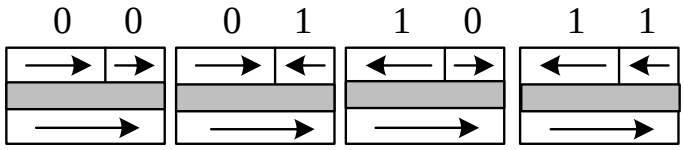


Figure 1-2. MLC STT-MRAM resistance states

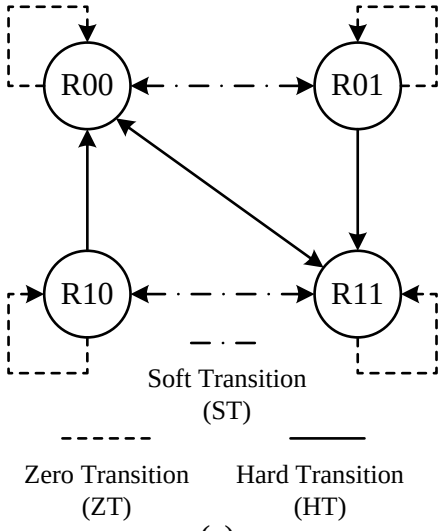
1.3 Two-step Access Scheme for MLC STT-MRAM

The read and write of the 2-bit stored in MLC STT-MRAM may involve two-step operations [7]. Three reference voltages are provided in the MTJ. In read operations, the first bit value is determined by comparing one reference voltage with the baseline voltage generated by a read current applied to the MTJ. Given the first bit value, each remaining reference voltage is used to determine the second bit value. In write operations, the write current passes through both the hard domain and the soft domain in the free layer at runtime. However, the hard domain requires a larger switching current than the soft domain. Soft domain can be switched alone if a small write current is applied, and the switching of hard domain requires larger current, which changes soft domain to the same direction at the same time.

Transitions between some pairs of resistance states need take two-step switching shown in Figure 1-3 (a). For example, the transition of 00 → 10 can only be accomplished through two steps: the 1st step is 00 →11 by applying high switching current; the 2nd step is 11 →10 by applying low switching current. Per the original and the new 2-bit values in MTJ, the transitions (updates) of the MTJ resistant states can be categorized into 4 types:

- **Zero transition** (ZT): MTJ stays at the same state;
- **Soft transition** (ST): Only the soft domain is switched in the transition, i.e. 00↔01 and 10↔11;
- **Hard transition** (HT): Both soft and hard domains are switched in the transition, i.e. 00→11, 01→11, 10→00 and 11→00.
- **Two-step transition** (TT): Hard domain is switched and in the new value hard domain and soft domain are in different directions. So, such transition completes with two steps, including one HT followed by one ST, i.e. 00→10, 01→10, 10→01 and 11→01.

Figure 1-3 (b) summarizes all combinations for MTJ transitions, where R stands for the resistant state and the subscript indicates the values of the two bits [7][10]. We also classify ST and ZT into two subtypes indicated by the subscripts which is useful for making remapping decisions and will be described in Chapter 3. Clearly, TT encounters extra updates to the soft domains, which has the highest write energy, extra update also hurts endurance. HT must apply high current to change MD of both hard and soft domains, also incurs high write energy. ST has considerable lower write energy since it requires low current to change only the soft domain MD. Certainly, there is no write energy for ZT since the 2-bit value remains unchanged.



(a)

From / To	R_{00}	R_{01}	R_{10}	R_{11}
R_{00}	ZT_0	ST_1	TT	HT
R_{01}	ST_0	ZT_1	TT	HT
R_{10}	HT	TT	ZT_1	ST_0
R_{11}	HT	TT	ST_1	ZT_0

(b)

Figure 1-3. MLC STT-MRAM transitions (a) state transition diagram (b) 4 types of transitions

1.4 Drawbacks of STT-MRAM and Proposed Solutions

STT-MRAM's high density, fast read speed, low leakage power, and non-volatility make it a promising candidate for future on-chip caches. However, the energy and latency overheads associated with the write operation as well as less than desirable cell endurance are the major drawbacks for this technology. Studies show that the STT-MRAM generates high write current which consumes 6-14 times more energy per write access than the SRAM counterpart [3]. Many efforts to reduce the high write energy have been proposed, both at architecture level [3][11], and at circuit level [12]. Our proposed intelligent content-aware replacement policy recognizes the content of the missed and the selected replacement block to minimize the number write operations for reducing the write energy.

The lifetime of STT-MRAM is determined by the first wearing out memory-cell and thus cannot guarantee the correctness of data storage in cache after such permanent cell failure. Although a prediction of up to 10^{15} programming cycles is often cited by many works, the best endurance test result for SLC STT-MRAM devices so far is reported as less than 4×10^{12} cycles [13]. For MLC STT-MRAM, the even larger write current shortens the lifetime of memory cells as dielectric breakdown [7]. So, decreasing the total number of write operations can help increase the endurance of STT-MRAM cache as well.

To prepare the STT-MRAM for future cache design, we proposed two innovative solutions focusing on content-aware cache replacement policies to overcome such drawbacks of STT-MRAM, which improve both the cache endurance and energy efficiency.

1.5 Performance Evaluation Methodology for Dissertation Research

We use two different evaluation frameworks for our STT-MRAM studies: trace-driven simulation and timing simulation.

We start with trace-driven simulation for studying performance of large SLC STT-MRAM caches. The memory traces are generated on the QEMU storing both memory addresses and the corresponding data content. Trace-driven simulation can finish a long simulation run with multi-billion instructions in a couple of hours and significantly reduce the simulation complexity in multi-program workloads.

To better understand the latency overhead introduced by our proposed method, we move on to Gem5 [14], a cycle accurate system simulator. The simulation results show the IPC variance caused by different access latencies of write operation designs in MLC STT-MRAM cache and the effect of different replacement policies. Simpoint [15] is also employed to automatically pin down the most representative phase out of a long program execution.

Besides, the detailed parameters of access latency, energy, and leakage power of both SLC STT-MRAM and MLC STT-MRAM are generated with NVSim [16].

1.6 Outline of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 shows a content-aware cache replacement study for SLC STT-MRAM. In Chapter 3, we proposes an energy-efficient MLC STT-MRAM Cache design with Content Awareness. Chapter 4 describes a new cache compression method by taking advantage of data locality among nearby blocks. At the end, we conclude this work and discuss future works.

CHAPTER 2
CONTENT-AWARE NON-VOLATILE CACHE REPLACEMENT IN SLC STT-MRAM

2.1 Endurance Concern with STT-MRAM

A block is written into low-level STT-MRAM caches either when a missed block is brought into the cache, or when a dirty block from higher level caches is replaced and must be written back. Previous work [17][18] discussed mechanisms to reduce write-backs of dirty blocks.

In this work, we focus on the write endurance and write energy issues when a new block is brought into the cache. With frequent on-chip cache misses from large-scale, parallel workloads, it is essential to improve write endurance and reduce write energy for STT-MRAM cache. The early-write termination (EWT) technique [11] avoids redundant writes to those bits which have the same value as that in the replaced cache block. Such redundant writes are detected and terminated in their early stage to reduce write energy. The Flip-N-Write (FNW) [19] takes one step further to reduce write energy and improve write endurance. They introduce a 'flip' bit for each word in a cached block to indicate whether the content of the word has been flipped. The flip bit is on if flipping the bit value of the word can save more redundant writes.

For example, assume the content of an 8-bit LRU block is '00111011' and the 8-bit content of the new block is '11001101'. Only 2 bits have the same value, hence 6 bits must be updated by switching the bit value. In this case, however, the flip bit can be set and the 8-bit content becomes '00110010' to reduce the switch bits from 6 to 2. When a word with the flip bit on is accessed, the content will be flipped to recover the original value.

In this chapter, we explore a new dimension to further reduce the actual total switch bits to write a cache block. Although both the EWT and the Flip-N-Write avoid the redundant writes, they do not take the block content into consideration on cache replacement. In other words,

instead of replacing the LRU block, we want to replace the block whose content is most like the missed block and reduce the switch bits.

2.2 Motivations and Related Work

In this section, we demonstrate the opportunity to reduce the total write switch bits based on content-aware cache replacement. Meanwhile, the results also show that significant increase in cache misses when the evicted block is chosen solely by minimizing the switch bits. Related works in handling the high-energy issue for writes to a STT-MRM cache are also given.

2.2.1 Opportunity of Reducing Switch Bits

In Figure 2-1, we show the average switch bits (left Y-axis) of each cache replacement for the P-LRU and the optimal write replacements. In the optimal write replacement, we compare the content of the new block with the contents of all cache blocks in the target set and select the replacement block with minimum switch bits. We also show the impact on the cache miss rate (right Y-axis). Both EWT and Flip-N-Write schemes are included in our evaluation of the two replacement policies. This evaluation is done using trace-driven simulation with 16 SPEC CPU2006 benchmarks [20]. We collect the L2 miss traces and use them to drive a shared L3 cache model with 4MB capacity and 32-way set-associativity. Detailed descriptions of the simulation tool and workloads will be given in Section 3 of this chapter.

As can be observed, the optimal write replacement reduces the average switch bits per insertion significantly, ranging from 16% to 93% over the 16 workloads as shown by the two bars for each workload. Such a huge gap motivates us to search for a realistic content-aware replacement for reducing the write energy and improving write endurance by the means of minimizing the switch bits. Unfortunately, the optimal write replacement increases the cache miss rate quite substantially, especially for a few workloads with low miss rates, such as soplex, astar, gcc, xalacbmk, and gobmk as shown by the two lines.

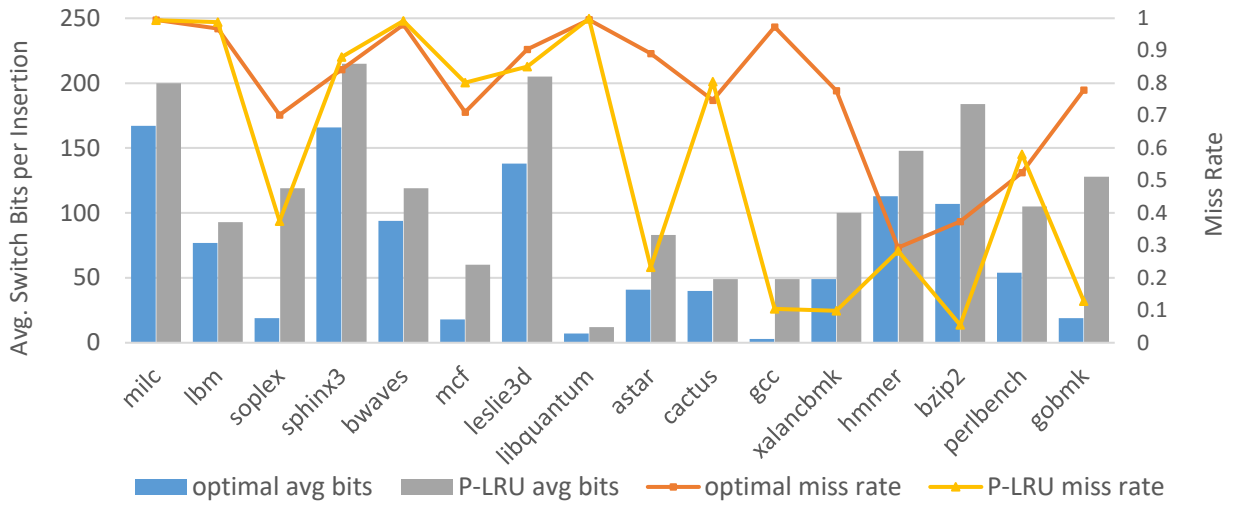


Figure 2-1. Pseudo-LRU vs. Write-Optimal replacements on avg. switch bits and miss rates

The significant increase of miss rates encounters more block replacement. As a result, the total switch bits of the optimal replacement could be even higher than the P-LRU replacement. Furthermore, cache misses incur high latency and additional energy consumption to bring in the block from memory. Therefore, the new content-aware replacement approach must satisfy two conditions. First, it requires a practical way to decide the content similarity without costly fetching and comparing the content of all blocks within the set. Second, the content-aware replacement must maintain comparable cache miss rates in comparison with the P-LRU replacement.

2.2.2 Related Work

Spin-torque magnetic random-access memory (STT-MRAM) has been considered as the most promising SRAM replacement to build on-chip large-scale caches. The STT-MRAM has zero leakage power, non-volatility, comparable read speed and higher density with that of SRAM [21][22]. Good survey papers can be found in [23][24]. Although STT-MRAM has these nice features, it also has major issues in write energy and latency. To reduce the dynamic energy consumption and improve endurance of write operations to a STT-MRAM cache, Zhou et al.

proposed early write termination (EWT), which performed a read operation during the write operation, and terminate the redundant writes which do not change the bit value at their early stages [11]. The Flip-N-Write introduces a few ‘flip’ bits for each cached block to indicate whether the portion of the block content has been flipped to further minimize the actual writes as described in Section 1.

It has been noticed by researchers that block content is an important factor to reduce total switch bits. Jung, et al. discover that a large portion of the written data have zero value [25]. They added all-zero flag for each word or byte. During block writes, only non-zero words or bytes need to be written. Yazdanshenas et al. [26] also report that a high percentage of words in L2 have similar content. They record the most frequent 4-byte patterns in a table and use a few flag bits to indicate the pattern for the 4-byte content. So, only the flag bits need to be updated when a frequent pattern is written into the STT-MRAM caches.

Previous works also reduce the actual writes bits for writeback of dirty lines. In [9], it tracks the dirty sub-blocks of each cache line in high-level SRAM caches. When a data block is evicted from the caches, only the dirty sub-blocks are written into the low-level STT-MRAM cache. Rasquinha et al. design a small fully-associative SRAM write cache that sits between the high-level cache and the STT-MRAM cache to store the dirty lines evicted from the high-level cache [27]. They also modify the cache replacement policy to keep the dirty blocks in the high-level cache longer to avoid frequent writebacks.

To mitigate the impact of high write energy, hybrid STT-MRAM and SRAM caches have been studied [3][28][29][30][31][32], which combine the high density and low power benefits from STT-MRAM and the fast read and write speeds of SRAM. The hybrid cache architectures can be at inter-level or intra-level, and these two types are orthogonal. In the inter-level, higher

level caches usually employ SRAM for fast access while lower level caches usually utilize dense STT-MRAM for large capacity and high hit rate. In the intra-level, the cache level is usually partitioned into two regions in a way-based manner with a small SRAM region and a large STT-MRAM region. The existing work [29][32] discusses various ways to manage the hybrid caches and to swap blocks between the two caches to reduce high energy writes to the STT-MRAM cache.

G. Sun et al. [3] study using STT-MRAM as the L2 cache in CMP systems. They recognize the write energy and latency issues and propose architecture solutions of using a read-preemptive write buffer and a SRAM-STT-MRAM hybrid L2 cache. Z. Wang et al. [28] describe an adaptive block placement and migration policy for hybrid caches. They classify writes to LLC caches into prefetch-write, demand-write and core-write and place a new block into either the STT-MRAM or SRAM part of the cache based on the predicted write class to lower the write overheads. J. Hu et al. [31] study hybrid on-chip SRAM and non-volatile memory for embedded processors.

The P-LRU replacement algorithm is commonly used in caches with high set-associativity and large capacity [33]. We applied the binary-tree scheme to implement the P-LRU policy in this chapter. Hamming Distance has been used in measuring the content difference as discussed in [26]. To reduce the actual write bits, they record frequent word patterns for encoding the word content with a few flag bits. The Hamming Distance has also applied in minimizing the switch bits in the ECC code in STT-MRAM caches [34].

2.3 Encoded Content-Aware Cache Replacement Policy

In this section, we present the details of the proposed ECAR in four parts, a content encoding method, the content-aware replacement based on the P-LRU policy, selection of replacement policy with set-dueling, and a high-level STT-MRAM L3 cache design using the

ECAR. We measure the content similarity of two blocks by their Hamming Distance (HD) which is equal to the number of '1's of the pairwise exclusive-or results between the two block contents:

$$HD(A,B) = \sum_{i=0}^{n-1} A_i \oplus B_i$$

where A, B are two blocks, n is width of the blocks, Ai, Bi are the bit values in the i-th position of the two blocks. For example, if A = (10001010) and B = (10101011), then HD(A,B) = 2. Note that the smaller the HD value indicates higher content similarity between the two blocks.

2.3.1 Block Content Encoding

We introduce a content encoding method. The purpose of encoding is to represent the 512-bit content of a cache block with a few bits which can be used to avoid the comparison of the entire 512 bits. Since our target is to minimize the switch bits, we decide to encode each 8-byte double-word using one bit to indicate state of the corresponding 8 bytes as below.

'U' state (*uniform*): The content of the 8-bytes is dominated by either '0' or '1'. In other words, the number of bits with same value surpasses certain threshold.

'D' state (*diverse*): The content of the 8-bytes is not dominated by either '0' or '1'.

Thus, each block is encoded using just 8 bits, which incur low space overhead. The rationale behind this encoding is that when the contents of two double words are dominated by certain bit value, there is a good chance that the content similarity of the two double words is high, hence may lower the switch bits when one double word replaces the other. For example, previous studies had shown that there was a high percentage of block contents dominated by '0's [16] [17]. On the other hand, if both double words are classified as diverse, their content similarity is largely depending on the relative positions of '0's and '1's among the two double

words. But, replacing one by the other may have smaller switch bits than that when the two double words are in the opposite encoded states. Note that the encoding granularity is flexible. Instead of encoding 8-byte double words, we can encode 4-byte words in a block with the encoding bits doubled.

With content encoding, the similarity of two blocks is measured by the HD of the encoded bits ranging from 0 (most similar) to 8 (least similar). The performance evaluation results show that the proposed simple encoding scheme indeed provides good accuracy in selecting the most content-similar block for replacement to lower the switch bits and will be presented in Section 5. Note that based on our evaluation, we choose an encoding threshold by defining the first state as having 47 bits out of 64 with the same value. Note also that we apply the Flip-N-Write scheme with ECAR. By setting the flip bits, the switch bits can be minimized regardless whether the content is dominated by '0's or by '1's.

2.3.2 Selective Content-Aware Pseudo-LRU Replacement

Next, we introduce a content-aware P-LRU replacement scheme to help the cache miss rate. Given high set-associativity in lower-level caches, the P-LRU replacement is commonly used for limiting the space overhead in each cache set to (k-1) bits where k is the set associativity. The P-LRU position is determined by a binary tree where the leaf nodes are the k physical cache frames. There are $\log_2 k$ upper levels in the binary tree where each node maintains a single bit.

