MITIGATING LIMITED PCM WRITE BANDWIDTH AND ENDURANCE IN HYBRID MEMORY SYSTEMS

by

Yu Du

B.S. in Computer Science, Shanghai Jiao Tong University, 2001

M.S. in Computer Science, Shanghai Jiao Tong University, 2004

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Yu Du

It was defended on

April 8, 2015

and approved by

Prof. Rami Melhem, PhD, Department of Computer Science

Prof. Bruce R. Childers, PhD, Department of Computer Science

Prof. Daniel Mossé, PhD, Department of Computer Science

Prof. Hai Li, PhD, Department of Electrical and Computer Engineering

Dissertation Director: Prof. Rami Melhem, PhD, Department of Computer Science

MITIGATING LIMITED PCM WRITE BANDWIDTH AND ENDURANCE IN HYBRID MEMORY SYSTEMS

Yu Du, PhD

University of Pittsburgh, 2015

With the rise of big data and cloud computing, there is increasing demand on memory capacity to solve problems of large sizes and consolidate computation tasks. For large capacity memory systems, DRAM is a significant source of energy consumption. Non-volatile memory, such as Phase-Change Memory (PCM), is a promising technology for constructing energy-efficient memory. Unlike DRAM, PCM has negligible background (static) power and allows high density packaging. But PCM also has limited write bandwidth and write endurance. Hybrid memory systems have been proposed to combine the high-density and low standby power of PCM with the good write performance of DRAM.

This thesis addresses two challenges which are unique to hybrid memory systems. The first challenge is the *limited PCM bandwidth*, which can become a performance bottleneck. The second challenge is the *non-contiguous physical memory* due to retired memory pages. Since PCM cells have limited write endurance, it is inevitable to gradually have increased number of uncorrectable errors during the lifetime. Memory pages that have detected errors are normally retired by the OS, which create unusable "holes" in the physical memory. These unusable holes make it difficult to construct traditional superpages, which can incur significant performance overhead.

In this thesis, I propose three solutions to address these two challenges. First, I observed that an unbalanced distribution of modified data bits among PCM chips significantly increases PCM write time and hurts effective write bandwidth. I propose new XOR-based mapping schemes between program data bits and PCM cells to improve PCM write throughput by spreading modified data bits evenly among PCM chips. Second, I propose a compressed DRAM cache scheme to improve

DRAM effective capacity and reduce write traffic to PCM. A new adaptive delta-compression technique for modified data is used to achieve a large compression ratio. Third, I propose Gap-tolerant Sequential Mapping, a new memory page mapping scheme, to construct superpages from non-contiguous physical memory. The proposed three solutions have simple and practical designs, and can be easily adopted in future hybrid memory systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ALGORITHMS

1.0 INTRODUCTION

1.1 GOAL

Driven by multi-core processors and data center applications, there is an increasing demand on main memory capacity to solve large problem sizes and consolidate computation tasks. For example, Intel's 15-core 30-thread server processor [30] can support 12TB of memory in an 8-socket system. Along with the demand for colossal memory capacity, energy consumption of DRAM is now a significant portion of total energy consumption. Even equipped with Samsung's 20nm class 1.2V DDR4 DRAM, the average power of 12TB DRAM for a typical server workload is more than 2 Kilowatts [51]. There is a critical need to find energy-efficient memory organizations [57].

Phase change memory (PCM) is a promising non-volatile memory for constructing energy-efficient systems [34, 44, 50, 69]. Unlike charge-based DRAM, PCM uses different resistance states of phase change material to represent data. When used as a large capacity main memory, PCM has two major advantages. First, PCM cells have good scalability and can achieve excellent memory density. A high performance PCM cell design has been demonstrated in sub-20*nm* technology [33]. With a multi-level cell design (MLC), PCM can achieve even greater densities by storing multiple bits in one cell. Second, and most importantly, PCM has negligible background (static) power due to its non-volatile nature. Given the same power budget, PCM can achieve much higher capacity than DRAM. With low background power, PCM allows high-density memory chip packagings (e.g. 3D TSVs) without having a severe heat dissipation issue. Despite its advantages, PCM also has drawbacks. It has long write latency, high write energy, and limited write endurance in comparison to DRAM. Specifically, PCM devices have severely constrained write bandwidth. For example, a prototype 20*nm* 8Gb PCM chip [13] has been demonstrated that has a read bandwidth of 800MB/s and a write bandwidth of 40MB/s. When DRAM is replaced with PCM, the

limited PCM write bandwidth can become a major performance bottleneck.

Hybrid memory systems [69, 37, 50] have been proposed with DRAM and PCM to combine the high-density and low standby power of PCM with the good write performance of DRAM. Frequently-modified data are stored in DRAM to reduce write traffic to PCM. Common hybrid organizations use DRAM as a cache or an extension to PCM. When used as a cache, the DRAM's address space is not visible to the software. When used as an extension, OS is responsible for memory page migration between DRAM and PCM. For hybrid memory systems, memory compression can be used to increase DRAM effective capacity, which allows more frequently modified data to be cached in DRAM to reduce memory write traffic to the non-volatile memory.

Besides the challenge of limited PCM write bandwidth, memory reliability is another challenge for hybrid memory systems. To avoid system crashes caused by memory errors, modern OSes support page retirement, a self-healing capability which removes physical memory pages containing memory errors from the system's address space [28]. For DRAM, retired memory pages only account for a small portion of total memory capacity(between 0.1% and 10%) [28]. With process scaling, the percentage of retired pages is expected to increase due to increased process variations [38]. For PCM, page retirement is a more critical issue. Compared to DRAM, PCM cells have very limited write endurance (10^6 to 10^8 writes on average). Some cells have much lower write endurance than other cells. Even with various wear leveling and error correction techniques, it is inevitable for PCM to gradually have increased number of retired memory pages during its lifetime.