The LRU frame can be located by traversal from the root. When the bit value is '0', the LRU block is on the left subtree and the LRU block is on the right subtree when the bit value is '1'. When a cache access occurs, it either hits a cache frame or it encounters a miss and the block in the LRU frame is replaced by the missed block. In either case, the bit values of the nodes from the root to the hit frame or to the replaced frame must be updated. Regardless the original value,

the new value for the nodes along the traversal path is set to the opposite subtree of the hit or the replaced frame. After updates, the new MRU block can thus be protected when searching for the LRU frame again for replacement.

Figure 2-2 illustrates a 16-way binary tree for the P-LRU replacement. We assume four consecutive misses and illustrate their LRU frames, 5, 11, 1, and 13 as indicated by red, green, blue, and purple paths after each miss is processed. The binary values along the paths are listed and toggled after the missed block is placed in the LRU frame. Different intensities of the shaded nodes indicate different levels of the subtrees. In the following, we describe the implementation details of the proposed ECAR based on a selective P-LRU scheme for remedying the damage to cache miss rate.

Since the node's binary bit is toggled along the way from the root to the P-LRU replacement frame, the paths to the LRU frames for consecutive misses are changed from left to right and vice versa at each level of the binary tree. Therefore, the four consecutive P-LRU frames are located in four regions; each has four consecutive cache frames. Instead of replacing the block in the first P-LRU frame, we select the replaced block from a set of candidate frames where each candidate is the P-LRU frame located in each region of four consecutive frames. For example, in Figure 2-2, frames 1, 5, 11, and 13 are the P-LRU frames in the four regions. We can then use the encoded contents of these candidates to select the one with highest similarity with the missed block for replacement. Such replacement selection takes the advantage of content similarity for reducing the switch bits while alleviates the negative impact on the cache hit rate by restricting the replacement selection within a few blocks closer to the LRU position.

In the above example, the four P-LRU frames are determined by four consecutive misses. We use the blocks in these frames to approximate the near LRU blocks in the set. In real cache

accesses, however, both hits and misses are interleaved and the consecutive P-LRU frames may not be distributed evenly. Nevertheless, during a cache hit, the binary values along the path are still changed to the opposite subtree to protect the hit frame where the MRU block resides. By selecting the P-LRU blocks in each region, it still can reasonably pick the candidate blocks closer to the LRU position.

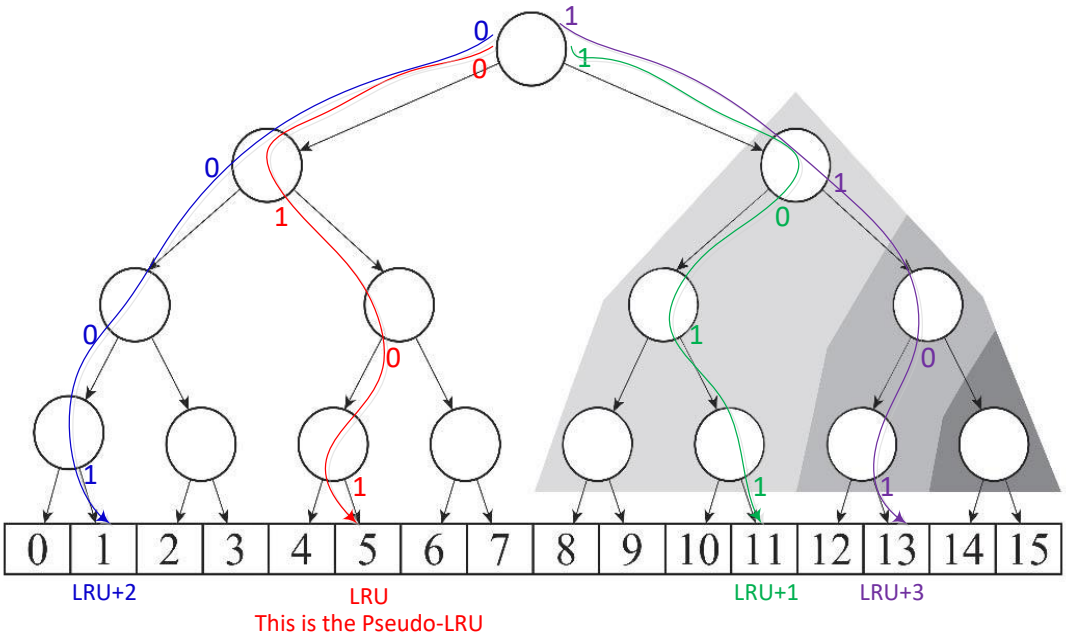


Figure 2-2. Pseudo-LRU replacement using a binary tree

2.3.3 Replacement Selection with Set Dueling

From detailed performance evaluation of SPEC 2006 workloads, we found that in a few workloads, the P-LRU replacement works better than the content-aware replacement for the total switch bits. This is due mainly to an increase of cache miss rate using the content-aware replacement. Therefore, we consider using the set-dueling technique [35] for dynamically selecting a better replacement policy between the ECAR and the P-LRU replacements. We put aside two groups each with a small number of sampled cache sets dedicated to experiment both

replacement policies. We examine the total switch bits between the two policies periodically and apply the policy with less switch bits for the rest of the cache sets.

2.3.4 High-Level Design of Content-Aware Replacement

Figure 2-3 shows the high-level schematics of the encoded content-aware replacement (ECAR) for a 3-way set-associate STT-MRAM L3 cache. When a L3 cache miss occurs, the missed block is fetched from memory. The encoding bits of the missed block are constructed when the block comes back and compared with the encoding bits of the selected LRU blocks to determine their content similarity. Given the Hamming Distance results, the weighted ECAR Logic selects the block with the minimum HD value as the replacement victim.

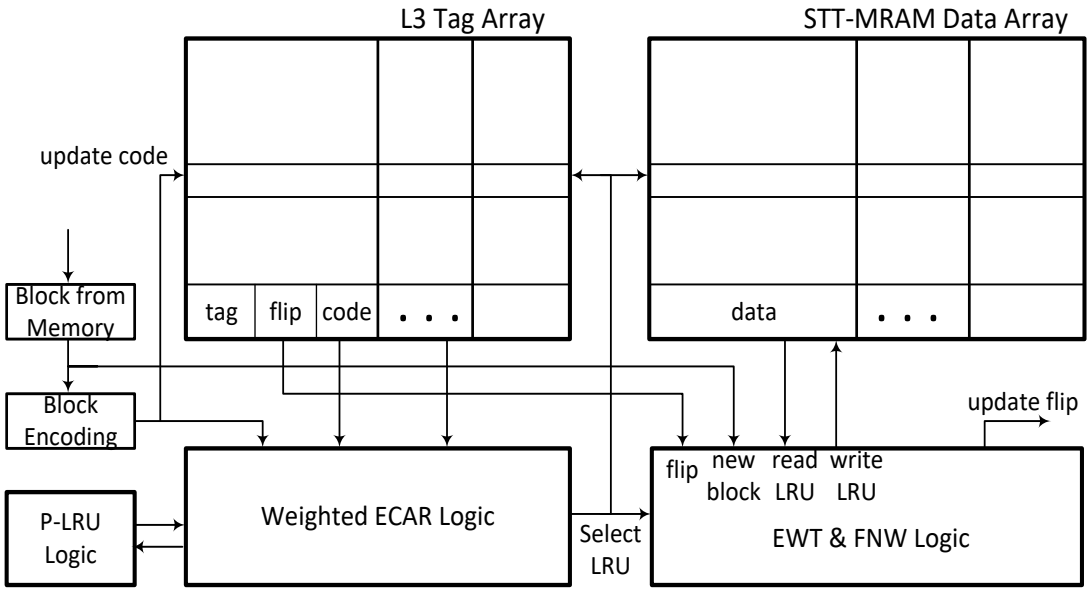


Figure 2-3. Schematics of Content-Aware replacement

The EWT & FNW Logic implements Early Write Termination and Flip-N-Write schemes to minimize the switch bits of each block insertion. It first uses the flip bits of the victim block to recover the correct content and writes the block to memory if the victim block is dirty. When inserting the new block into the frame, the EWT & FNW Logic switches only necessary bits

according to the content of the new block and actual data bits stored in the cells of that frame. During this process, the flip bits and the encoding bits of the new block are set accordingly and saved in the L3 tag array along with the tag of the new block. Note that from our design, the blocks with lower Hamming Distance in encoding bits are more likely to be selected, which has high similarity with the encoding bits of the missed block, hence the extra switch of the encoding bits is also minimum.

In this work, our goal is to minimize the total switch bits for placing blocks into STT-MRAM L3 cache. It is well-known that the frequency of switching bit value among cache locations is unbalanced and causes endurance concerns. Wear-leveling techniques have been studied as in [36] to resolve this issue and are orthogonal to our design. Further discussion on wear-leveling is out of the scope.

2.3.5 Overhead Analysis

This section analyzes the latency, area and power overheads with the block content encoding and the ECAR logic. First of all, the encoding circuits and the ECAR logic are off the critical path since the missing data block from memory will be sent directly to L1/L2 before cache replacement in L3. Second, with an efficient Hamming weight comparator proposed by Parhami [37], the content encoding circuits to count number of '1's are fast with O (log n) levels of one-bit full adders where n = 64 in our design. The entire encoding and ECAR circuits are estimated for 0.6 ns latency with the full adder design in [38]. As we addressed before, this extra replacement delay is off the critical path, so the overall cache performance will not be jeopardized. Third, the encoding byte overhead is 256 KB for 16 MB L3 cache, which costs about 1.4% area overhead (including tags) as shown in Table 2-1. Furthermore, from results in [37], the encoding circuits come with O(n) complexity with respect to the number of gates, which results in a minimal impact on power consumption as well.

2.4 Evaluation Methodology

We apply the trace-driven simulation to simulate sufficiently long and representative program regions for studying performance of large STT-MRAM caches. The memory traces are generated on the QEMU multi-core emulation virtualizer running Linux with the KVM kernel module [39], which contain both memory addresses and the corresponding data content. These memory traces are processed through the AMD x64 memory hierarchy configuration with private L1, L2 SRAM caches and a shared STT-MRAM L3 that interfaces with DRAM main memory, as shown in Table 2-1.

Table 2-1. Simulation system configuration

Component	Parameters
Private L1 SRAM Cache	I/D split, 32KB, 8-way, 64B block SRAM
Private L2 SRAM Cache	512 KB, 8-way, 64B block SRAM
L3 STT-MRAM Cache	32-way, 64B block *Capacity*: 4 MB for single-core; 16 MB for quad-core *Content Encoding*: 1 bit per 8 bytes, '0' state if 47 bits are same *Content-aware Replacement*: Four Pseudo-LRU candidates from 4 8-frame regions in 32-way cache *Space overhead***:** 256KB for 16MB L3 *Flip-N-Write***:** 1 bit per 8 bytes
DRAM	8GB, DDR3-1600, 64bit I/O, 8 banks, 2KB row buffer, tCL-tRCD-tRP-tWR: 11-11-11-12

For our study need, we collect the L2 miss traces to drive a STT-MRAM based shared L3 cache. We use the Sniper simulator [30] and modify the cache controller inside the Sniper to support the STT-MRAM L3 with various replacement policies. We implement the proposed ECAR and count the total switch bits as the measure on the STT-MRAM endurance. For 8-bit

content encoding, each bit represents 8 bytes in a block: the bit is '0' when 47 or more out of 64 bits are the same, otherwise the encoding bit is '1'. For Flip-N-Write, there are 8 flip bits, each bit indicates whether the corresponding 8 bytes of data are flipped. Furthermore, we provide an accurate dynamic energy evaluation for the STT-MRAM cache. The energy consumption parameters of a 22 nm STT-MRAM based 16MB L3 Cache are estimated using NVSim [16]. Different from a conventional SRAM cache configuration with a constant write/read dynamic energy, the write energy for a cache block varies considerably per the amount of switch bits, no longer a fixed value used in [11].

Table 2-2. Multicore workload mixes

Mixes	Benchmarks
mix 1	bzip2 astar gobmk mcf
mix 2	mcf soplex perlbench bwaves
mix 3	soplex cactus bzip2 gcc
mix 4	perlbench sphinx3 libquantum lbm
mix 5	milc gcc xalancbmk gobmk
mix 6	lbm mcf cactus soplex
mix 7	astar lbm gobmk sphinx3
mix 8	bwaves libquantum gobmk leslie3d
mix 9	hmmer soplex milc perlbench
mix 10	leslie3d soplex astar bzip2
mix 11	gcc sphinx3 bwaves leslie3d
mix 12	cactus leslie3d hmmer milc

We evaluate both single-core and multi-core workloads [26]. For single-core, we evaluate all benchmarks from SPEC CPU2006 and selected 16 applications with high L3 MPKI and large amount of total switch bits during simulation. We collect L2 misses from 10 billion instructions of each program. For multi-core evaluations, we run four applications on QEMU and collect L2 misses from 40 billion instructions. Similar to those multi-program workloads used in [28], we

present the results with twelve different quad-core mixes of SPEC CPU2006 benchmarks as listed in Table 2-2.

2.5 Performance Results

We present the performance evaluation results for the total switch bits, the cache miss rate, and the energy consumption in this section. Conventional P-LRU replacement without any enhancement for reducing the switch bits is used as the baseline in our evaluation. We compare the improvement of three enhancement methods: 1. Early Write Termination and Flip-N-Write (EWT+FNW); 2. The Encoded Content-Aware Replacement along with EWT+FNW (ECAR); and 3. ECAR plus dynamic Set Dueling method (SD) (ECAR+SD). Sensitivity studies of the proposed ECAR with respect to the encoding threshold and the encoding granularity, are also presented.

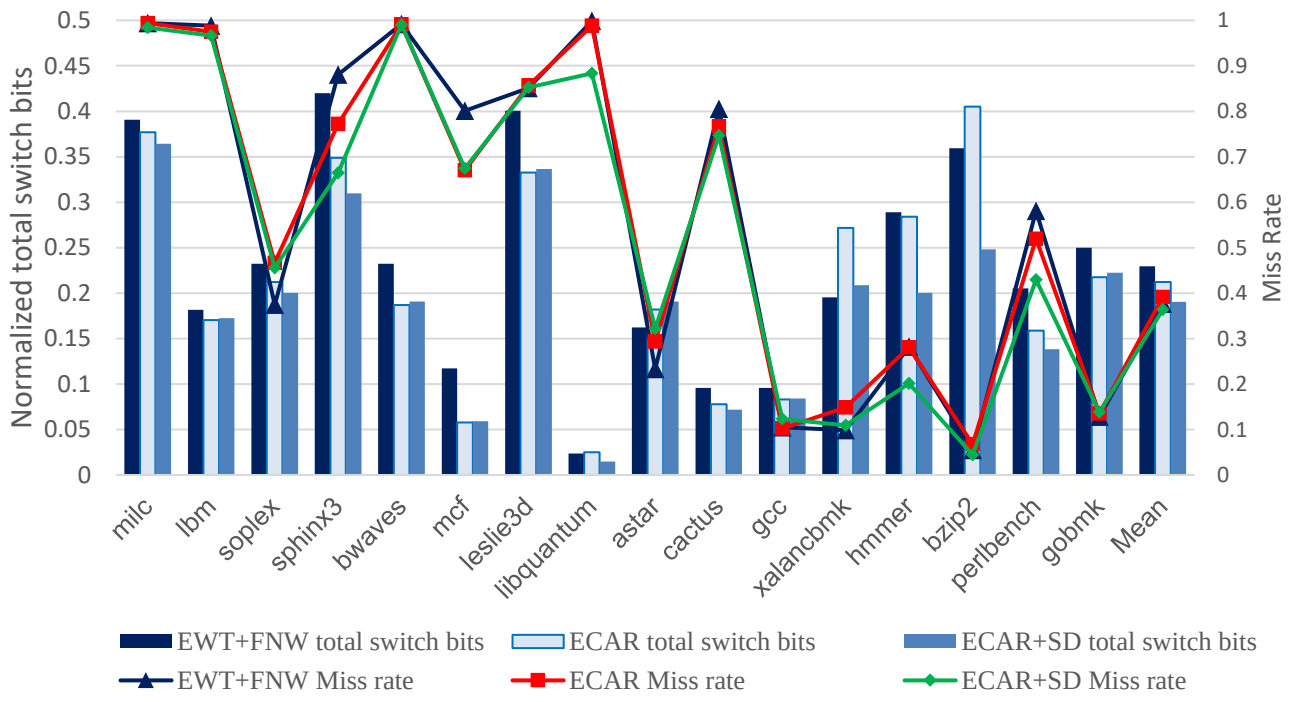


Figure 2-4. Comparisons of total switch bits and cache miss rates – single-core

2.5.1 Total Switch Bits, Miss Rate, and Energy Consumption

In Figure 2-4, we show the single core results for the switch bits and the cache miss rate. The left Y-axis indicates the total switch bits normalized by the switch bits in the baseline model while the right Y-axis is the L3 cache miss rate. The three bars for each workload are normalized total switch bits and the three lines connect the miss rates for the three enhancement schemes, EWT+FNW, ECAR, and ECAR+SD.

We can make several observations. First, with same replacement policy and exact same amount of cache insertions, the EWT+FNW effectively cut the total switch bits to 23.1% compared to the P-LRU baseline, implying that the average switch bits of each insertion are 117 out of 512 for a 64B block. Using ECAR, the normalized total switch bits are about 21.2%, and by including Set-Dueling, the total switch bits are further reduced to 19.0%, showing improvements about 9% and 18% respectively.

Second, Set-Dueling is effective for sphinx3, libquantum, xalancbmk, hammer, and bzip2 in reducing the total switch bits as a result of lower miss rate. Further examine the results, we also find out that ECAR performs poorly also because of matching ‘diverse’ state in calculating the HD. The switch bits between two double-words in ‘diverse’ state vary widely when one replaces the other due to the higher distribution randomness compared to ‘uniform’ state. This is true especially for xalancbmk and bzip2. In these two workloads, ECAR got negative improvement in switch bits due to high diverse-state matching. Periodical switch with ECAR and P-LRU based on the total switch bits can help the cache miss rate and more importantly to alleviate the negative effect of matching the diverse state for these two workloads.

Third, the selective ECAR from a few candidate blocks near the LRU position indeed help the cache miss rate. In contrast with the drastic increase in miss rates in Figure 2-1, the miss rate of ECAR changes within a much smaller range. In fact, ECAR reduces the miss rate for

spinx3, mcf, hammer, and perlbench, due mainly to memory reference behavior in individual workloads. With Set-Dueling, the miss rates are further reduced for a majority of the workloads, especially for sphinx3, libquantum, hammer, and perlbench.