With the rise of big data and cloud computing, workload memory footprints keep increasing, putting more pressure on the virtual memory subsystem [5]. Superpages are mandatory for memory-intensive workloads with large memory footprints and random access patterns. A traditional superpage is a large virtual memory page that is mapped to an equivalent amount of contiguous physical memory pages. Superpage mapping assumes that physical memory does not contain retired pages, which is an important technique to improve memory resilience: the OS avoids allocating physical pages that have detected errors. Retired pages create unusable "holes" in the physical memory. Even a small percentage of retired pages makes it very difficult to find enough contiguous memory to form superpages.

1.2 OVERVIEW OF THE PROPOSED APPROACHES

Since **limited write bandwidth of non-volatile memory** and **non-contiguous physical memory** are two fundamental new challenges for hybrid memory systems, this thesis explores three architecture techniques to address the new challenges.

Balanced PCM Bit Mapping studies the mapping between program data bits and PCM cells to **improve PCM write throughput** by spreading modified data bits evenly among PCM chips. For each PCM write, the data bits of the write request are typically mapped to multiple cell groups and processed in parallel. I observed that an unbalanced distribution of modified data bits among cell groups significantly increases PCM write time and hurts effective write bandwidth. To address this issue, I first uncover the cyclical and cluster patterns for modified data bits. Next, I propose *double XOR mapping (D-XOR)* to distribute modified data bits among cell groups in a balanced way. D-XOR can reduce PCM write service time by 45% on average, which increases PCM write throughput by $1.8\times$. As error correction (redundant bits) is critical for PCM, I also consider the impact of redundancy information in mapping data and error correction bits to cell groups. My techniques lead to a 51% average reduction in write service time for a PCM main memory, which increases IPC by 12%.

Delta-compressed DRAM Caching studies an adaptive delta-compressed DRAM caching scheme to **improve DRAM effective capacity and reduce write traffic** to PCM. Since the write bandwidth is severely restricted in PCM devices, it is more important to reduce PCM write traffic than to reduce PCM read latency for write-intensive applications. To reduce the number of PCM writes, I propose a DRAM cache organization that employs compression. A new delta-compression technique for modified data is used to achieve a large compression ratio. My approach can selectively and predictively apply compression to improve its efficiency and performance. It is designed to facilitate adoption in existing main memory compression frameworks. I describe an instance of how to incorporate delta-compression in IBM's MXT memory compression architecture when used for DRAM cache in a hybrid main memory. For fourteen representative memory-intensive workloads, on average, the proposed delta-compression technique reduces the number of PCM writes by 54.3%, and improves IPC performance by 24.4%.

Gap-Tolerant Superpage studies a new memory page mapping scheme to **construct super-**

pages from non-contiguous physical memory. **Gap-tolerant Sequential Mapping** (GTSM) is proposed to allow superpages to be formed even in the presence of retired physical pages. A new page table format is also proposed to support GTSM. This format has similar storage efficiency as traditional superpaging to hold address translations in the last-level cache. To further compress the page table and improve cache hit rates for address translation in large memory footprint workloads, I also propose an extended format that reduces the page table size by 50%. In comparison to an ideal memory without any retired physical pages, I show that my technique, with retired pages, achieves nearly 95.8% of the performance of traditional 2MB superpaging.

1.3 THESIS ORGANIZATION

The rest of the thesis is organized as follows: Chapter 2 is a background study of the related work for phase change memory, hybrid main memory, memory compression, memory error and virtual memory. Chapter 3 introduces new XOR-based bit mapping functions to improve PCM write throughput. Chapter 4 describes a new delta-compressed DRAM caching scheme to reduce PCM write traffic. Chapter 5 develops a new page table design to construct superpages from memory with retired pages. Finally, Chapter 6 presents conclusions, as well as future work.

2.0 BACKGROUND AND RELATED WORK

2.1 PHASE CHANGE MEMORY

Limited write bandwidth is a major performance bottleneck for PCM. The limited write bandwidth is due to three reasons. First, it takes much longer to program a PCM cell than a DRAM cell. Unlike charge-based DRAM, PCM uses different resistance states of phase change material to represent data. Precisely-controlled SET and RESET pulses are used to program PCM cells (to heat and cool a cell). For single-level cell (SLC) PCM, it takes 50-100ns for RESET programming (1→0) and 150-400ns for SET programming (0→1) [13, 35]. Second, programming a PCM cell requires much higher programming current than programming a DRAM cell. Given a fixed programming current budget, a PCM device cannot simultaneously program all data bits of a write request. Instead, the data bits are statically divided into small divisions and programmed sequentially [22]. Third, process variation has a non-negligible impact on cell programming time.

Due to process variation, different cells requires different programming currents. Using a single high programming current level to program all cells is not a valid choice because the high programming current will significantly degrade the write endurance of cells that requires a much smaller programming current. Hence, PCM write circuits typically use a staircase programming policy which gradually increases the programming current. With such a staircase policy, to successfully program the cells that need a high programming current, multiple programming iterations are used until the desired programming current level is reached.

Programming current is the major constraint which limits the number of cells that can be concurrently programmed per chip [22]. For example, Samsung's 20nm 8Gb PCM chip supports only simultaneous programming of 128 cells with the default power supply [13]. Therefore, PCM devices have asymmetric read and write data widths. For PCM reads, all data bits are read con-

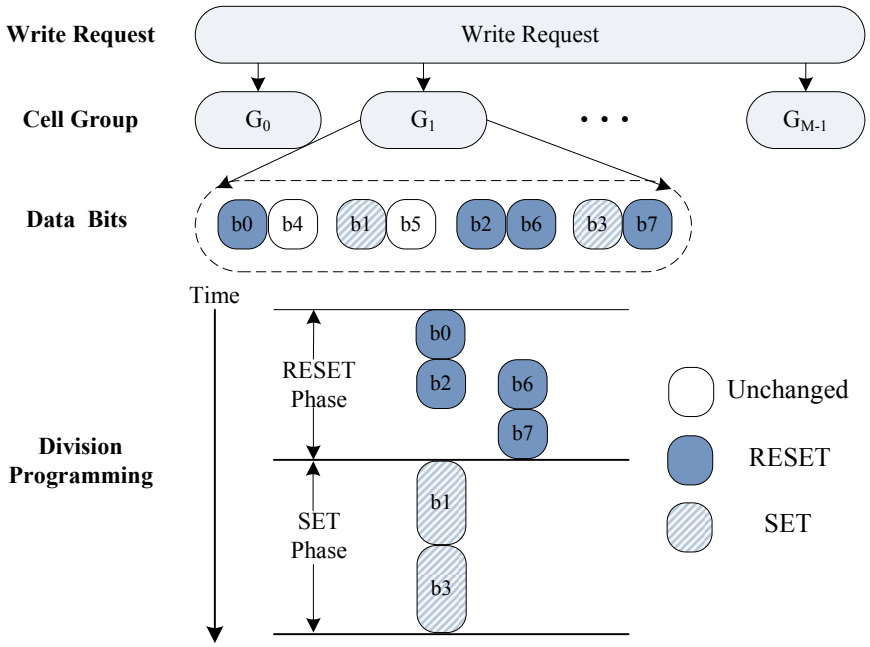


Figure 1: Division program operations in PCM devices.

currently, while for PCM writes, the modified bits of each cell group are written using *division program operations* [22]. For each division program operation, a subset (i.e., a *division*) of memory cells among a cell group are programmed rather than all cells in the group. Figure 1 shows an example of 2X division program operation [22]. For each cell group, the data bits are statically divided into divisions. 2X means that each division consists of two cells. In this example, an 8-bit cell group is divided into four divisions: (0, 4), (1, 5), (2, 6) and (3, 7). Cells in the same division can be concurrently programmed, while cells in different divisions must be programmed in different iterations.

Divisions in the same cell group are programmed in a fixed sequential order. A division will be skipped if it has no cell to program. The programming process is divided into two phases: a RESET phase and a SET phase. In the RESET (SET) phase, only cells that need RESET (SET) programming are programmed. In Figure 1, cells 2 and 6 are programmed concurrently because they belong to the same division. Cells 3 and 7 are programmed separately because they have different programmed states. With division program operations, the programming time of each

cell group is no longer a constant. On average, a cell group with more modified bits requires more programming iterations and tends to have a longer programming time. For each write request, the write service time depends on the cell group that takes the longest programming time.

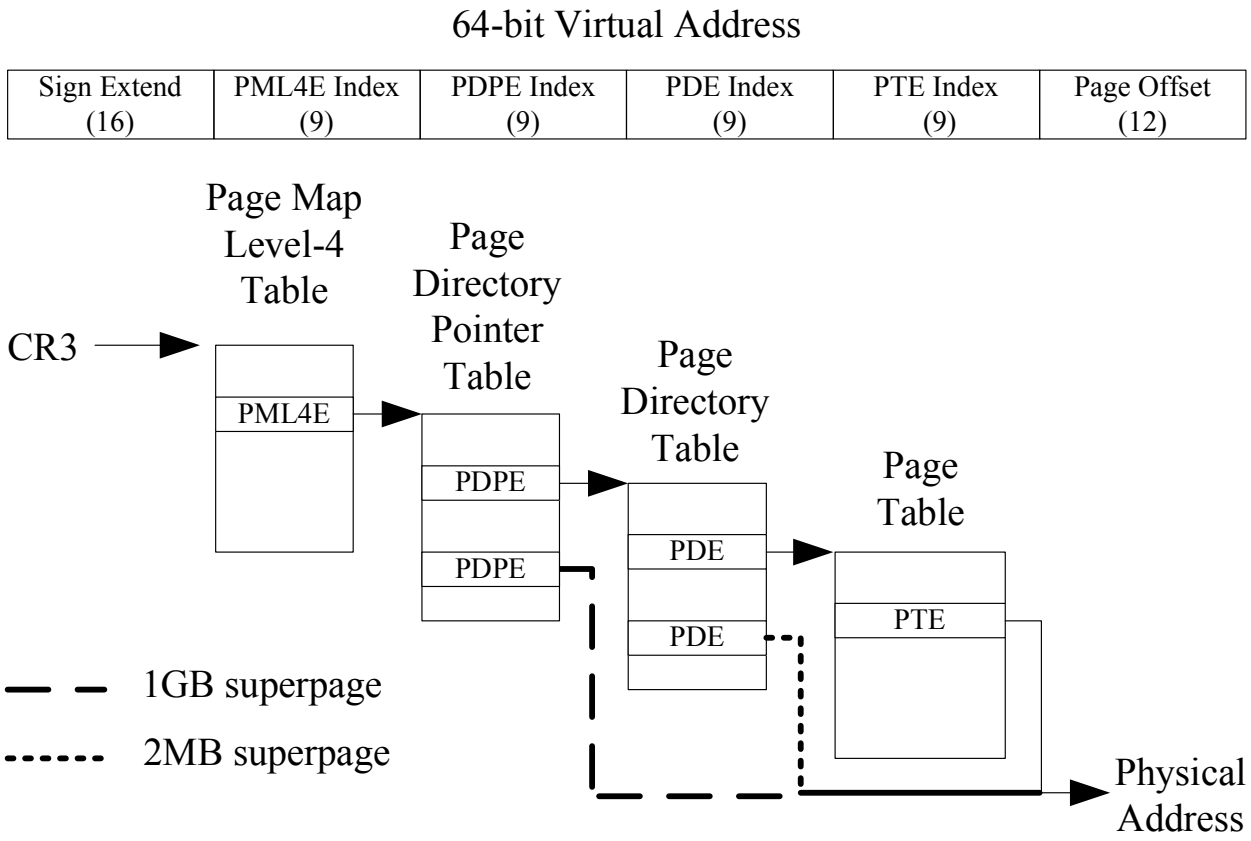
2.2 VIRTUAL ADDRESS TRANSLATION OVERHEAD AND SUPERPAGE

Virtual memory mechanisms use page tables to map between virtual pages and physical pages for every memory access. To speed up translation, physical addresses of recently-accessed virtual pages are cached in the TLB. On a TLB miss, a hardware page walker traverses the page table to translate the virtual page address. We use x86-64 architecture as an example to explain why superpage can reduce virtual address translation overhead.