Figure 2-5 shows multi-core results. For all multi-program workload combinations shown in Table 2-2, the Set-Dueling has little help to further improve the total switch bits of ECAR, hence we did not include ECAR+SD results in the figure. This is due to the fact that ECAR almost always performs better than P-LRU, therefore, including Set-Dueling makes little difference by always choosing ECAR over P-LRU. Once again, EWT+FNW alone makes a significant reduction, and on average the total switch bits are down to 29.3% compared to the baseline model. Moreover, all multi-program quad-core workloads can benefit from ECAR by reducing total switch bit to 23.3% on average, which is 20.5% better than EWT+FNW. We also noticed that for all 12 quad-core mixes, the negative impact on the cache miss rate from ECAR almost disappears. The overall reduction on the total switch bits is clearly better than that in the single-core workloads.

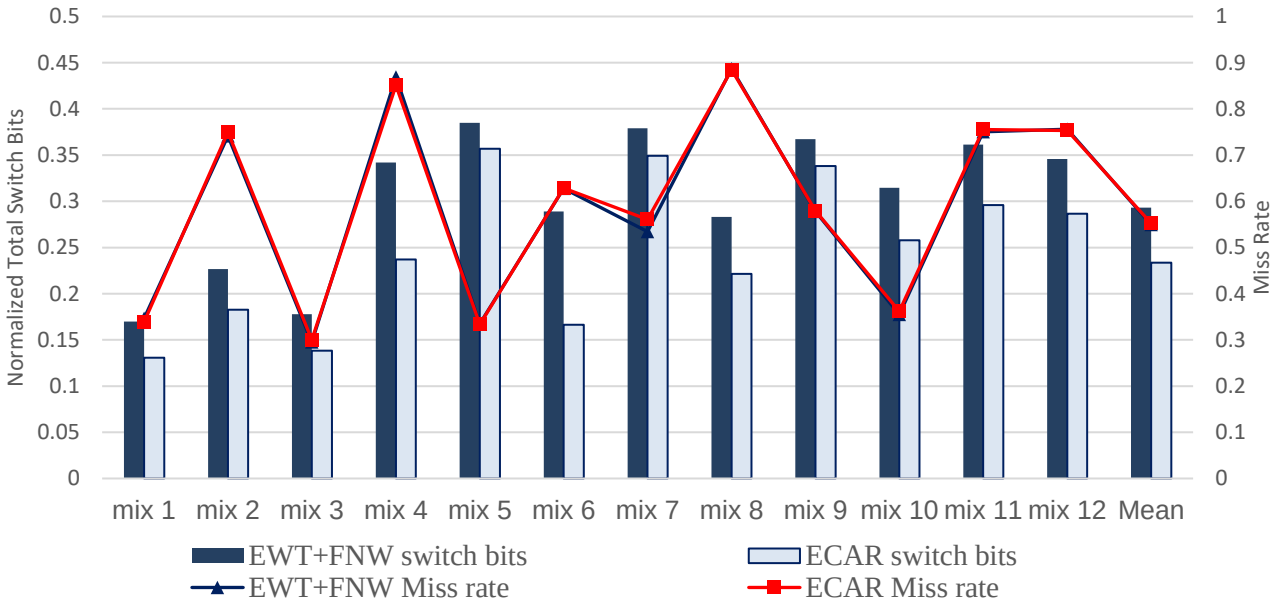


Figure 2-5. Comparisons of total switch bits and cache miss rates – quad-core

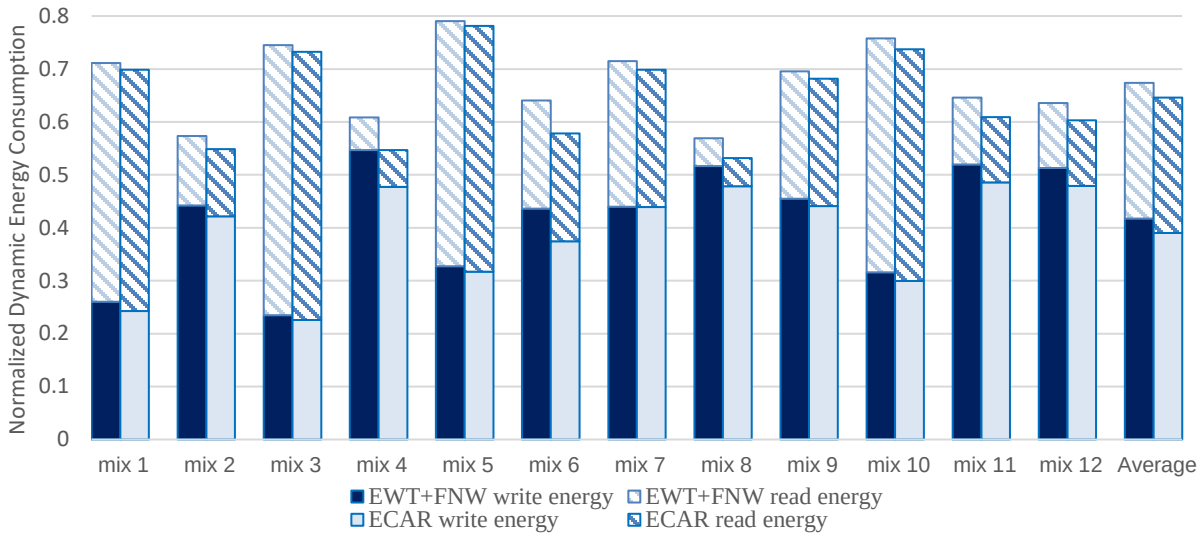


Figure 2-6. Normalized dynamic energy consumption – quad-core

The energy reduction is shown in Figure 2-6 in which the Y-axis is the normalized energy consumption with respect to the baseline model. Each workload has two bars showing the dynamic energy consumption of the EWT+FNW and the ECAR methods. Total dynamic energy is composed of write energy as shown with bottom solid bar and read energy as top gradient bar. Among all quad-core workloads, there is little difference for read energy consumptions between both replacement schemes since their cache miss rates are almost identical. The major reduction comes from savings in the write energy due to less total switch bits. Mix 4 and mix 6 show considerable savings on write energy in ECAR compared to EWT+FNW with 14.2% and 15.3% improvements respectively, which are consistent with the significant drop in total switch bits for these two workloads as shown in Figure 2-5. On average, EWT+FNW helps reduce write energy to 41.9% in comparison with the baseline, while the ECAR can further reduce the energy to 38.5%, as 8.1% improvement over EWT+FNW.

2.5.2 Sensitivity Study

The sensitivity of ECAR with respect to the encoding threshold and the granularity of encoding are discussed in this section. For our primary interest in parallel computation

environment, we only present the quad-core results. Note that when we do sensitivity on one parameter, we keep the other parameters unchanged. With the primary target on improving write endurance, we measure the sensitivity based on the total switch bits.

The encoding threshold can affect the result of encoding thus change the victim selection during cache replacement. To find the optimal threshold option, we test different ECAR configurations with thresholds of 39, 43, 47, 51, and 55. The results demonstrate that, threshold 47 has the overall lowest total switch bits in most cases. Figure 2-7 shows the normalized total switch bits based on the encoding threshold of 47. The total switch bits increase by 2.3% 0.5%, 2.9%, and 7.5% for thresholds 39, 43, 51 and 55 respectively. Due to diversity of the workload mixes, their behaviors are quite different. For example, threshold 39 has the worst switch bits by a large margin for mix 1 and 3. However, it has the lowest switch bits for mix 4 and 7 even lower than the threshold 47. Threshold 55 has the highest overall total switch bits, especially for mix 4 and mix 6.

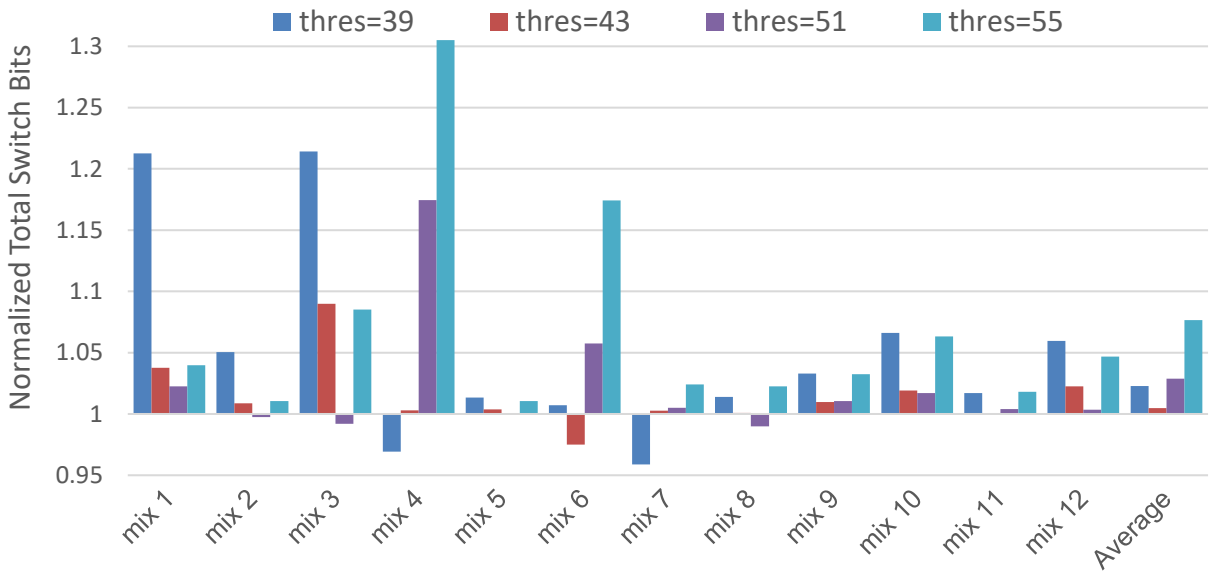


Figure 2-7 Encoding threshold sensitivity for normalized total switch bits
(based on threshold = 47 bits)

Next, we present the sensitivity of the encoding granularity in Figure 2-8. As expected, smaller encoding granularity provides higher accuracy in content similarity representation and improves the total switch bits with the cost of extra space overhead for more encoding bits storage. It is important to note that in this simulation, we also change the granularity of FNW operations to match with the encoding granularity, and shrink the encoding threshold proportionally. Again, different mixes show different behaviors. In mix 6, three out of four workloads, milc, soplex and cactus are write-intensive floating-point applications generally with more '1's than '0's inside the cache content, which would not benefit much from the FNW. With less FNW operations, the smaller encoding granularity could play a similar role as higher encoding threshold, which leads to the increase of total switch bits in some workloads.

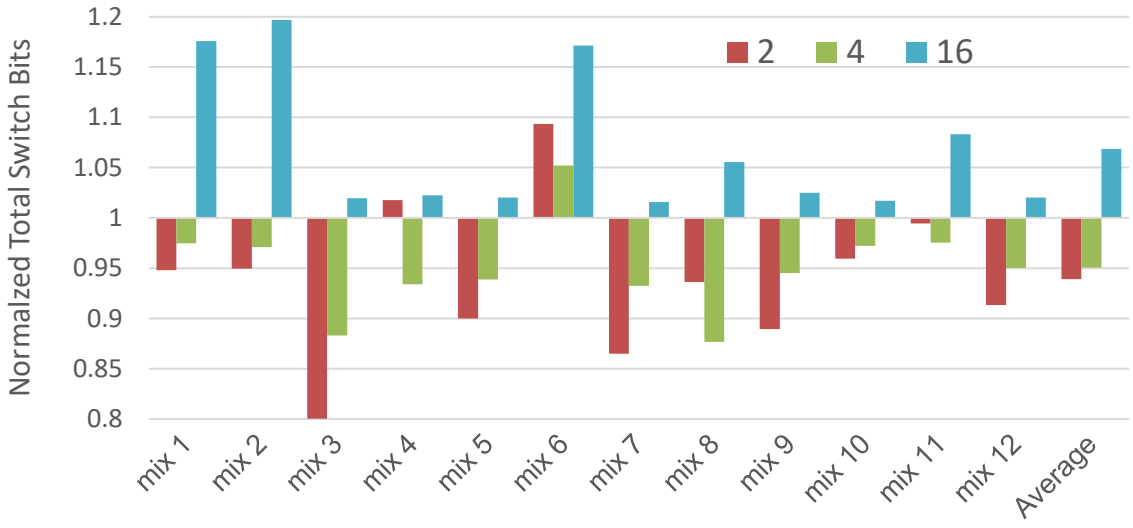


Figure 2-8. Encoding granularity sensitivity for normalized total switch bits (based on granularity = 8 bytes)

2.6 Summary

We presented in this chapter a new encoded content-aware replacement policy, or ECAR for STT-MRAM caches to reduce the total switch bits for cache replacement with minimum

impact on the cache performance. To accomplish this goal, ECAR selected the replacement victim block which had the most similar content with the missed block from a set of blocks close to the LRU position.

Experimental results with a set of multi-program workloads showed that our technique helped reducing the total switch bits by 20.5% with low complexity and overhead. With significant reduction in the total switch bits, the endurance of STT-MRAM cache could be improved. Meanwhile, it also reduced the energy consumption by 8.1%.

CHAPTER 3
TOWARDS ENERGY-EFFICIENT MLC STT-MRAM CACHES WITH CONTENT AWARENESS

3.1 New Challenge of MLC STT-MRAM

Recent trends in CMOS integration and scaling lead microprocessor manufacturers to integrate more cores in a single chip. As the number of cores grows, the size of on-chip cache grows as well to offset the limitation imposed by the off-chip memory bandwidth. Lately, the on-chip cache size of the Intel Xeon E7 launched in Q2'16 has been increased to 60 MB. Such large capacity of on-chip caches occupies a significant portion of the chip area and power budget under the conventional SRAM technology. Spin-Transfer Torque Magnetoresistive Random-Access Memory (STT-MRAM) has high density, low leakage power, fast read speed, and non-volatility making it a promising candidate to replace SRAM for future on-chip caches.

In write operations, the transition energy of the four resistive states in 2-bit MTJ device varies widely depending on both the original and the new 2-bit values. There have been many proposals to lower the write energy and increase write endurance for MLC STT-MRAM caches [40][41]. We present a new dimension to reduce MLC write energy. We focus on the write energy when a missed block is brought into the cache. During each cache replacement, we intelligently test existing cache block contents and select the replacement block which consumes the lowest write energy.

3.2 Motivation and Related Work

3.2.1 Observation and Opportunity

. From Figure 1-3 (b), we can observe that write energy is determined by both the replaced 2-bit value and the newly allocated 2-bit value. Existing proposals focus on different encoding methods to lower the write energy by remapping the new 2-bit values based on their

occurrences to reduce high energy transitions [10][40][41] . However, they fail to recognize opportunities to select the replacement block with considerations of minimizing the write energy. In figure 3-1, we compare average write energy per replacement of 4MB MLC STT-MRAM L3 cache using practical Pseudo-LRU replacement policy against an optimal replacement which selects the replacement block with the lowest write energy. We use SPEC workloads and Gem5 simulation tool to drive our simulations. Detailed description of the workloads and simulation environment will be given later.

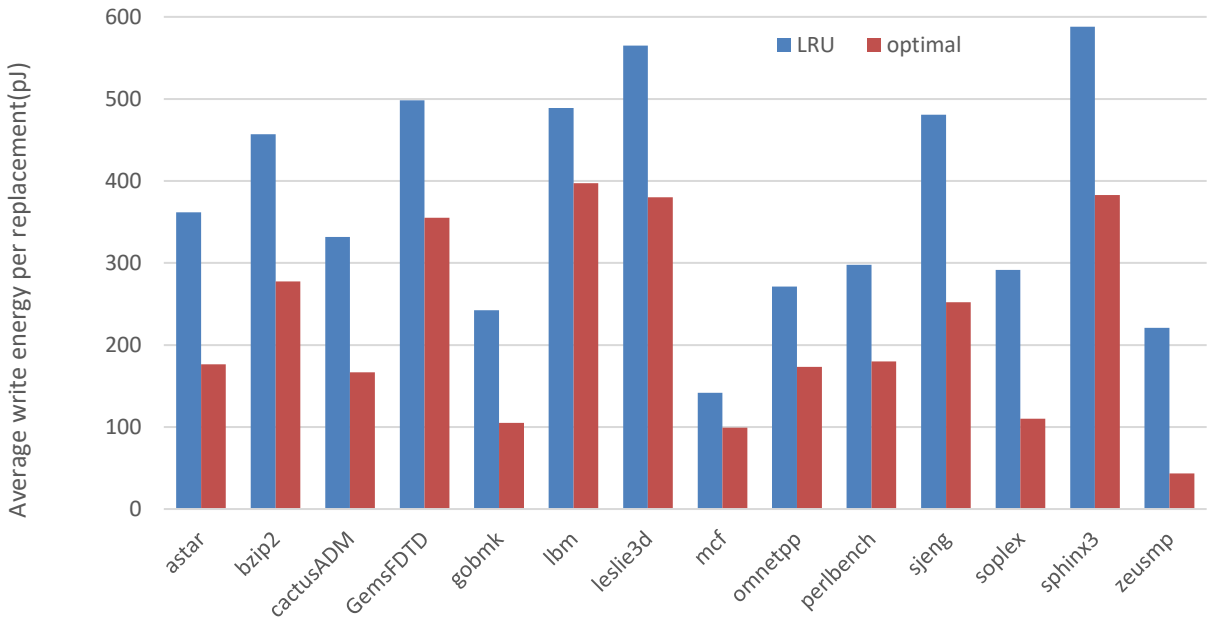


Figure 3-1. Comparisons of average write energy per replacement

We can notice significant difference in average write energy consumption per replacement between LRU and optimal for each workload, 39.1% on average, which show potential improvement could be achieved by intelligent replacement policy. Workloads such as zeusmp, soplex and gobmk show huge energy gap between the two replacement policies. Note that in this oracle study, we examine the contents of all blocks in the target set and replace the block with lowest write energy without considering the impact on the cache miss rate. The

proposed content-aware, energy-efficient replacement method addresses both issues for lowering the write energy and will be presented next section.

3.2.2 Related Work

There have been many proposals for lowering the write transition energy for MLC caches. Y. Chen et al. [40] calculate the average energy consumption based on the new bit values and found that R11 < R00 < (R01, or R10). A new encoding method based on the frequency of the write values is proposed. They report that 00 has the highest 37% of the new 2-bit value and should be mapped to the lowest average write energy R11, similarly for the rest mappings 01 → R10, 10 → R01, and 11 → R00 for achieving the lowest overall write energy. They also propose a wear-leveling technique to improve endurance. P. Chi, et al. [10] enhances this static encoding to dynamic encoding. They use counters to collect the frequencies of new 2-bit values and dynamically modify the encoding to minimize the energy consumption.