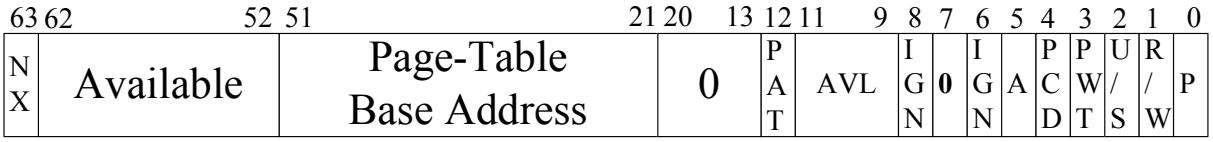
In x86-64, as shown in Figure 2(a), the page table has four levels, and a system register (*CR3*) points to the PT root node. The corresponding translation entry at each level is *Page-Map Level-4 Entry* (PML4E), *Page Directory Pointer Entry* (PDPE), *Page Directory Entry* (PDE) and *Page Table Entry* (PTE). For each valid 4KB virtual page, the translation entries (PML4E, PDPE and PDE) point to the base address of the next level node. The size of the translation entry at each level is 8 bytes. With a 4KB page, there are 512 translation entries per node, which are indexed by 9 virtual address bits. Only 48 virtual address bits are used in current x86-64 implementations: the high 36 bits (9 x 4) are used to traverse the page table levels and the low 12 bits are the page offset.

Besides the TLB, recently accessed translation entries are also cached in the MMU as partial translations, which can be used to speed up page walking [3]. For example, PDE entries can be cached in a PDE cache. If the PDE of a virtual address hits in the PDE cache, the page walker needs to access only the last-level PTE to complete address translation.

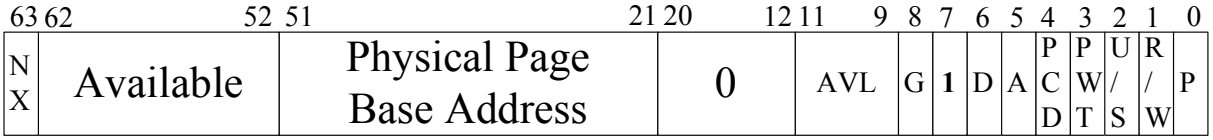
X86-64 superpage implementation has a similar structure. Because the mapping is one-to-one, a 2MB superpage needs only a three-level page table (PML4, PDPE and PDE). The 7th bit of a PDE indicates whether the PDE points to a page table of PTEs, or to the physical base address of a 2MB superpage. Figures 2(b) and 2(c) show the format of PDE as a 4KB-page and 2MB-page PDE, respectively. Similarly, for a 1GB superpage, the 7th bit of a PDPE indicates whether a PDPE points to a page directory table of PDEs, or to the physical base address of the superpage.



(a) x86-64 page table



(b) 4KB-page PDE



(c) 2MB-page PDE

Figure 2: VA-to-PA translation in the x86-64 architecture.

2.3 RELATED WORK

2.3.1 Phase Change Memory

Despite promise, PCM has weaknesses, which are the subject of much research. Wear-leveling techniques have been proposed to evenly distribute writes to PCM memory lines [70, 47, 54]. Fault-tolerance techniques have been proposed to protect PCM chips from weak cell failures [52, 43, 65, 31, 32].[1] Write-pausing has been proposed to reduce the performance penalty from long PCM write service time [45].

2.3.2 Hybrid Main Memory

DRAM/PCM hybrid main memory is a natural solution for future large-capacity energy-efficient main memory because it combines the high-density and low standby power of PCM with the good write performance of DRAM. With hybrid main memory, frequently modified data are transparently cached in DRAM to offload write traffic to PCM. Qureshi *et al.*[48] proposed to add a DRAM buffer between the CPU and the PCM main memory to cache frequently accessed data. Ferreira *et al.*[23] proposed PMMA architecture which integrates DRAM page cache with PCM main memory using an improved page replacement policy. Zhang and Li [69] proposed a 3D-stacked DRAM/PCM hybrid memory architecture which uses an OS-based page migration mechanism to cache hot-modified pages in DRAM. Ramos *et al.*[50] proposed a hardware-based page migration mechanism for hybrid memory which puts both frequently-modified and frequently-read pages into DRAM. My work considers the use of memory compression to improve DRAM effective capacity and reduce PCM write traffic in hybrid systems which is orthogonal to the above works.

2.3.3 Memory Compression

Memory compression is a common technique to allow more data to be stored in the memory and has been well studied in DRAM-only systems. Douglis [19] studied using compression to free up memory pages to reduce paging overhead. Ekman and Stenstrom [21] proposed a low latency main

[1] Weak cells have much lower write endurance than other PCM cells.

memory compression scheme based on the FPC compression algorithm. IBM Memory Expansion Technology (MXT) technology [62] was introduced as a hardware-based high performance architecture and is implemented in commercially available chips. Suel and Memon [59] use delta-compression to reduce data traffic for remote file synchronization. Zhang and Li [69] proposed to apply compression to PCM data. Their proposal is not to improve DRAM effective capacity, but to reduce the number of PCM bits that need to be updated. The studies in [19, 62] focus on DRAM-only systems. I propose delta-compressed caching which is not discussed by traditional DRAM-only memory compression. I give a detailed example to show how to extend an existing compression architecture to support delta-compression for hybrid memory.

2.3.4 Memory Error and Page Retirement

Recent studies show that modern DRAM error rates are orders of magnitude higher than previously reported [53, 28]. Error Correcting Codes (ECC) are commonly used to protect memory from one or multiple bit errors. Recent studies also showed that memory blocks that suffer from correctable memory errors are much likely to subsequently face uncorrectable errors [53]. A field study showed that retiring 1% of pages can cover 92% of memory errors [28]. Memory errors can be tolerated using managed runtime systems [26], but this requires the program to be written in managed code (e.g., Java).

2.3.5 TLB and Superpage

Recent study shows that traditional page-based virtual memory has significant performance overhead for big-memory workloads [5]. Many new TLB designs have been proposed to alleviate the problem by reducing TLB translation overhead. Barr *et al.*[4] proposed a speculative translation scheme which exploits the predictable behaviour of reservation-based physical memory allocators. Bhattacharjee *et al.*[7] proposed shared last-level TLBs for chip multiprocessors which improve the effective capacity of TLBs. Shekhar Srikantaiah and Mahmut Kandemir [58] proposed a distributed variation of shared last-level TLBs, which makes a balance between TLB capacity and TLB access latency. Pham *et al.*[42] proposed Coalesced Large-Reach TLBs (CoLT), which coalesces multiple virtual-to-physical page translations into a single TLB entry. Basu *et al.*[5]

proposed a TLB-less design to eliminate TLB translation overhead for big-memory workloads. My work studies new storage-efficient page table designs for non-contiguous phyiscal memory. It assumes that a small percentage of memory pages are retired and not available to use, which is significantly different from previous work.

Both software and hardware changes are necessary to support superpages. Talluri et al. discussed the tradeoffs and challenges to support superpages in hardware [60]. Ganapathy and Schimmel described possible ways to support superpages in the OS [25]. Navarro et al. described a design to transparently support superpages in the OS [39]. Zhang et al. described a design to map superpages to disjoint physical pages using traditional base page table format [68]. In their proposed design, page table still needs to be accessed when there is a cache miss. To the best of my knowledge, this is the first work to propose a new storage-efficient superpage format designed for memory with retired pages. By utilizing a block selection bitmap, a superpage is mapped to multiple equal-sized small memory blocks (i.e., physical pages) instead of a single large contiguous memory block.

There are much work on improving TLB performance. TLB hit rate can be improved by sharing TLB entries among CPU cores [8, 63, 7]. TLB miss penalty can be reduced by prefetching [8]. Recently, TLB coalescing has been studied to improve TLB reach [41, 42]. Similar to TLB coalescing, MMU cache coalescing has been proposed to reduce TLB miss penalty [6]. My work does not require any changes to TLB and is orthogonal to the work proposed for TLB performance improvement. For workloads with large memory footprints, improving the TLB performance alone is not enough to solve the problem.

3.0 BIT REMAPPING FOR BALANCED PCM CELL PROGRAMMING

3.1 PROBLEM STATEMENT

Unlike charge-based DRAM, non-volatile memory cells need much more programming power to change their states. Programming power is the major constraint which limits the number of PCM cells that can be concurrently programmed per chip [22]. For PCM writes, *division program operations* are used to program the modified bits [22]. First, data bits are mapped to multiple cell groups. For each division program operation, a subset (i.e., a *division*) of memory cells among a cell group are programmed rather than all cells in the group. Divisions in the same cell group are programmed in a fixed sequential order.

With division program operations, the programming time of each cell group is not a fixed value. A cell group with more modified bits are likely to have a longer programming time. Memory service time of a write request is determined by the cell group that has the longest programming time. Therefore, the mapping function (abbreviated as *mapping*) between data bits and cell groups will impact the write service time. To illustrate the importance of the mapping, Figure 3 shows an example in which the size of the write request is sixteen bits, seven of which are modified. In this example, the sixteen data bits are mapped to four cell groups. Figure 3(a) shows a possible mapping in which adjacent data bits are directed to the same group. Notice that bits 4-7 are adjacent and modified. They are mapped to the same cell group and cause a bottleneck on write service time for the write request. Figure 3(b) shows an alternate mapping in which adjacent data bits are distributed to different cell groups. With this new mapping, the four adjacent modified bits are distributed more uniformly among cell groups and the write service time is reduced. The intent of this example is not to show that the second mapping is always better than the first one, but to show that the mapping has an impact on the write service time. A good mapping should spread