Y. Chen et al. [7] observe that write energy varies due to both the original and the new 2-bit values. They propose reading the old 2-bit content before deciding the encoding the new 2-bit for energy-efficient write transitions. Recently, H. Luo, et al. [41] propose an interesting two-step encoding method. They consider both the old and the new 2-bit values and use an auxiliary bit for remapping the 2-bit value to 3 bits. For efficient use of the 2-bit MTJ cell, they use three cells (6 bits) to encode two cells (4 bits) of data with an extra cell to save two auxiliary bits. By extending four bits to six bits, they can minimize the TT transaction with the cost of extra 50% of storage overhead.

The two-step operations for reads and writes to MTJ of MLC STT-MRAM are described in [7]. Based on the physical principles of the resistance state transition of MLC STT-MRAM, they propose a read circuitry based on Dichotomic search algorithm and three write schemes

with various design complexities – simple, complex, and hybrid schemes. In [42], they observe that each 2-bit MLC cell consists of one hard-bit and one soft-bit. The hard-bit is fast to read but slow to write while the soft-bit is fast to write but slow to read. They propose line pairing and line switching techniques by separating cache blocks into read-fast-write-slow (RFWS) and read-slow-write-fast (RSWF) lines and dynamically promote write-intensive data to RSWF lines and read-intensive data to RFWS lines.

J. Wang et al. [43] propose to dynamically reconfigure the cache block size for MLC STT-MRAM last-level caches. They place certain hot data chunks in smaller blocks to fit into cells in soft-domain, benefit from the lower energy and latency, while keeping the rest in larger blocks to maintain an overall hit rate. X. Chen et al. [44] consider serial-MLC based caches. The cache blocks in each cache set is separated into the soft-line and hard-line regions. They observe that restoration of soft bit in two-step write operation is unnecessary if the block located in the soft-line region still exists in higher-level caches and predicted not to be reused before eviction. Lately, data compression technique is applied to lower MLC write energy [45]. They compress cache blocks to fit into the soft-domain region to lower the write energy and to increase capacity of the hard-domain region to further improve system performance.

3.3 Content-Aware, Energy-Efficient Cache Replacement

In this section, we present design of the proposed Content-Aware, energy-efficient cache REplacement (CARE) method. We first introduce some key terminologies and data structures. A data block is divided into segments of n-byte called chunks to achieve proper level of granularity for our design. For each chunk, we use 1-bit as encoding and 2-bit as remapping choice for selecting the replacement block to lower the write energy.

On a cache miss, cache replacement decision logic first encodes the block from main memory, then chooses the victim block by matching the encoding bits of candidate blocks and

those of the incoming block. Finally, for each chunk, remapping logic decides an energy-efficient remapping choice for writing the chunk into data array. The encoding and the remapping bits are recorded in the corresponding tags in the tag array. Detailed descriptions are given next.

3.3.1 Observation and Opportunity

We first introduce a block content encoding method. The purpose of encoding is to represent 512-bit content of a cache block using a few bits which can be used for selecting a replacement block with lower write energy [16]. From Figure 1-3, we can observe that for every two-bit tuple in the block represents hard and soft domains. A switch of high-bit causes high energy transitions, HT and TT, whereas switching only the low-bit causes low energy transitions, ZT and ST. Therefore, we decide to encode the content based on the high bits of such two-bit tuples. Using this observation, we represent an n-byte chunk of data using one encoding bit:

'U' (uniform) **chunk**: When the value of high-bit of all two-bit tuples in the chunk, is dominated by either '0' or '1'. In other words, the number of high-bits with same value surpasses certain threshold, represented with encoding bit '1'.

'D' (diverse) **chunk**: When the value of high-bit in the chunk is not dominated by either '0' or '1', represented with encoding bit '0'.

For convenience, we define the dominating value of a U chunk as its direction, either 1 or 0. Note that the encoding chunk size is flexible. Smaller chunks generally perform better, but require more encoding bits with extra storage cost.

3.3.2 Replacement Block Selection

We want to use the encoded chunks to find the replacement block with low write energy. Based on the encoding method, if a U chunk in the block to be evicted is at same position as a U

chunk in the incoming block (we call it a *UU* pair), the chance of low-energy 0→0 and 1→1 same high-bit transitions (ZT and ST) is better. For example, previous studies had shown that there was a high percentage of block contents dominated by '0' s[25][33]. The profiling results from SPEC CPU2006 workloads presented in Figure 3-2 show that the UU pair consumes significantly less write energy than other three combinations, UD, DU, DD for the incoming and replacement chunks. This possibility of low energy transition is further augmented by block content remapping described later.

Next, we describe a content-aware Pseudo-LRU (P-LRU) replacement scheme to ease negative impact on cache miss rate. Given high set-associativity in lower-level caches, P-LRU replacement is commonly used for limiting the space overhead in each cache set to (k-1) bits where k is the set associativity [19]. The P-LRU position is determined by a binary tree where the leaf nodes are the k physical cache frames. To alleviate negative impact on performance, we partition physical cache frames into equal size regions and choose the LRU frame in each region as the candidate blocks for replacement. The LRU frames from each partitioned region may not be exact blocks closest to the LRU block. However, by selecting the P-LRU blocks in each region, it still can reasonably pick the candidate blocks closer to the LRU, limiting the impact on the cache miss rate.

Thus, our energy efficient selection evicts one of candidate blocks with maximum number of UU pair when matching with the incoming block. For some replacement situations, the number of UU pairs is lower than certain number (in our design the threshold is set as 2) across all candidates blocks, then the selection based on UU pairs makes little impact and we can dynamically roll back to the original P-LRU only for the current replacement.

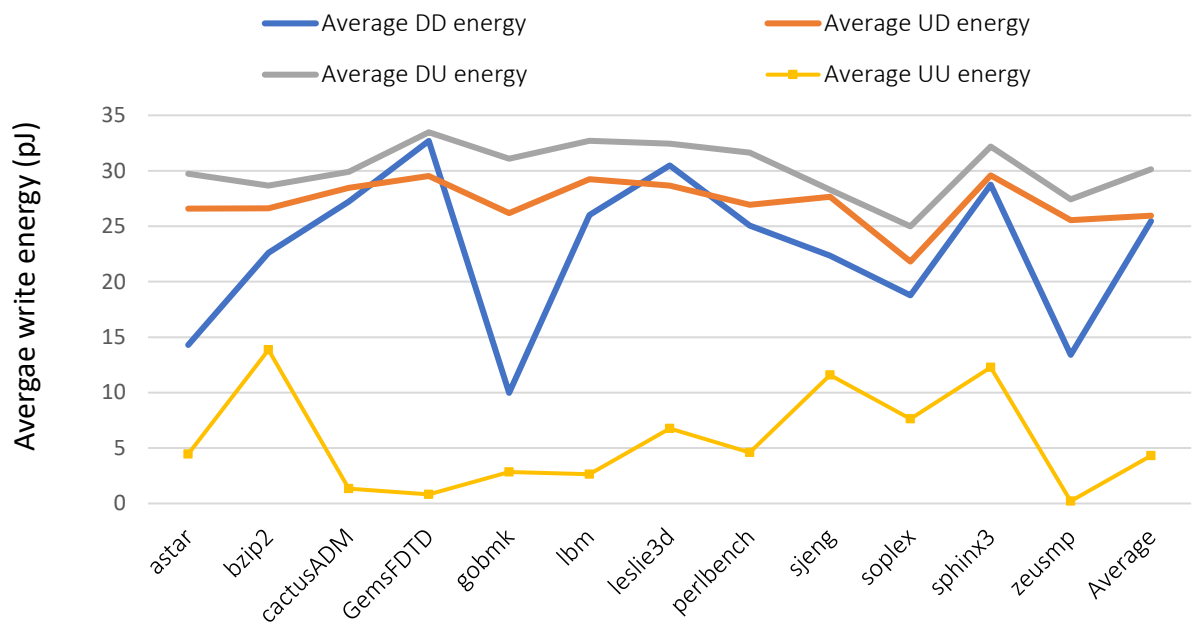


Figure 3-2. Energy profiling results for different chunk pair

3.3.3 Remapping of 2-bit Value

In the previous section we argued that, if the two chunks at same position are both encoded as U, the energy transition will be low. This is intuitive if both U chunks are in the same '1' or '0' direction, but if their directions are different, like one U chunk with all 0s and the other one with all 1s, then remapping of the content of the incoming block can re-align both chunks to the same direction.

Simple remapping from the logic value, L_{xx}, of two-bit tuple in each chunk to cell resistance values, R_{yy}, can help re-align the two chunks for better energy transitions [10]. For the sake of aligning opposite U chunks, we prefer the remapping types that reverse the chunk direction. It is important to note that we want to prevent remapping operations from changing the encoding state of a U chunk to a D chunk, otherwise it will impact the future victim selection. Keeping this in mind, we pick three state-preserving remapping types that change the value of

high-bit. We keep the original (no-remapping) as the fourth type. Table 3-1 summarizes the four remapping types.

Table 3-1. Remapping bits lookup table

Type	Code	Remapping logic states to resistance states
Original	00	$L_{00}\rightarrow R_{00}$, $L_{01}\rightarrow R_{01}$, $L_{10}\rightarrow R_{10}$, $L_{11}\rightarrow R_{11}$
Flip All	10	$L_{00}\rightarrow R_{11}$, $L_{01}\rightarrow R_{10}$, $L_{10}\rightarrow R_{01}$, $L_{11}\rightarrow R_{00}$
Flip Hi	01	$L_{00}\rightarrow R_{10}$, $L_{01}\rightarrow R_{11}$, $L_{10}\rightarrow R_{00}$, $L_{11}\rightarrow R_{01}$
Mixed Flip	11	$L_{00}\rightarrow R_{10}$, $L_{01}\rightarrow R_{11}$, $L_{10}\rightarrow R_{01}$, $L_{11}\rightarrow R_{00}$

3.3.4 Overhead Analysis

This section analyzes the latency, area and power overheads with the block content encoding and the CARE logic. First, the encoding circuits and the CARE logic are off the critical path since the missing data block from memory will be sent directly to L1/L2 before cache replacement in L3.

Second, with an efficient hamming weight comparator proposed by Parhami [38] the content encoding circuits to count number of ‘1’s among high bits are fast with O (log n) levels of one-bit full adders where n = 32 in our design. The entire encoding circuits are estimated for 2 cycles latency with the full adder design in [39]. As we addressed before, this extra encoding and replacement delay for a write is off the critical path. On the other hand, the decoder placed on the critical path of the cache read operation for recovering remapped data is simple and fast, which increases cache read latency by one cycle as presented by Hong et al. [47]. Our experimental results show that the extra delays from the encoding and remapping procedures have negligible impact on overall performance.

Third, for each 4-byte chunk, there are 1 encoding bit and 2 remapping bits, so overhead are 128 KB and 256 KB respectively for 4 MB L3 cache, which costs about 8 % area overhead in

total (including tags) as shown in Table 3-2. Furthermore, from [20], the encoding logics come with O(n) complexity with respect to the number of gates, which results in a minimal impact on power consumption as well and is considered in the energy evaluation.

3.4 Evaluation Methodology

The evaluation focuses on the energy consumption and system performance of STT-MRAM. Gem5 [46], a cycle accurate system simulator, is employed to evaluate Dynamic Resistance Remapping (DRMP), two-step transition minimization (TSTM) and the proposed CARE STT-MRAM (CARE) cache, while conventional P-LRU MLC without any energy enhancement (LRU) is the baseline. The cache model of Gem5 is modified to support an asymmetric cache read/write latency model and integrated encoding and remapping unit, as well as replacement policy for CARE. The processor is configured as a processor with a three-level cache hierarchy. The configurations of the CPU and memory are summarized in Table 3-2. The proposed L3 cache uses SRAM for tag array and STT-MRAM for data array and the encoding and remapping bits are stored in tag array. A set of 13 benchmarks selected from SPEC CPU2006 [24] benchmark suite is adopted in the experiment.

We first take one checkpoint for each selected workload using SimPoint [15] to locate the most representative phase of whole program execution, and then execute 1 billion instruction from the checkpoint to warm up the cache and continue the simulation for next 200 million instructions to collect statistics.

Table 3-3 summarizes the design details of the evaluated STT-MRAM LCC designs. The read latency of CARE is 1 cycle larger than MLC due to the overhead in the CARE decoder for recovering the remapped data, while it’s 2 cycles for TSMT decoding procedure. The latency and energy parameters of HT and ST operations are derived from the work in [7][41].

Table 3-2. Simulation system configuration

Component	Parameters
Processor	Alpha, 3GHz, out-of-order, 8-issue
Private L1 SRAM Cache	I/D split, 32KB, 8-way, 64B block SRAM
Private L2 SRAM Cache	256 KB, 8-way, 64B block SRAM
L3 STT-MRAM Cache	4 MB, 32-way, 64B block STT-MRAM *Content Encoding*: 1 encoding bit and 2 remapping bits per 4-byte segment, *'U'* state if more than 11MSB bits are same. *Content-aware Replacement*: Four Pseudo-LRU candidates from 4 8-frame regions in 32-way cache
DRAM	8GB, DDR3-1600, 64bit I/O, 8 banks, 2KB row buffer, tCL-tRCD-tRP-tWR: 11-11-11-12

Table 3-3. Different configurations of 4MB STT-MRAM L3 cache

Designs	CARE	TSTM	MLC
Read Latency (cycles)	19	20	18
Write Latency (cycles)		ST/HT: 23 TT: 46	
Extra write overhead (cycles)	14	0	20
Read energy (pJ)	0.6/bit	0.6/bit	0.6/bit
Write energy (pJ)		ST:1.92 HT:3.192	

3.5 Evaluation Results

In this section, the proposed CARE is evaluated against DRMP and TSTM in terms of write energy reduction and the system performance. DRMP is the state-of-the-art of reducing the STT-MRAM write energy by intelligently remapping the incoming block content. We also select TSTM as a competitive scheme to see their write energy change for expanding the data block to eliminate the TT transition. One thing we should keep in mind is that the TSTM design will require 50% extra space overhead for storing 4-bit data into 3 2-bit cells.

3.5.1 Energy Comparison

Figure 3-3 shows the write energy consumption normalized to LRU for all 13 workloads. We compare CARE with DRMP and TSTM for savings in the write energy consumption. In

STT-MRAM cache, the write energy consumption is significantly more than the reading counterpart due to larger current and longer duration when switching the STT-MRAM cell. With content-aware replacement selection and remapping technique we used in CARE, the write energy reduction is 21.9% on average and up to 35.5% for benchmarks with good encoding-based selection efficiency and comparable miss rate, e.g. zeusmp and soplex. Besides, with lower miss rate as shown in Figure 3-6, mcf and perlbench also show really good write energy reduction

The DRMP performance varies a lot among workloads, and it sometimes worsens the write energy such as lbm and perlbench mainly due to a) relying on LRU instead of opportunistically finding a better candidate like CARE does; b) neglecting impact of outgoing block content and deciding the remapping solely on incoming block; c) coarse remapping granularity as single remapping type for whole block. In fact, the DRMP includes 12 remapping options with 4 bits, significantly more than options used by CARE, but the single-dimensional decision mechanic may not benefit from the remapping flexibility. On the other hand, TSTM manages to eliminate most of the two-step transitions but incurs considerable extra writes to update auxiliary cells, netting in 11% improvement in write energy.

To measure the efficiency of cache line selection in CARE for eviction with respect to write energy consumption in replacement, we rank the four available candidates from P-LRU during each replacement in the order of their energy requirement. We also record the rank of the cache line selected by two policies to get the average statistics on efficiency of selection. The percentage of the best write energy and second-best one shown in Figure 3-5, proves the efficiency and accuracy of our proposed encoding method in selecting cache line with lowest write energy for replacement.