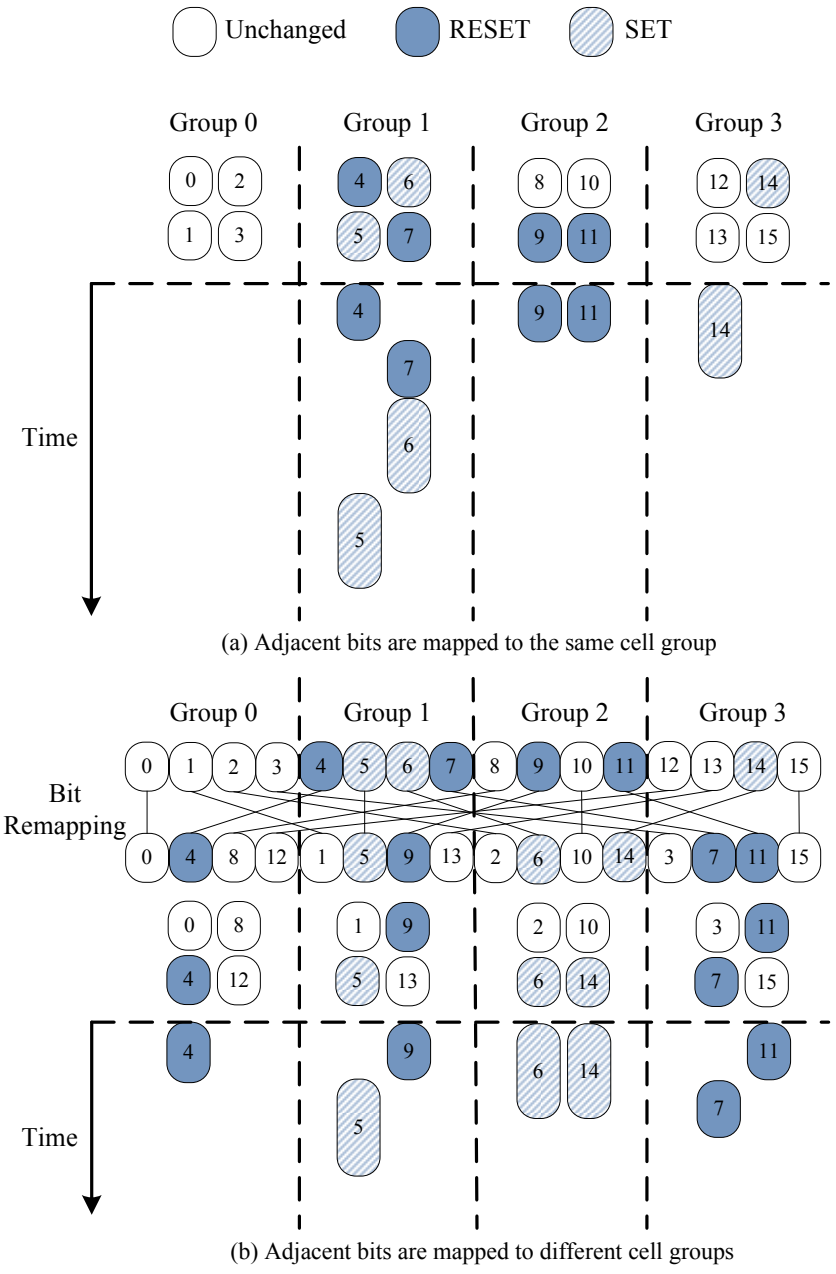


Figure 3: An example showing that the mapping between data bits and PCM cells can affect write service time.

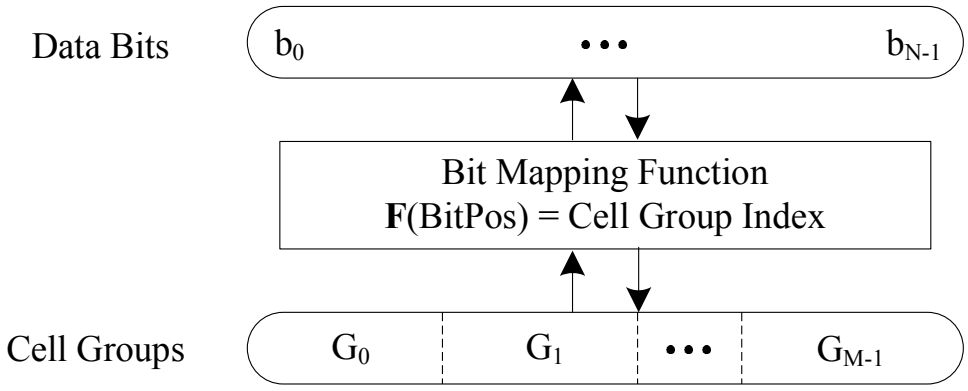


Figure 4: Bit mapping function.

the modified bits among cell groups, which depends on the distribution of the modified bits in the write request.

A *mapping function* F defines a static partition of N data bits from write requests into M PCM cell groups (N/M cells per cell group). Ideally, the problem is to find the function F that minimizes the longest programming time among all cell groups. Since over the long term, a modified bit is equally probable to be SET or RESET, the problem is approximated to minimize the imbalance in the number of modified data bits among all cell groups, which is achievable with a better mapping function. The concept is shown in Figure 4.

The input of the mapping function F is a n-bit binary address $a_{n-1}a_{n-2}...a_0(n = \log N)$ representing the bit position of each data bit. The output of the mapping function F is an m-bit binary number $t_{m-1}t_{m-2}...t_0(m = \log M)$ representing the index of the cell group to which the data bit is mapped. A constraint on F is that the number of data bits mapped to each cell group should be equal. Otherwise, some cell groups will not have enough PCM cells to establish a 1:1 mapping between data bits and cells. This work is focused on the mapping between data bits and cell groups, the order of mapped cells inside a cell group is not considered, which is assumed to be the same as the order of data bits.

3.2 DISTRIBUTION OF MODIFIED BITS

3.2.1 Distribution Patterns

Given that the distribution of modified bits can affect PCM write service time, the *bit flip rate* is characterized to understand the distribution (patterns) of modified bits in write requests. The bit flip rate at a given bit position is defined as the probability that the bit at that position changes its state (flips) on a memory write. To calculate the bit flip rate at position K, the number of flips in the K^{th} bit of all write requests is counted and divided by the total number of write requests. I characterized all bit positions in the memory write traces (after a 32MB DRAM cache) for fourteen applications.[1] The write granularity of collected memory traces are 256 bytes. Figure 5 shows the bit flip rate histogram of the first 512 bit positions; the remaining positions are similar. Applications are ordered by the average percentage of modified data bits per write request in ascending order. As the figure shows, the bit flip rate histograms differ for each application. Two general patterns in the bit flips are identified from the figure: *cyclical* and *cluster*.

Cyclical Pattern. In this pattern, the flip rate histogram shows cyclical fluctuations. As shown in Figure 5, a typical cycle length can be 32 bits, 64 bits or 128 bits (*mcf* has a cycle length of 512 bits). Bit positions with the same low address bits but different high address bits tend to have similar bit flip rates. To distribute modified data bits with a cyclical pattern, the high address bits should be used as an index to map the bits to cell groups. For example, in application *libquantum*, data bits with the same low 7-bit address should be mapped to different cell groups.