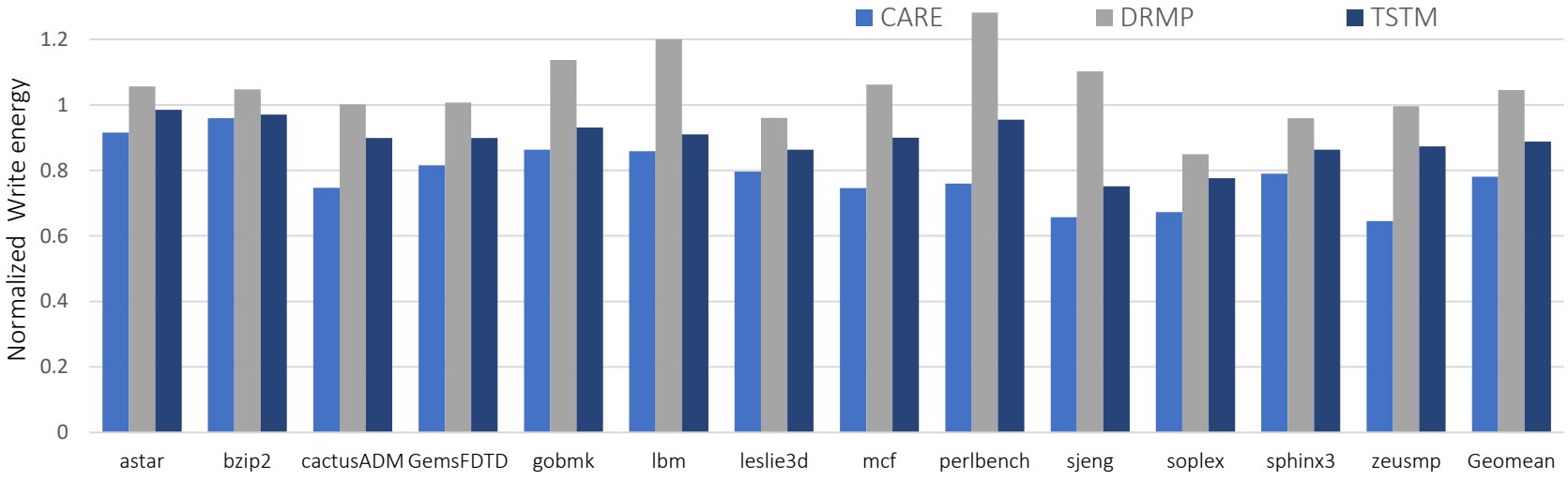


Figure 3-3. Write energy evaluation normalized to LRU+noflip

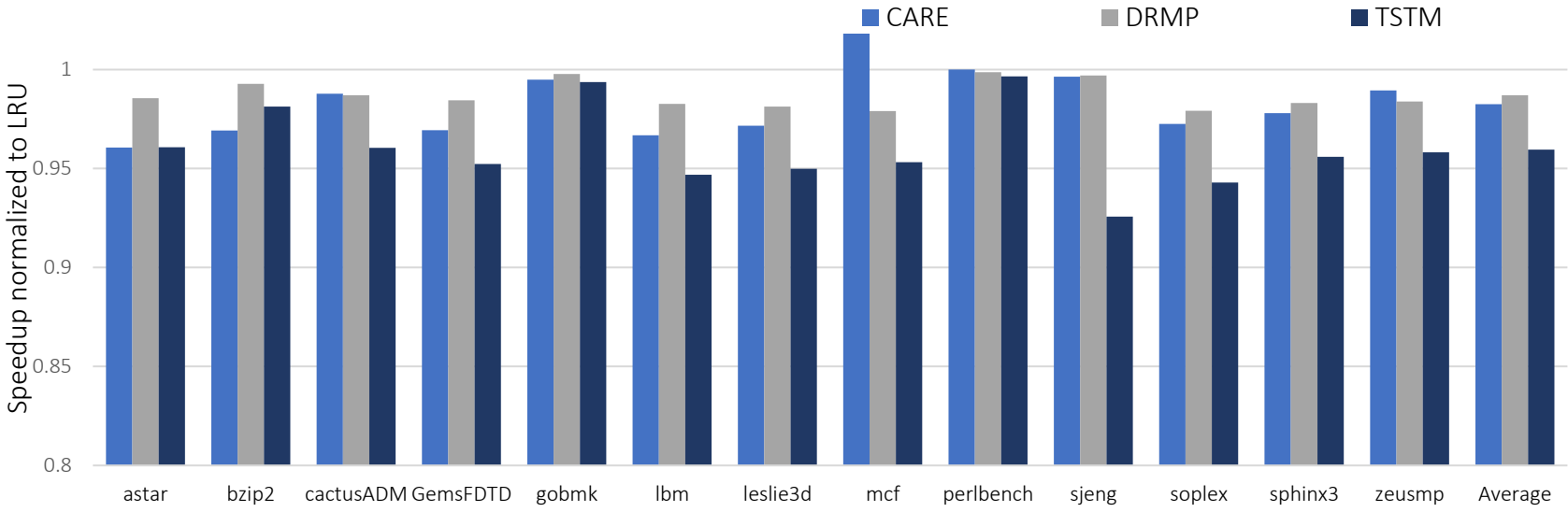


Figure 3-4. IPC results normalized to LRU-noflip

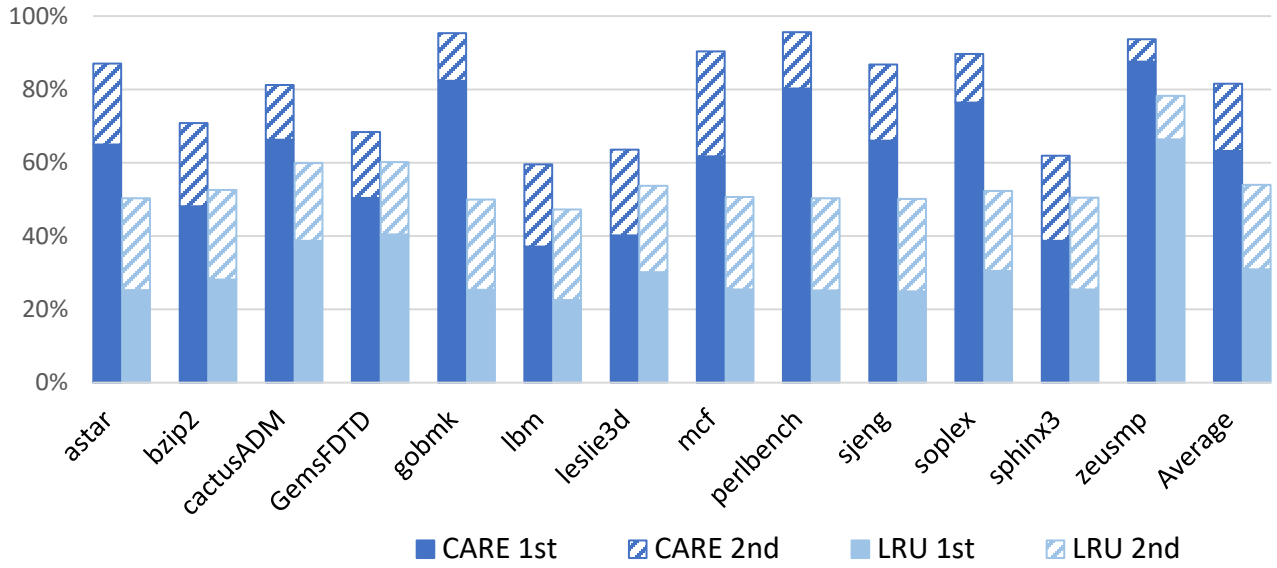


Figure 3-5. Selection efficiency comparison between CARE and LRU

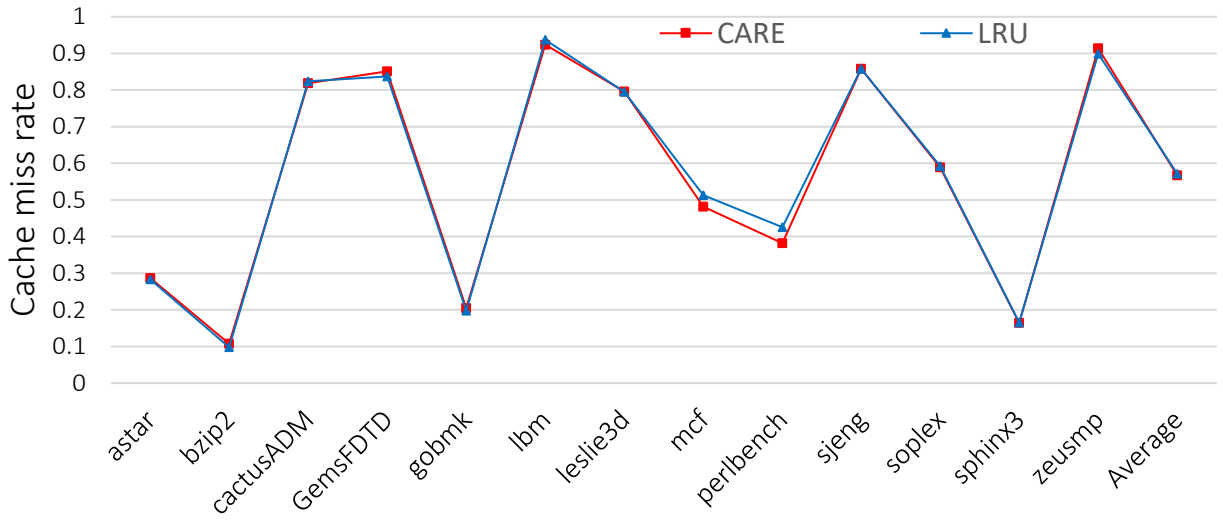


Figure 3-6. Miss rate comparison between CARE and LRU

Among 4 P-LRU candidates, CARE can accurately identify the cache line with the best or second-best write energy in 81.6% cases. On the other hand, without consideration for energy, on average, LRU selects the most energy-efficient block in 30.9% cases merely by chance and around 23.0% for second-best ones except for zeusmp. It may also be noted that while CARE outperforms LRU in all workloads, some workloads like lbm, leslie3d and sphinx3 show less efficiency

compared to other workloads. This indicates these workloads have less UU pair concentration to effectively select replacement blocks using UU alone and so we fall back on LRU often. These workloads may perform better with a more sophisticated selection mechanism considering encoding transition pairs other than UU which will be our future work.

3.5.2 Miss Rate and IPC Performance Comparison

Figure 3-6 shows the miss rate of CARE and LRU. Miss rate is widely accepted as the metric for measuring L3 cache performance. The rise in L3 misses and replacements will cost more energy and slow down the execution with longer the average memory access time. Result shows that selective pseudo-LRU policy is capable of keeping the miss rate for CARE almost the same as P-LRU for most workloads except for zeusmp and GemsFDTD, with marginally higher miss rate. For a few workloads like mcf and perlbench, CARE outperforms LRU with less cache misses. These differences can be attributed to the fact that some workloads with higher miss rates and longer reuse distances of blocks can be more sensitive to selection changes whereas most other workloads can absorb the difference and at times can perform better for a close approximation like selective P-LRU for LLC.

Instruction Per Cycle (IPC) is used as the metric to evaluate overall system performance. As shown in Figure 3-4, the proposed CARE performance is comparable to baseline LRU. There is 1.7% on an average and up to 3.9% degradation in IPC due to the extra delay in encoding operation during cache write. Our target cache is the last level cache, which is insensitive to the off critical-path delay during cache write operation, and the extra 1 cycle read latency has limited effect on system performance. The increase on cache miss rate also influences the performance for some workloads.

For most workloads, CARE replacement policy keeps similar miss rate as LRU, except for GemsFDTD and astar, which show 3.3% and 3.9 % IPC loss respectively due to extra cache

miss penalties from CARE. Mcf, on the other hand, shows 1.8% IPC gain from a 3.2 % miss rate reduction from LRU. With least read and write delay, DRMP is the best out of three designs in terms of IPC, only 1.2% worse than baseline on average. TSTM also sees 4.1% IPC degradation because of extra read and write delays. It is important to emphasize that energy consumption for extra cache misses have been taken into consideration in our simulation. In summary, CARE shows significant energy reduction with minor adverse impact on the IPC.

3.6 Summary

We presented in this chapter a new content-aware, energy-efficient cache replacement (CARE) method for future MLC STT-MRAM caches. Performance evaluation showed that CARE could lower the write energy by 21.9% in comparison with regular LRU-based cache replacement without any enhancement for write energy reduction. Such a significant reduction came from both intelligent selections of replacement blocks as well as an energy-conscious remapping with minimum impact (1.7%) on the overall IPC. We first encoded the chunks inside a block using a few bits. Based on the pairwise match of the encoding bits from the existing resistance state to the new blocks, one of the candidate blocks close to P-LRU positions with the most UU paired chunks is selected. This content encoding and replacement selection are new and effective to reduce replacement write energy consumption. We also observed that remapping of new block content with considerations of old block content was a powerful mechanism to further reduce the write energy.

CHAPTER 4
DATA LOCALITY EXPLOITATION IN CACHE COMPRESSION

4.1 Cache Compression in Need

New applications and emerging technology are the driving forces to advance microarchitecture innovations and designs. Emerging big data, cloud computing, deep learning applications present profound storage impact on future processors. Caches continue playing a critical role in hiding long memory latency and alleviating high memory bandwidth requirement.. In this chapter, we introduce a new cache compression scheme which can effectively enlarge the last-level cache capacity for improving cache performance without changing conventional cache design and access mechanism.

It is well-known that memory references exhibit temporal and spatial address locality. After a data element is referenced, this data item (temporal) and its nearby data (spatial) tend to be referenced in the near future. On the other hand, memory references also exhibit data locality. A repeated instruction has a good chance to produce data with constant or regular (stride) values in nearby memory locations [47]. Existing dictionary-based cache compression solutions [49][50] capture the repeated data patterns in a dictionary and use pointers to link to the data patterns for compressing a cache block.

Compressing multiple data blocks into a single block with conventional block size (e.g. 64 bytes) maintains the mapping between cache tag and data arrays and helps preserving conventional cache access mechanism. In a recent cache compression proposal, dictionary-sharing (DISH) compresses up to four neighboring blocks into a single 64-byte block with sharing of the frequent data patterns. However, we discover that confining the compressed block at 64-byte granularity limits the number of frequent data patterns and lowers the compression ratios. We also discover that data locality exists across neighboring blocks in several applications

in such a way that the content of one block varies slightly to the content of its neighboring blocks. By using the entire content of a block as dictionary records more repeated data patterns and improves the compression ratios.

4.2 Motivations and Block Content Similarity

Cache compression has been researched in recent years [48] [50] [51] [52] [53] [54] [55] [56] [57]. Besides a high compression ratio, the state-of-art approach made a critical accomplishment; accessing a compressed cache the same way as accessing an uncompressed cache to confine the complexity and overhead for identifying cache hit/miss and decompression during cache access. To accommodate normal cache access through tag and data arrays, compression is performed at fixed cache block granularity. Several key features can be observed:

- Maintaining one-to-one mapping of cache tag and data, thus each tag in the tag array is associated with a corresponding data block in the data array with a conventional block size (e.g. 64 bytes), either compressed or uncompressed.
- To save tag space and exploit content-similarity of neighboring blocks, sector-cache design [58][59] has been adopted. Four blocks in a sector (also referred as super-block) are allocated in the same cache set as the target for compression.
- Frequent pattern based compression is adopted which records several common 4-byte data patterns as the dictionary and uses remaining space to record up to four pointer groups to the dictionary, each representing a compressed 64-byte data block.

One recent approach, DISH [48], demonstrated the above and resulted decent compressing ratios. DISH records up to 8 4-byte data in the dictionary and uses the remaining space to record pointer groups for up to 4 64-byte blocks. Each block requires 16 3-bit pointers

to link to the 8 4-byte data in the dictionary. Sector-based tag array records the sector tags with individual block ids to determine cache hit/miss. Each sector has 4 consecutive 64-byte blocks and may occupy multiple entries in the tag array. Each tag associates with one or more blocks in an uncompressed or compressed form in the data array.

Based on cache block contents from SPECCPU 2006 benchmarks, we observe two fundamental shortcomings in the DISH compression approach as described below. We also present innovative solutions to overcome such shortcomings.

- First, we noticed that many benchmark programs require a large number of common data patterns (referred as keys) in the dictionary, the size of which may be beyond the capacity available in a 64-byte block for compressing blocks in a sector. We also observe that these workloads reveal high inter-block content similarity such that the contents of neighboring blocks differ only by a small number of keys. In Figure 4-1, we pick four snapshot examples from libquantum, mcf, perlbench, and soplex to demonstrate the existence of such inter-block content similarity among 4 sector blocks. For example, in libquantum, each block has 8 distinct 4-byte keys (marked bold in block 92214). However, between any two blocks, the difference is reduced to 4 4-byte keys (marked bold). Any two blocks cannot be compressed into a 64-byte block due to the need for 12 keys. This behavior motivates us to pick one uncompressed block as the master block and compress other content-similar blocks in a separate companion block. The companion block records the different keys along with a few pointer groups to link to 16 4-byte data in the master block. We call this approach as a dual-block compression, which extends the number of keys in two blocks and use them as the dictionary.

4-byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
libquantum																
(92214)	**3a3504e8**	**0**	**01004008**	**001b2278**	3a3504e8	0	**01000000**	**001b2280**	3a3504e8	0	01004008	**001b2288**	3a3504e8	0	01000000	**001b2290**
(92215)	3a3504e8	0	01004008	**001b2298**	3a3504e8	0	01000000	**001b22a0**	3a3504e8	0	01004008	**001b22a8**	3a3504e8	0	01000000	**001b22b0**
(92216)	3a3504e8	0	01004008	**001b22b8**	3a3504e8	0	01000000	**001b22c0**	3a3504e8	0	01004008	**001b22c8**	3a3504e8	0	01000000	**001b22d0**
(92217)	3a3504e8	0	01004008	**001b22d8**	3a3504e8	0	01000000	**001b22e0**	3a3504e8	0	01004008	**001b22e8**	3a3504e8	0	01000000	**001b22f0**
mcf																
(8d397c)	**0**	0	**1e**	0	1e	0	**003ac708**	0	**00155a70**	0	**1**	0	**23313410**	0	**2342fed0**	0
(8d397d)	0	0	1e	0	1e	0	**003ac638**	0	00155a70	0	1	0	**233c3a10**	0	**2342ff10**	0
(8d397e)	0	0	1e	0	1e	0	**003ac3c8**	0	00155a70	0	1	0	**233ddbd0**	0	**2342ff50**	0
(8d397f)	0	0	1e	0	1e	0	**003ac228**	0	00155a70	0	1	0	**233ee450**	0	**2342ff90**	0
Perlbench																
(60604)	**0**	0	**450ba8b6**	0	**2030fbe8**	**1**	**6**	0	0	0	**aefcbc10**	0	**20f182e8**	1	**c**	0
(60605)	0	0	**ef94b794**	0	**20416008**	1	**4**	0	0	0	450ba8b6	0	2030fbe8	1	6	0
(60606)	0	0	**aefcbc10**	0	**20c49c78**	1	**12**	0	0	0	**21534396**	0	20416008	1	4	0
(60607)	0	0	**450ba8b6**	0	2030fbe8	1	6	0	0	0	aefcbc10	0	210a2fa8	1	12	0
Soplex																
(bf80)	**0**	**bff00000**	**00001bb5**	**00000b00**	0	**40000000**	**2**	0	0	**3ff00000**	**0000375e**	**00000700**	0	bff00000	**0000372c**	00000b00
(bf81)	0	40000000	2	0	0	3ff00000	**000052d5**	00000700	0	bff00000	**000052a3**	00000b00	0	40000000	2	0
(bf82)	0	3ff00000	**00006e4c**	00000700	0	bff00000	**00006e1a**	00000b00	0	40000000	2	0	0	3ff00000	**000089c3**	00000700
(bf83)	0	bff00000	**00008991**	00000b00	0	40000000	2	0	0	3ff00000	**0000a53a**	00000700	0	bff00000	**0000a508**	00000b00

Figure 4-1. 4-byte hex contents in 4 sector blocks

- Second, we observe that spatial locality among blocks in a sector is weak in several benchmark programs. As a result, not all blocks in a sector are presented in cache during the life time of a sector, limiting the blocks that can be compressed together. It is challenging to increase the sector size and allocate all sector blocks in the same cache set since allocating a larger sector in the same set causes threshing among sectors and degrades cache performance. To enlarge the scope of compressible blocks, we present an innovative idea which uses a pair of adjacent cache sets as buddy sets. The least-significant index bits of the buddy sets are complementary to each other. In the dual-block compression described previously, the master block and its companion block can come from two adjacent 4-block sectors located in a pair of buddy sets. Furthermore, each master block can serve more than one companion block located in the buddy sets. Compressing blocks across two adjacent sectors doubles the compression candidates without alternating cache placement of sector blocks.

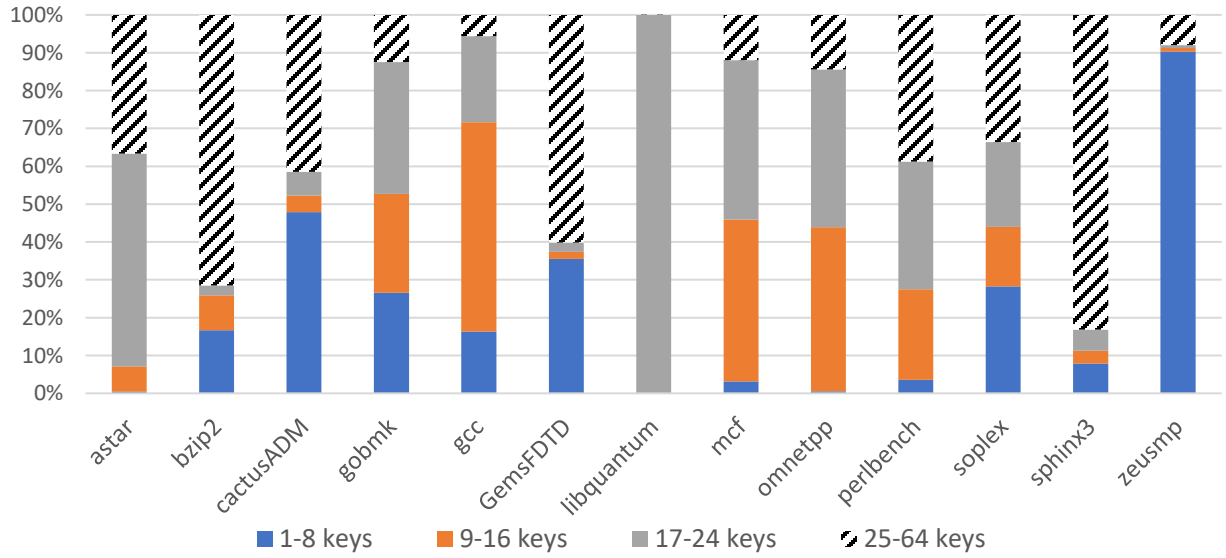


Figure 4-2. Number of keys covering blocks in a sector

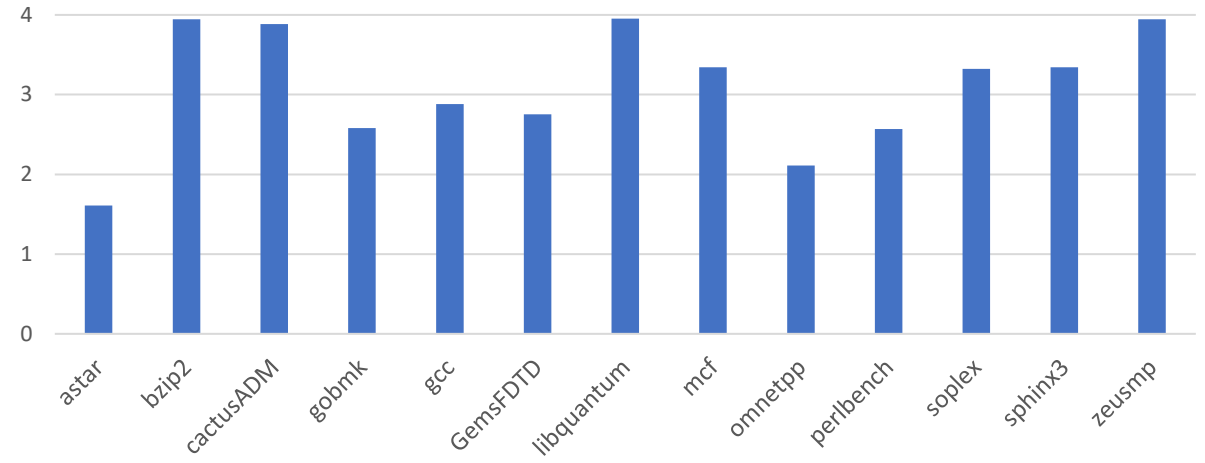


Figure 4-3. Average number of cached sector blocks

We explore block content similarity by measuring the number of 4-byte shared keys required to cover the contents of blocks in a sector. This experiment is performed using the GEM5 whole system simulation tool running SPEC2006 benchmarks. Detailed description of the simulation environment will be given in Section 4-5. For compression of 4 blocks in a sector, a maximum of 64 keys are needed if none of two 4-byte values are the same in all 4 blocks. On the other hand, it only needs a single key if all 4-byte values are identical, e.g., when the values of all 4 blocks are equal to zeros. For a high compression ratio, the number distinct keys must be limited. In DISH, for example, up to 8 keys can be recorded in the dictionary. When the number of keys is beyond 8, it cannot compress the cached blocks in a sector into a single 64-byte block.

In Figure 4-2, we separate the number of keys into 4 groups: 1-8, 9-16, 17-24, and >24. We can observe that although the percentages of 1-8 blocks are low in several benchmarks, the percentages of 9-16 and 17-24 keys are high in these benchmarks. Therefore, it is essential to increase the number of sharing keys for improving the compression ratio providing justification to potentially using the dual-block compression to provide more keys.

In the second experiment, we measure the average number of cached blocks in a sector. As observed in Figure 4-3, GemsFDTD, omnetpp, perlbench show poor spatial locality with an

average of 1.6-2.9 blocks presented in cache during the life time of a sector, which limits the candidate blocks for compression. In this chapter, we expand candidate blocks in adjacent sectors located in a pair of buddy sets to increase the number of blocks also taking advantage of the proposed dual-block compression.

4.3 Dual-Block Compression and Buddy Sets

In this section, we present the design of the proposed dual-block compression. We also include neighboring sectors in the buddy set in searching for compressible blocks. To simplify our presentation without loss of generality, we introduce our proposal on an 8MB, 16-way sector LLC with 64-byte blocks. A conventional sector cache design is used with 4 blocks per sector. The physical address has 48 bits. The addressing scheme and the tag array layout are given in Figure 4-4. Each tag entry consists of a 3-bit function code (FC), 29-bit sector tag, 4 valid bits for presences of sector blocks, and 4 coherence states (CS) for the 4 blocks. The tag entry for a companion block records individual block id and the coherence state for three compressed blocks. The definition of the FC and the content of a companion block will be given later. The proposed compression method consists of two new components.

- Dual-block compression (Dual-Block): Based on the observations in Figure 4-2, we use two blocks to compress three or four blocks when these blocks cannot be compressed in a single 64-block. The master block has 64-byte uncompressed data and the companion block can compress up to 3 blocks using the keys recorded in both the master and the companion blocks.

- Buddy set compression (Buddy-Set): Due to lack of spatial locality (Figure 4-3), not all sector blocks are referenced during the lifetime of a sector, we extend the compressible blocks from the sector in the same set to its neighboring sector located

in the buddy set to enlarge the number of compressible candidates. With additional blocks in the buddy set, it also permits sharing a master block with multiple companion blocks.

4.3.1 Dual-Block Compression

When a cache miss occurs and a new block is brought into cache from main memory, the first compression attempt is to compress and compact the new block with existing ones from same sector with blocks in this set. This step is similar to that used in DISH. If this step is not successful fails, the next step is to explore dual-block compression. When there are three uncompressed sector blocks, dual-block compression is tested to select a master block and two other blocks which are compressible in a separate companion block. Since the master block is uncompressed, we select the one which contains the most keys among the three blocks. In case that compressed dual-blocks already exist, the missed block is attempted to be compressed into the companion block. With 4-block sectors, a 3-block companion can achieve the compression ratio of 2. With 2-block companion, the ratio is 1.5.

The detailed bit layout of a companion block is given in Figure 4-4(d), in which we can record 8 different keys and used them as the extra dictionary for the companion blocks. Up to 3 pointers groups are used to compress 3 blocks. It is sufficient to cover the remaining 3 blocks in a sector. Since there are a total of 24 keys (16 in the master block and 8 in the companion block), 16 5-bit pointers are necessary for each pointer group to record the content of a compressed block. The size of a companion block is 63 3/8 bytes. Note that for the companion block used in the dual-block compression, three block IDs and their coherence states are encoded in the tag array (Figure 4-4(c)), and we don’t need to record them along with the pointer groups in the companion block.

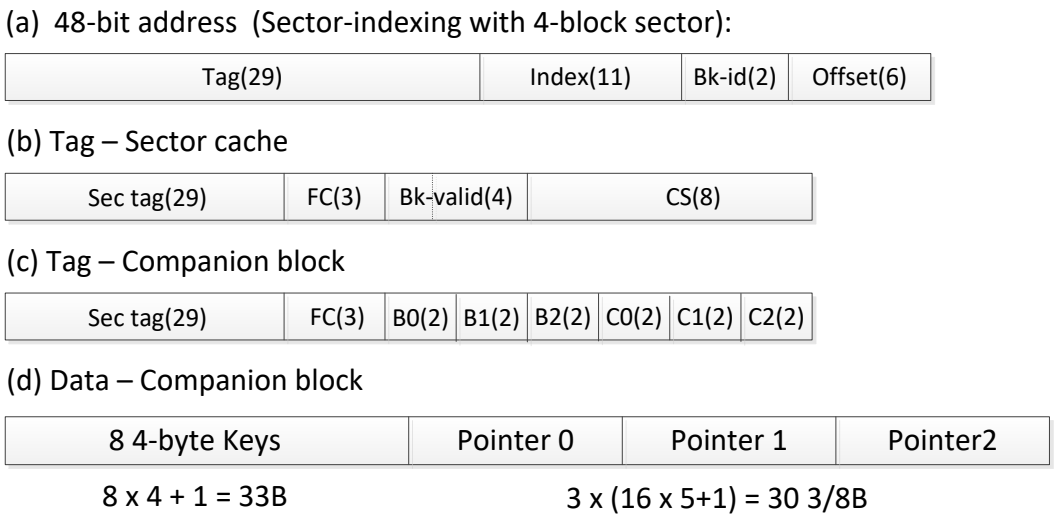


Figure 4-4. Address separations (a) 48-bit address; (b) tag for 4-block sector cache; (c) tag for companion block; (d) data for companion block

When a master block is replaced from the cache, its companion block must also be invalidated at the same time. Therefore, when a companion block is referenced, the master block is also promoted to the second to MRU position by proper modified LRU replacement policy.

4.3.2 Buddy Sets

Buddy sets are two adjacent cache sets, which use the same index bits, except for having complementary values in the least-significant bit. For simplicity, consider a 4-block sector cache with only 6 index bits in 10-bit block address (Figure 4-4(a)). Two block addresses: *aa, 000000, xx* and *aa, 000001, yy* represent block xx of sector aa in set *000000* and block *yy* of sector aa in set *000001* respectively. These two sets *000000* and *000001* are formed the buddy sets. Sectors aa in both sets are two consecutive 4-block sectors located in a pair of buddy sets, referred as *buddy sectors*. Note that the buddy sectors have identical tags located in the buddy sets.

To enlarge the compressible blocks beyond blocks in a 4-block sector, we can select the candidate blocks from both buddy sectors. When a missed block is uncompressible in the original cache set, a search in the buddy set is carried out if another uncompressible sector block

already exists in the original set. All blocks in the buddy sectors are tested for dual-block compression. If successful, the master block is selected with the most keys among all blocks in the buddy sectors. The remaining blocks are compressed into a companion block. The master block can be located in one set and two or more blocks from the buddy sectors can be compressed into a companion block placed in the other set.

Note that each master can serve one or more companions. When searching for dual-block compression in the buddy set, an existing master can be found for serving the new companion. Therefore, in an ideal case, a master block can serve two 3-block companions, one in each set to achieve the compression of 7/3=2.33. Similarly, a master block can potentially serve one 3-block companion in the same set and two 2-block companions in the buddy set to achieve the compression ratio of 8/4=2.

4.3.3 Putting It All Together

With different compress options, a 3-bit function code (FC) is associated with each tag in the sector tag array. The definition of FC is given in Table 4-1. Note that five FCs are used to define a master and a companion as well as their locations. The master and its companion can be located either in the same set or across the buddy sets.

During cache access, in case of a tag match, the corresponding data block is fetched from the data array. For an uncompressed block with FC = 000, 010, 011, or 100, the data block is accessed normally. For single compressed block with FC = 001, the data block is fetched and decompressed using the correct pointer group and the keys in the dictionary. In case that the hit is to a companion block, the master block must be fetched. The master block has the same tag as the companion block and can be located either in the same set with FC=101, or in the buddy set with FC=110. The data block is then decompressed using the keys in both blocks and the correct pointers in the companion block.

Table 4-1. Function code definition

Function Code (FC)	Description
000	Uncompressed
001	64-byte compression
010	Master, companion in same set
011	Master, companion in buddy set
100	Master, companions in both sets
101	Companion, master in same set
110	Companion, master in buddy set

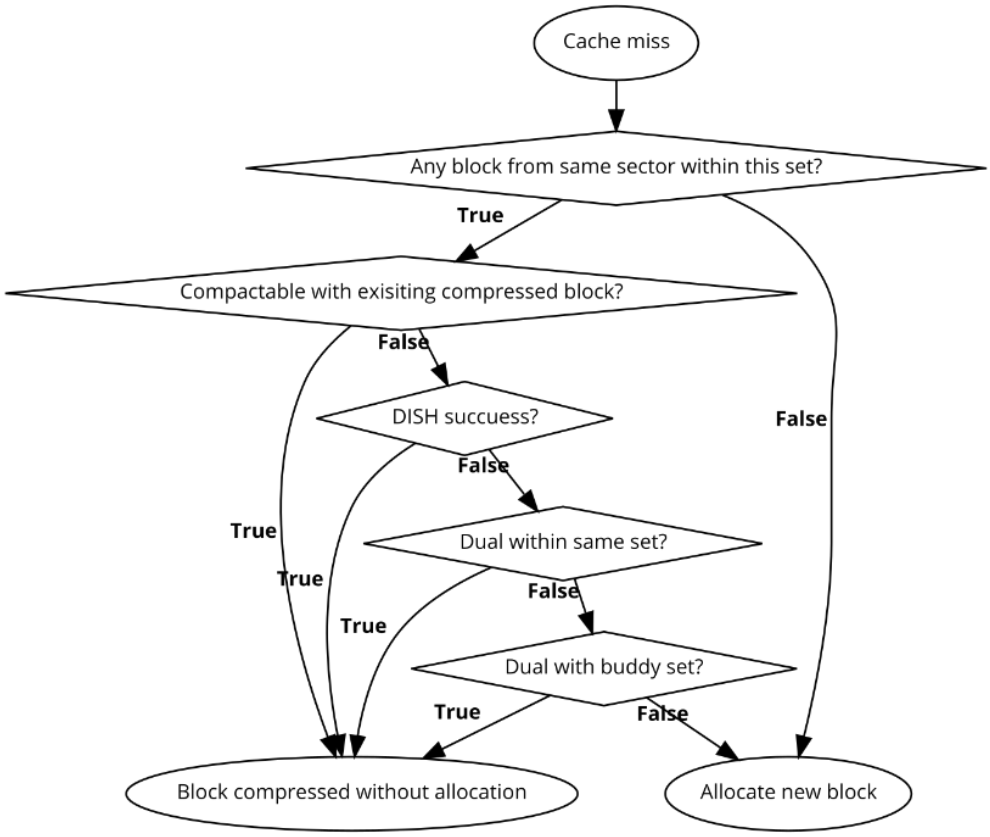


Figure 4-5. The flow chart of compression decision

Data writebacks involve updates to compressed blocks. In case of a writeback to compacted block in companion block, it could become non-compactable and when it happens we invalidate the block in companion and allocate a new block for it. If the writeback happens to a

master block, we need check if the companion block could be compressed with this master and we will try to compress them with the same compression process.

When a cache miss occurs, a new block is moved into cache, followed by attempting to compress the block with existing sector blocks in the set. In case that the block cannot be compressed with DISH or dual block, a search for dual-block compression to buddy set happens. The compression decision is shown in Figure 4-5 as a flow chart.

4.3.4 A Compression Example

In this section, we use a snapshot of 4-block sector cache contents taken from soplex to illustrate how dual-block compression works.

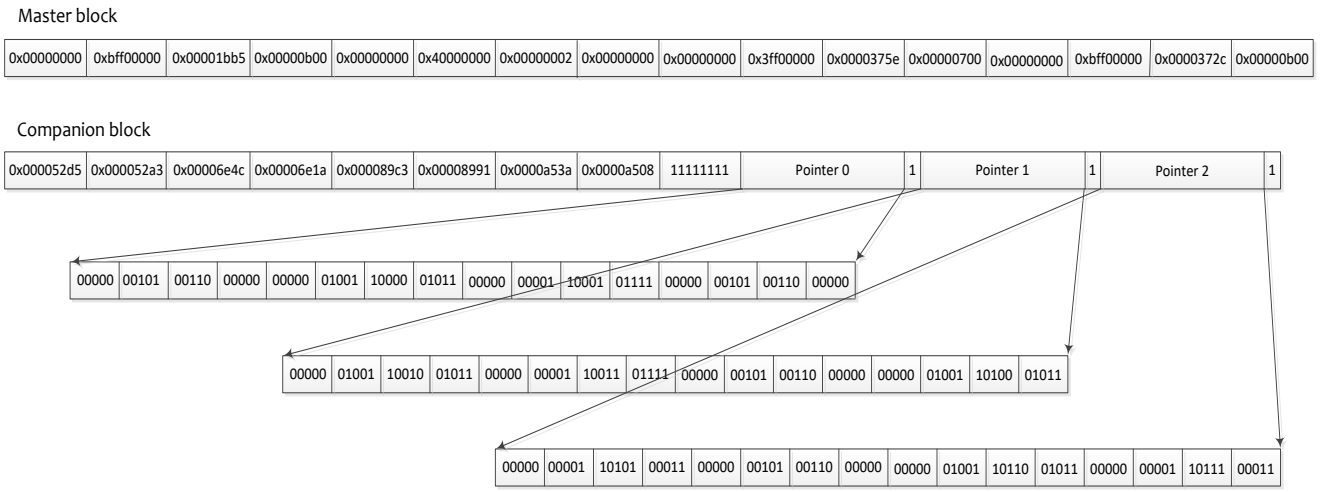


Figure 4-6. Dual-block compression for 4-block sector in *soplex*

As shown in Figure 4-1, each block in the 4-block sector has 10, 8, 8, and 9 distinct keys. None of any pair of blocks can be compressed into a 64-byte block using DISH-like compression scheme. However, if we select the first block (bf80) which has the highest number of keys as the master block, the other 3 blocks can be compressed in a companion block since only 2, 3, and 3 more keys required for the 3 remaining blocks respectively. Figure 4-6 illustrates the contents of the uncompressed master block and the compressed companion block. Each pointer group has 16

5-bit pointers to either the 16 4-byte dictionary in the master block, or the 8 4-byte dictionary in the companion block.