Cluster Pattern. Modified bits may also tend to aggregate into clusters. For example, in Figure 5, *astar*, *canneal* and *mcf* have most of their modified bits clustered in only a few bytes. To quantify the clustering of modified bits at byte granularity, I also characterize the average number of bytes that cover most modified bits of each memory write (choosing those bytes that have the most number of modified bits). Figure 6 shows the average number of bytes with 90% coverage. From the figure, *astar*, *mcf*, *omnetpp* and *canneal* have strong clusters. The remaining applications show moderate clusters, with 20% to 60% of the bytes covering 90% of the modified bits. *Libquantum* is a special case. It requires a few bytes to cover most modified bits because it only has one bit

[1] Twelve write-intensive benchmarks from SPEC CPU2006, SPEC JBB2005 and the only write-intensive benchmark (*canneal*) from the PARSEC suite are selected.

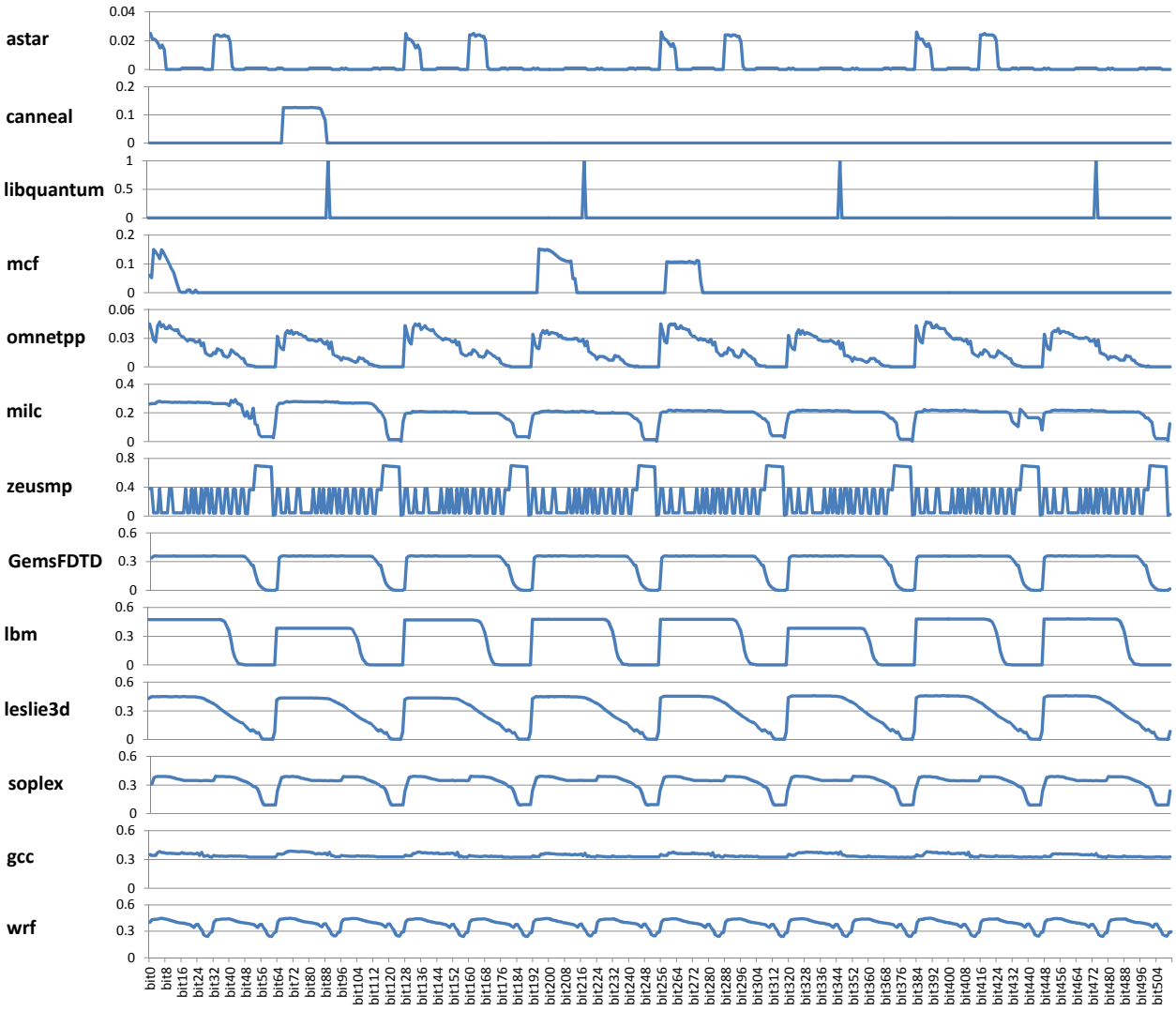


Figure 5: Average bit flip rate at different bit positions (first 512 bits of 256-byte memory requests).