4.3.5 Overhead Analysis

This section provides the area and latency overhead analysis that comes with proposed dual-block compression. With one-to-one tag and data mapping, only 3-bit function code is added and there is no other storage overhead for storing the compression related meta-data. We estimate the area and latency overhead by using CACTI 6.5 [16]. Regarding a 32-nm technology, for a 4MB cache that occupies 34mm2 (both tag and data array), the area used by the compressor and de-compressor is less than 0.7mm2, which is counted as only 2% extra space in terms of space overhead.

According to CACTI estimation, a read or write to 4MB SRAM cache with 32-nm technology normally takes 40 cycles. Additionally, the main source of latency overhead is from accessing two blocks during a hit to a compacted companion block. If both the master and the companion blocks are from the same set, the decompression process is fast due to the fact that the normal tag array search can locate both the companion block and its master. Fetching the companion and master blocks can be overlapped with multiple banks in the data array. We estimate it takes 1 cycle and 25 cycles for decompression and compression, respectively. In a worse case when the master is from the buddy set, an immediate accessing to the master block in the buddy set is triggered. It takes 2 circles decompression latency as a result and the compression takes 27 cycles also due to buddy set operations.

Notice that the cache compression process is off the cache access critical path. The compression happens only on a miss to last level cache. The data fetched from DRAM is sent back to L2 cache immediately to satisfy CPU data requests before any compression operations. Obviously, there is no extra latency when accessing the uncompressed master block.

4.4 Evaluation Methodology

The evaluation focuses on the energy consumption and system performance of STT-MRAM. Gem5[46], a cycle accurate system simulator, is employed to evaluate Dual-block compression including both home set and buddy set (Buddy), Dual-block compression considering home set only and DISH, while a SRAM cache without cache compression is the baseline. The cache model of Gem5 is modified to support the compression and decompression process while the extra latency is also simulated. The processor is configured as a processor with a three-level cache hierarchy. The configurations of the CPU and memory are summarized in Table 4-2.

The cache compression can help keep more blocks in cache, which is greatly useful for application with high MPKI and sensitive to cache size. From SPEC CPU2006 benchmark, we test and select workloads with relatively high MPKI, along with low-MPKI gobmk and perlbench for comparison purpose, to make a set of 13 benchmarks in the experiment.

To locate the most representative phase of whole program execution, we take advantage of SimPoint [17] to pick a checkpoint for each workload. We collect performance statistics for 500M instructions after a warm-up of 1B instructions from the checkpoints. These realistic and better representative checkpoints are usually hundreds, even thousands of billions of instructions away from the program starting points. In the simulation methodology described in DISH paper, they selected a different phase of the programs to do the simulation. Instead of using SimPoint, they fast-forward 20 billion of instructions for each application. As a result, their published compression performance is different from what we obtain by repeating the DISH method using the checkpoints generated by Simpoint.

We also notice that L2 prefetcher is a common technique used in modern cache design. Stream-based prefetcher [60] can bring in more neighboring blocks and may help the

compression ratio. However, as shown in Figure 4-3, for some workloads such as astar and omnetpp with weak spatial locality, the prefetcher may fetch neighboring blocks that will not be requested by the processor. As a result, compression ratios could be pumped up by the prefetcher and distorted the results. Therefore, we decided to exclude the prefetcher from our simulation model for evaluating DISH as well as the proposed dual-block compression.

Table 4-2. Parameters of the simulated system

Component	Parameters
Processor	3GHz, out-of-order, 8-issue
L1 D/I	Private, 32 KB, 4-way
L2	Private, 256 KB, 8-waye
MSHRS	Shared, 4 MB, 16-way, Non-inclusive
Cache line size	16, 16, 32 MSHRs at L1, L2, L3
DRAM	64B in L1, L2 and L3
	8GB, DDR4-2400, 64bit I/O 8 banks, 2KB row buffer tCL-tRCD-tRP-tWR: 13-13-13-14

4.5 Performance Results

In this section, we evaluate the effectiveness of dual-block compression with / without the buddy sets. We compare these approaches with the state-of-the-art DISH compression method in terms of compression ratio, misses per kilo instruction (MPKI), and overall performance improvement.

4.5.1 Compression Ratio

We first measure the effectiveness of the compression methods by the compression ratio, which is defined as the effective size of the cache based on the cache size without compression. We calculate the compression ratio at the set level granularity and take the average across all the

cache sets [48]. Figure 4-7 shows the compression ratios achieved by buddy-set compression (labeled *"Buddy-set"*), dual-block compression considering home set only (labeled "*Dual-block*") and DISH. The average compression ratios are 1.60, 1.43, and 1.29 respectively for Buddy-set, Dual-block and DISH. Comparing with DISH, Dual-block, and Buddy-set have about 11%, and 24% average cache size improvement.

The compression ratio varies quite significantly among workloads using three compression methods. Astar, gcc, libquantum, mcf, omnetpp, perlbench, soplex have substantial improvement from Buddy-set and for these workloads, the effective cache size improvements are ranging 25-86% over that of DISH. Meanwhile, we notice that some workloads have poor compression ratios for DISH due to lack of sufficient space for 9 or more keys to be recorded in the directory in a single block, such as astar, libquantum, mcf, omnetpp, and perlbench. By using Dual-block to provide more keys, the compression ratios are greatly improved.

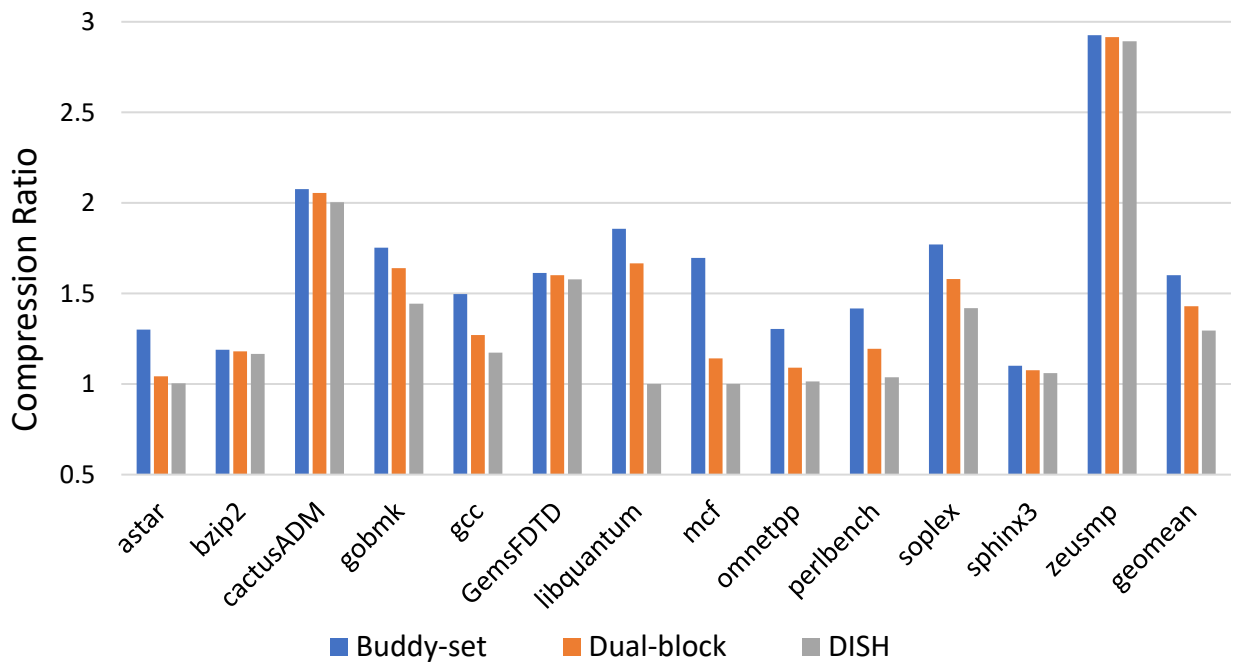


Figure 4-7. Compression ratios for different compression schemes

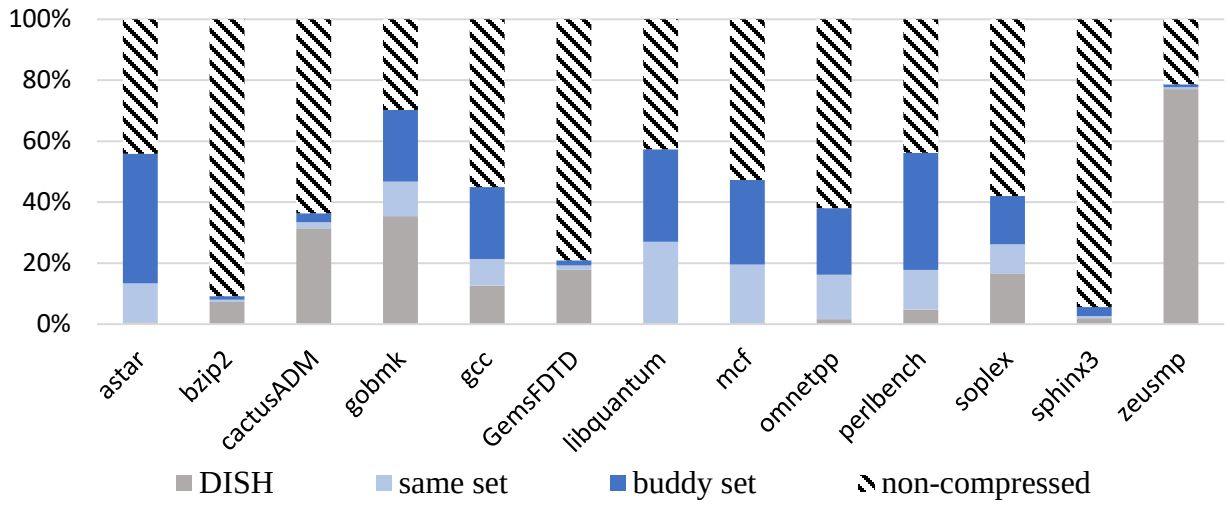


Figure 4-8. Distribution of different techniques used in cache compression

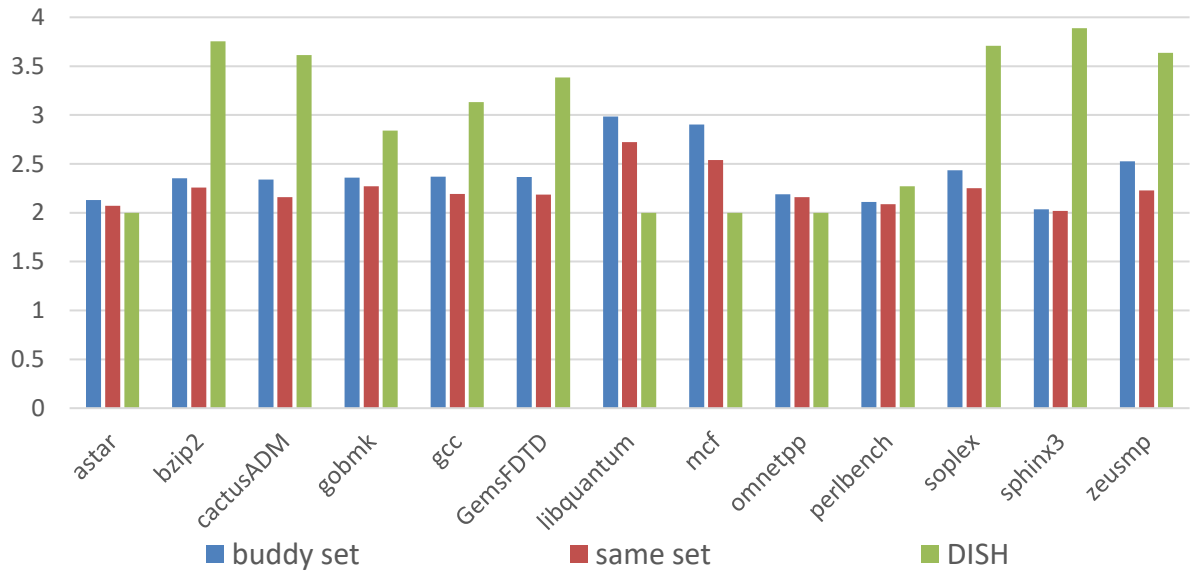


Figure 4-9. Average number of blocks compacted in a companion / DISH block

The compression ratio improvement comes from additional compressible blocks using Dual-block in the same set or in the buddy set. Figure 4-8 shows the distribution of the applied compression techniques for different workloads when Buddy-set is used. As expected, workloads with significant amount of dual-block compressions show higher compression ratios. Dual-block has little impact on workloads which is dominated by single 64-byte compression and/or uncompressed blocks, such as bzip2, GemsFDTD and zeusmp.

We further collect the average number of blocks compressed into a single 64-block or compressed into a companion block. As shown in Figure 4-9, the average number of blocks is ranging from 2 to 3.9. A few workloads such as astar, libquantum, mcf, and omnetpp display inefficient single 64-byte block compression with an average close to the minimum two out of possible four blocks. For the results of companion blocks, libquantum and mcf have an average of 3 and 2.9 respectively, which are close to the theoretical maximum three compressed blocks using Buddy-set and show higher compression ratio in Figure 6. Among all other workloads, the average compacted blocks range from 2.2 to 2.5 per companion block.

4.5.2 Speedup

Next, we measure performance speedups by effectively increasing the LLC size with different cache compression methods. Figure 4-10 shows that the average speedups of 8.9%, 5.6% and 3.6% can be achieved for Buddy-set, Dual-block, and DISH respectively, normalized to an uncompressed baseline 4MB cache. Again, the speedup varies widely from different workloads and different compression methods. Astar and soplex display the highest speedup about 34-39% for Buddy-set compression, zeusmp about 18%, while most others only experience 1-9% improvement. Besides, DISH is especially worse since many workloads have insignificant improvement on compression ratios. It even shows negative improvement for mcf, due to additional latency overheads from compression and decompression operations.

The speedup comes mainly from the MPKI improvement over the baseline design as shown in Table 4-3. Astar, gcc, mcf, omnetpp, perlbench, soplex, spinx3 show significant MPKI reduction for Buddy-set over the baseline, thus good speedups have exhibited in Figure 4-10. Among them, astar is the best one with 67.8% MPKI reduction, which leads to the 39% speedup over baseline with shorter average memory latency. For libquantum, the negligible MPKI reduction and speedup are somewhat unexpected since it demonstrates significant improvement

in compression ratio. It appears that this workload has large footprint and rather insensitive to the cache size. From early studies, even doubling the cache size cannot improve the MPKI for libquantum. Nonetheless, we expect good speedup with 1.86 compression ratio when the cache size reaches to the turning point.

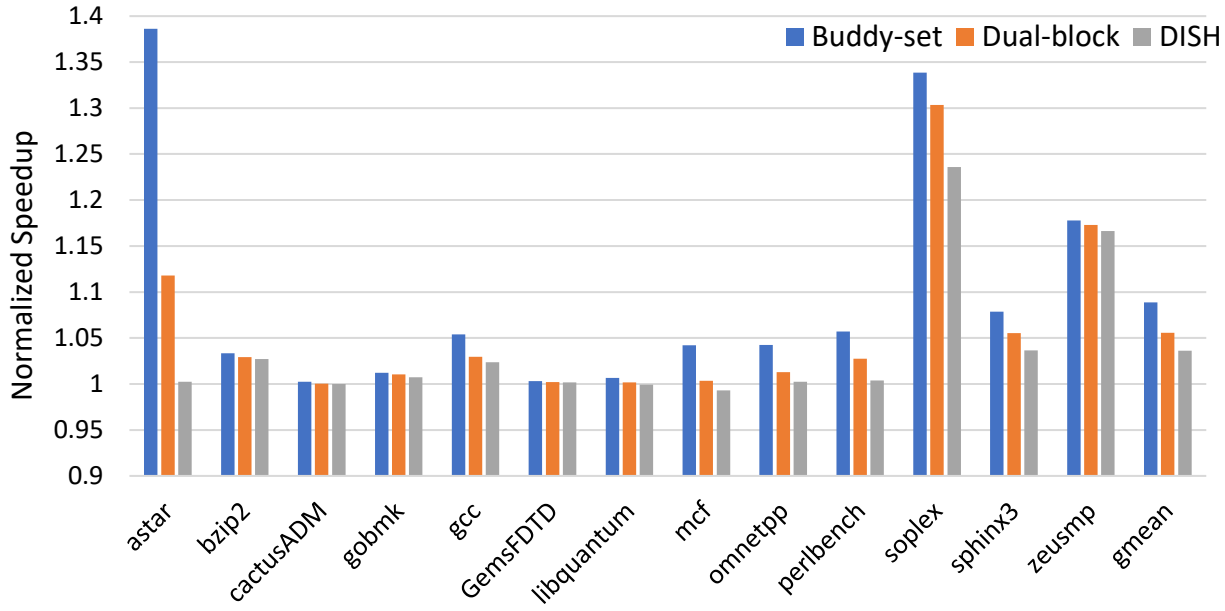


Figure 4-10. Speedup over a 4MB LLC

Table 4-3. LLC misses in term s of MPKI

Workload	Buddy-set	Dual-block	DISH	Baseline
astar	11.29	25.15	34.77	35.06
bzip2	1.44	1.46	1.49	1.82
cactusADM	6.28	6.29	6.29	6.31
gobmk	0.53	0.55	0.57	0.65
gcc	10.07	10.76	10.93	11.52
GemsFDTD	37.51	37.56	37.67	38.38
libquantum	73.59	73.59	73.59	73.59
mcf	170.98	181.74	185.64	185.64
omnetpp	32.45	34.20	35.02	35.16
perlbench	0.68	0.92	1.14	1.18
soplex	20.43	21.56	23.35	40.25
sphinx3	11.48	12.20	12.85	14.49
zeusmp	8.91	8.92	8.95	10.71
Geomean	10.14	11.32	12.05	13.31

4.6 Related Work

Data compression has been researched to enlarge the capacity of main memory and caches. IBM MXT [61] uses real-time main memory compression to effectively double the main memory capacity. In [62], an efficient memory compaction scheme is reported. It handles special zero value compaction as well as uses a TLB-like table to avoid indirect memory mappings. X-Match [63] is a dictionary-based compression algorithm. It matches 4-byte words using content-addressable memory and allows partial matching with dictionary entries to output variable-size encoded data. C-Pack [50] uses a single shared dictionary of 64 bytes and target for cache compression. It detects and compresses the frequently appearing data words into few bits. It also extracts other kinds of data patterns and compresses cache blocks by detecting and storing the other frequently appearing 4-bytes in a dictionary. Partial matching of 4-byte data in the dictionary is also permitted. Depending on the actual compressing ratio of individual blocks, it can pack multiple compressed blocks into a single uncompressed block frame. CPACK+Z [56] is a variation of C-PACK with additional feature that detects zero blocks.

Prior work has proposed compression algorithms for last-level cache (LLC) with variable block sizes using hardware implementations [53]. It must efficiently compact and retrieve variable-size compressed cache blocks. Instead of recording frequent data patterns, Base-Delta-Immediate [64] uses one base value for a cache block, and replaces the other data values of the block in terms of their respective delta (differences) from the base value. It compresses a cache block by exploiting the data value correlation property with minimum decompression latency.

Decompressed cache organization must provide address tags to map additional (compressed) blocks, support variable-size data allocation, and maintain the mappings between tags and data. Decoupled Compressed Cache (DCC) [56] proposes tracking compressed blocks at super-block (sector) level to reduce tag overhead. DCC uses sector tags, where each tag tracks

up to four consecutive blocks. DCC compresses each 64-byte block into zero to four 16-byte sub-blocks and uses a decoupled tag-data mapping [59] to allow blocks to be stored anywhere in a cache set. To decouple tag-data mapping, it requires extra metadata to hold the backward pointers that identify a block's location.

Skewed Compressed Cache (SCC) [51] stores neighboring compressed blocks in a power-of-two number of 8-byte sub-blocks in a compressed 64-byte block. It only allows the blocks with the same compressibility to be compressed into a 64-byte block. It uses sparse super-block tags and a skewed associative mapping [65] that preserves a one-to-one direct mapping between tags and data. SCC eliminates extra backward pointer of DCC in tag metadata. SCC allows different super-blocks to map blocks with different compressibility to different cache ways. To reduce conflict misses, SCC uses different hash functions to access ways holding different size compressed blocks.

Yet Another Compressed Cache (YACC) [52] is essentially the same as SCC while eliminating the limitations associated with skewing. YACC uses a conventional sector tag array and an unmodified data array with direct tag and data mapping. In addition to simplifying the tag array, YACC allows the use of any replacement policy and leads to more predictable behavior. YACC stores neighboring blocks in one data entry if they have similar compressibility and could fit in a single data block entry.

The latest state-of-the-art compressed cache, Dictionary Sharing (DISH) is similar to SCC with more flexibility to compress multiple sector blocks with different compressibility in a single 64-byte block. Each compressed 64-byte block records up to 8 4-byte frequent data patterns in the dictionary. It uses the remaining space to build 4 pointer groups pointing to the 4-

byte data in the dictionary. As described in Section 2, DISH uses sector tags for saving tag space and maintains one-to-one tag / data mapping for fast cache access.

In this chapter, we identify two essential issues for the DISH compression: lack of space to record sufficient 4-byte data patterns in dictionary, and missing sector blocks in cache for compression. We propose a new data compression mechanism which compresses multiple neighboring blocks into dual blocks to hold more keys and expands the compression candidates across the adjacent buddy sets to increase the scope for compressible blocks.

4.7 Summary

We propose an efficient cache compression method which compresses and compacts neighboring blocks into a single 64-byte cache block. As in other compression approaches, it records multiple 4-byte data (keys) in a dictionary and uses pointers to compress multiple cache blocks in a sector. It is different from other approaches; however, the proposed scheme can use two blocks for compaction when the number of distinct dictionary keys are too big to be compacted into a single block. This dual-block compression uses one uncompressed block as the master and compacts other sector blocks in a separate companion block. The entire master content is used for the dictionary for compressing sector blocks with large number of keys. This approach is based on our observation that strong data locality exists among neighboring blocks in many workloads such that their contents differ only by a small number of keys.

Furthermore, the proposed method enlarges the scope of searching for compressible blocks. The existing approaches limit the compressible targets within a 4-block sector. Studies show that during the life time of a sector, many workloads only touch a subset of the sector, thus constrained the compressible candidates. We propose the idea of buddy sets where two adjacent sectors are allocated. By expanding the compressible candidates across the buddy sets, it helps

the compression ratio. The performance evaluation using SPECCPU workloads demonstrates the proposed compression method can effectively increase the cache capacity by 60%.

CHAPTER 5
CONCLUSION AND FUTURE WORK

New applications and emerging technology are the driving forces to advance microarchitecture innovations and designs. Emerging big data, cloud computing, deep learning applications present profound storage impact on future processors. Caches continue playing a critical role in hiding long memory latency and alleviating high memory bandwidth requirement. As the speed gap between the processor cores and the memory subsystem has been continuously widening in this multi-core era, the need for larger cache becomes more urgent.

To achieve the goal of placing larger last level cache on chip, we provided two different architecture solutions: applying new memory technology STT-MRAM into cache design and exploiting the data locality with innovative cache compression mechanism.

Due to the inherent endurance shortcoming of STT-MRAM, we proposed a new encoded content-aware replacement policy for both SLC and MLC STT-MRAM caches to reduce the total switch bits for cache replacement with minimum impact on the cache performance. Performance evaluation showed that lower write energy consumption and longer endurance as a result of such new replacement policies.

With the observation that data locality exists across neighboring blocks in several applications and the content of one block is similar to that of its neighboring blocks, we presented a compression method using entire content of a block as dictionary records more repeated data patterns and improves the compression ratios. By expanding the neighboring blocks in the buddy set as candidates for compression, the compression becomes more effective.

Future research will further explore the intelligent combination of those successful works on cache design and consider special compression mechanism based on the characteristics of STT-MRAM. The disparity of write latency and energy among soft and hard cells in MLC STT-

MRAM could be exploited for a potential optimization of dictionary entries storage, which are frequently used in block compression.

LIST OF REFERENCES

[1] Z. Sun, “High-Performance and Low-Power Magnetic Material Memory Based Cache Design,” Ph.D. Dissertation, University of Pittsburgh, 2013.

[2] X. Wang, Y. Chen, H. Li, D. Dimitrov, and H. Liu, “Spin torque random access memory down to 22 nm technology,” *IEEE Trans. Magn.*, vol. 44, no. 11 PART 2, pp. 2479–2482, 2008.

[3] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, “A novel architecture of the 3D stacked MRAM L2 Cache for CMPs,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, pp. 239–249, 2009.

[4] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, “Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM),” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 3, pp. 483–493, 2011.

[5] T. Ishigaki, T. Kawahara, R. Takemura, K. Ono, K. Ito, H. Matsuoka, and H. Ohno, “A Multi-Level-Cell Spin-Transfer Torque Memory with Series-Stacked Magnetotunnel Junctions,” pp. 234–235, 2010.

[6] X. Lou, Z. Gao, D. V Dimitrov, and M. X. Tang, “Demonstration of multilevel cell spin transfer switching in MgO magnetic tunnel junctions,” *Appl. Phys. Lett.*, vol. 93, no. 24, p. 242502, Dec. 2008.

[7] Y. Chen, X. Wang, W. Zhu, H. Li, Z. Sun, G. Sun, and Y. Xie, “Access scheme of Multi-Level Cell Spin-Transfer Torque Random Access Memory and its optimization,” in *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, 2010, vol. 1, no. 2, pp. 1109–1112.

[8] Y. Zhang, L. Zhang, W. Wen, G. Sun, and Y. Chen, “Multi-level cell STT-RAM: Is it realistic or just a dream?,” *IEEE/ACM Int. Conf. Comput. Des.*, pp. 526–532, 2012.

[9] J. Park, S. Hur, J. Lee, J. Park, J. Sel, J. Kim, S. Song, J. Lee, J. Lee, S. Son, Y. Kim, M. Park, S. Chai, J. Choi, U. Chung, J. Moon, K. Kim, K. Kim, and B. Ryu, “8Gb MLC (Multi-Level Cell) NAND Flash Memory using 63nm Process Technology,” *IEEE Electron Devices Meet.*, pp. 8–11, 2004.

[10] P. Chi, C. Xu, X. Zhu, and Y. Xie, “Building energy-efficient multi-level cell STT-MRAM based cache through dynamic data-resistance encoding,” *Proc. - Int. Symp. Qual. Electron. Des. ISQED*, pp. 639–644, 2014.

[11] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “Energy reduction for STT-RAM using early write termination,” in *Proceedings of the International Conference on Computer-Aided Design*, 2009, pp. 264–268.

[12] C. Xu, D. Niu, X. Zhu, S. H. Kang, M. Nowak, and Y. Xie, “Device-architecture co-optimization of STT-RAM based memory for low power embedded systems,” in *2011*

IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2011, pp. 463–470.

[13] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai, "Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory," *J. Phys. Condens. Matter*, vol. 19, no. 16, p. 165209, 2007.

[14] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, "The gem5 Simulator," *Comput. Archit. News*, vol. 39, no. 2, p. 1, 2011.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, p. 45, 2002.

[16] X. Dong, C. Xu, N. Jouppi, and Y. Xie, "NVSim: A circuit-level performance, energy, and area model for emerging non-volatile memory," *Emerg. Mem. Technol. Des. Archit. Appl.*, vol. 9781441995, no. 7, pp. 15–50, 2014.

[17] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili, "An Energy Efficient Cache Design Using Spin Torque Transfer (S T T) RAM," *Islped Acm*, pp. 389–394, 2010.

[18] S. P. Park, S. Gupta, N. Mojumder, A. Raghunathan, and K. Roy, "Future cache design using STT MRAMs for improved energy efficiency," *Proc. 49th Annu. Des. Autom. Conf. - DAC '12*, p. 492, 2012.

[19] 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*, 2009, p. 347.

[20] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[21] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram," *Electron Devices Meet. 2005. IEDM Tech. Dig. IEEE Int.*, vol. 0, no. c, pp. 459–462, 2005.

[22] W. Zhao, E. Belhaire, Q. Mistral, C. Chappert, V. Javerliac, B. Dieny, and E. Nicolle, "Macro-model of Spin-Transfer Torque based Magnetic Tunnel Junction device for hybrid Magnetic-CMOS design," *BMAS 2006 - Proc. 2006 IEEE Int. Behav. Model. Simul. Work.*, pp. 40–43, 2007.

[23] S. Mittal, J. S. Vetter, and S. Member, "A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9219, no. c, pp. 1–14, 2015.

[24] P. Chi, S. Li, S. H. Kang, and Y. Xie, "Architecture design with STT-RAM: Opportunities and challenges," *2016 21st Asia South Pacific Des. Autom. Conf.*, pp. 109–114, 2016.

[25] J. Jung, Y. Nakata, M. Yoshimoto, and H. Kawaguchi, “Energy-efficient Spin-Transfer Torque RAM cache exploiting additional all-zero-data flags,” *Proc. - Int. Symp. Qual. Electron. Des. ISQED*, pp. 216–223, 2013.

[26] S. Yazdanshenas, M. R. Pirbasti, M. Fazeli, and A. Patooghy, “Coding last level STT-RAM cache for high endurance and low power,” *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 73–76, 2014.

[27] M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili, “An energy efficient cache design using spin torque transfer (STT) RAM,” *Proc. Int. Symp. Low Power Electron. Des.*, pp. 389–394, 2010.

[28] Z. Wang, D. a. Jimenez, C. Xu, G. Sun, and Y. Xie, “Adaptive placement and migration policy for an STT-RAM-based hybrid cache,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, pp. 13–24, 2014.

[29] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, “Hybrid cache architecture with disparate memory technologies,” *Proc. Int. Symp. Comput. Archit.*, pp. 34–45, 2009.

[30] J. Li, C. J. Xue, and Y. Xu, “STT-RAM based energy-efficiency hybrid cache for CMPs,” *2011 IEEE/IFIP 19th Int. Conf. VLSI Syst. VLSI-SoC 2011*, pp. 31–36, 2011.

[31] J. Hu, C. J. Xue, Q. Zhuge, W. C. Tseng, and E. H. M. Sha, “Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 21, no. 6, pp. 1094–1102, 2013.

[32] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, “Design exploration of hybrid caches with disparate memory technologies,” *ACM Trans. Archit. Code Optim.*, vol. 7, no. 3, pp. 1–34, Dec. 2010.

[33] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, “IBM POWER8 processor core microarchitecture,” *IBM J. Res. Dev.*, vol. 59, no. 1, p. 2:1-2:21, 2015.

[34] T. Kojo, M. Tawada, M. Yanagisawa, and N. Togawa, “A write-reducing and error-correcting code generation method for non-volatile memories,” in *2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2014, pp. 304–307.

[35] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, “Adaptive insertion policies for high performance caching,” *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 381, Jun. 2007.

[36] P. M. Palangappa and K. Mohanram, “Flip-Mirror-Rotate: An Architecture for Bit-write Reduction and Wear Leveling in Non-volatile Memories,” *Proc. 25th Ed. Gt. Lakes Symp. VLSI - GLSVLSI ’15*, pp. 221–224, 2015.

[37] B. Parhami, “Efficient Hamming weight comparators for binary vectors based on accumulative and up/down parallel counters,” *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 56, no. 2, pp. 167–171, 2009.

[38] S. Wairya, R. K. Nagaria, and S. Tiwari, “Performance analysis of high speed hybrid CMOS full adder circuits for low voltage VLSI design,” *VLSI Des.*, vol. 2012, 2012.

[39] F. Bellard, “QEMU , a Fast and Portable Dynamic Translator,” *USENIX Annu. Tech. Conf. Proc. 2005 Conf.*, pp. 41–46, 2005.

[40] Y. Chen, W.-F. Wong, H. (Helen) Li, and C.-K. Koh, “Processor Caches Built Using Multi-Level Spin-Transfer Torque RAM Cells,” in *Low Power Electronics and Design (ISLPED)*, 2011, vol. 1, pp. 73–78.

[41] H. Luo, J. Hu, L. Shi, C. J. Xue, and Q. Zhuge, “Two-step state transition minimization for lifetime and performance improvement on MLC STT-RAM,” *Proc. 53rd Annu. Des. Autom. Conf.*, p. 171, 2016.

[42] L. Jiang, B. Zhao, Y. Zhang, and J. Yang, “Constructing large and fast multi-level cell STT-MRAM based cache for embedded processors,” *DAC ’12 Proc. 49th Annu. Des. Autom. Conf.*, vol. 1, pp. 907–912, 2012.

[43] J. Wang, P. Roy, W. F. Wong, X. Bi, and H. Li, “Optimizing MLC-based STT-RAM caches by dynamic block size reconfiguration,” *2014 32nd IEEE Int. Conf. Comput. Des. ICCD 2014*, vol. 1, pp. 133–138, 2014.

[44] X. Chen, N. Khoshavi, J. Zhou, D. Huang, R. F. Demara, J. Wang, W. Wen, and Y. Chen, “AOS : Adaptive Overwrite Scheme for Energy-Efficient MLC STT-RAM Cache,” *Proc. 2016 53rd ACM/EDAC/IEEE Des. Autom. Conf.*, pp. 2–7, 2016.

[45] L. Liu, P. Chi, S. Li, Y. Cheng, and Y. Xie, “Building Energy-Efficient Multi-Level Cell STT-RAM Caches with Data Compression,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, vol. 1, pp. 751–756.

[46] E. E. Binkert, “The Gem5 Simulator,” *SIGARCH Comput. Arch. News*, vol. 39, no. 2, pp. 1–7, 2011.

[47] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value locality and load value prediction,” *Proc. seventh Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS-VII*, no. October, pp. 138–147, 1996.

[48] B. Panda and A. Seznec, “Dictionary sharing: An efficient cache compression scheme for compressed caches,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[49] Jun Yang, Youtao Zhang, and R. Gupta, “Frequent value compression in data caches,” in *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, no. Cc, pp. 258–265.

[50] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, “C-pack: A high-performance microprocessor cache compression algorithm,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 18, no. 8, pp. 1196–1208, 2010.

[51] S. Sardashti, A. Seznec, and D. a. Wood, “Skewed Compressed Caches,” *2014 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, pp. 331–342, 2014.

[52] S. Sardashti, A. Seznec, and D. A. Wood, “Yet Another Compressed Cache,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 1–25, Sep. 2016.

[53] A. R. Alameldeen and D. A. Wood, “Adaptive Cache Compression for High-Performance Processors,” *ACM SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 212, Mar. 2004.

[54] A. Arelakis and P. Stenstrom, “SC2: A statistical compression cache scheme,” *Proc. - Int. Symp. Comput. Archit.*, pp. 145–156, 2014.

[55] A. Arelakis, F. Dahlgren, and P. Stenstrom, “HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods,” *Proc. 48th Int. Symp. Microarchitecture*, pp. 38–49, 2015.

[56] S. Sardashti and D. A. Wood, “Decoupled compressed cache,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*, 2013, pp. 62–73.

[57] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Exploiting compressed block size as an indicator of future reuse,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 51–63.

[58] J. S. Liptay, “Structural aspects of the System/360 Model 85, II: The cache,” *IBM Syst. J.*, vol. 7, no. 1, pp. 15–21, 1968.

[59] A. Seznec, “Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio,” *Proc. 21 Int. Symp. Comput. Archit.*, pp. 384–393, 1994.

[60] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers,” *Proc. - Int. Symp. High-Performance Comput. Archit.*, pp. 63–74, 2007.

[61] 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 J. Res. Dev.*, vol. 45, no. 2, pp. 271–285, 2001.

[62] M. Ekman and P. Stenstrom, “A robust main-memory compression scheme,” *Proc. - Int. Symp. Comput. Archit.*, vol. 0, no. C, pp. 74–85, 2005.

[63] J. L. Núñez and S. Jones, “Gbit/s lossless data compression hardware,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 3, pp. 499–510, 2003.

[64] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Base-delta-immediate compression,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT ’12*, 2012, p. 377.

[65] A. Seznec, “A case for two-way skewed-associative caches,” in *Proceedings of the 20th annual international symposium on Computer architecture - ISCA ’93*, 1993, vol. 21, no. 2, pp. 169–178.

BIOGRAPHICAL SKETCH

Qi Zeng studied electronic engineering and information science in University of Science and Technology of China from 2008 to 2012. After finishing his undergraduate degree, he came to University of Florida and started pursuing his doctoral degree on computer science, where he is working on next generation cache design with new memory technology.

For 2015 summer, he worked in Memory Research Group of Intel Labs and proposed several engineering innovations on cache design together with his mentor.