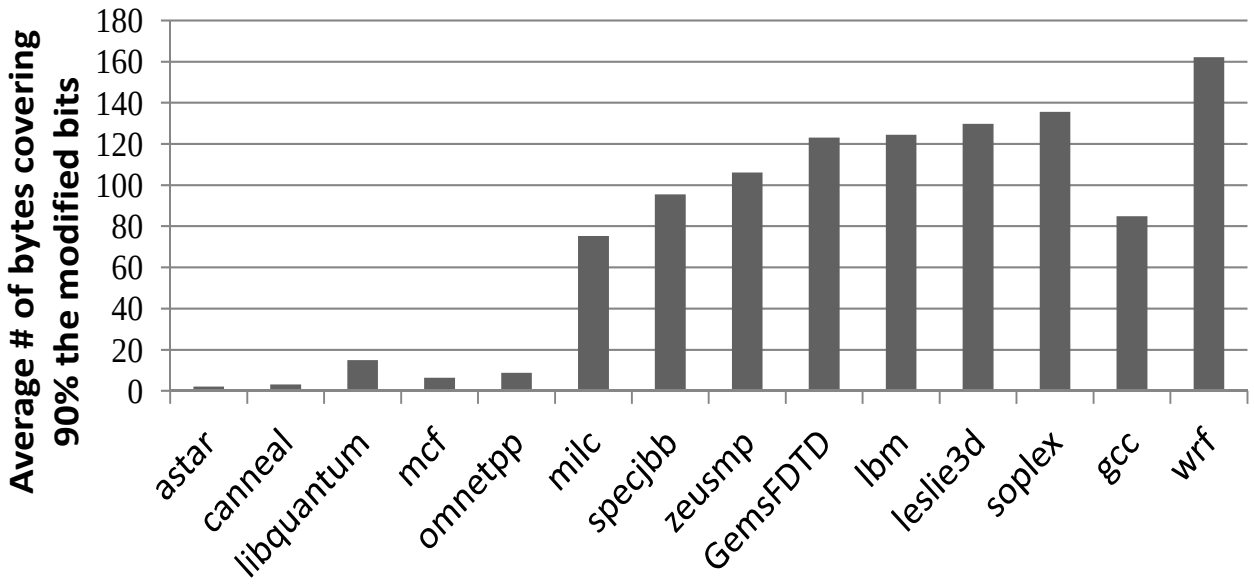


Figure 6: Average number of bytes covering 90% of the modified bits of 256-byte write requests.

modified per sixteen bytes. When clustered together, modified bits have the same high address bits but different low address bits. A good mapping function should use low address bits to distribute adjacent bits to different cell groups. For example, in *canneal*, adjacent data bits should be mapped to different cell groups.

The distribution of modified data bits is affected by the data structure and data type used in the program. Programs whose modified bits have a cluster pattern tend to update only some data object fields, possibly even just a part of a field. These programs usually have more random object access. Programs whose modified bits have a cyclical pattern often update specific fields in arrays of objects. Programs dominated by integer types tend to modify adjacent bits within an integer, while programs dominated by floating point (FP) types modify bits with a more random distribution within the float. Also, FP programs typically modify more bits per write. These behaviours are due to integer and FP encoding, and how the types are used. For example, *libquantum* has an array of objects with an integer flag. One bit in the integer is updated frequently, causing spikes as seen in Figure 6. The flip rates for other programs can be similarly explained according to data structures

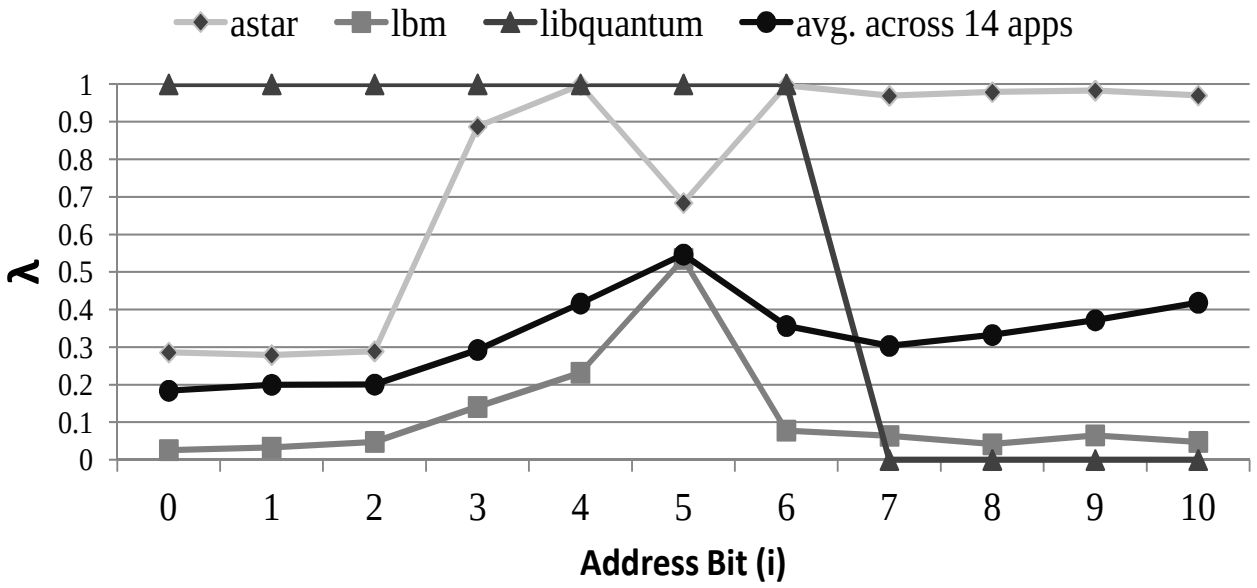


Figure 7: Distribution imbalance of modified bits at different address bits.

and types.

3.2.2 Distribution Imbalance of Modified Bits

To determine which address bits should be used in the mapping function, the distribution imbalance of modified bits at different address bits for a data bit position is characterized. For the i^{th} address bit, modified bits can be divided into two subsets:

$S_0 = \{b_k, k = a_{n-1}, \cdots, a_{i+1}, 0, a_{i-1}, \cdots, a_0\}$ and

$S_1 = \{b_k, k = a_{n-1}, \cdots, a_{i+1}, 1, a_{i-1}, \cdots, a_0\}$.

S_0 includes all modified bits with the a_i bit of the bit position equal to 0 and S_1 includes all modified bits with the a_i bit of the bit position equal to 1.

Similar to [40], I use the *percent imbalance* metric, $\lambda(i)$, to characterize the distribution imbalance of modified bits at the i^{th} address bit.

$$\lambda(i) = \left(\frac{max(|S_0|, |S_1|)}{(|S_0| + |S_1|)/2} - 1 \right) \times 100\% \tag{3.1}$$

When $\lambda(i) = 0$, modified bits are equally distributed between S_0 and S_1. When $\lambda(i) = 1$, there is no modified bit in S_0 or S_1. The distribution imbalance of an address bit is calculated by averaging $\lambda(i)$ over all write requests of each application.

An address bit with low λ implies that the address bit is a good candidate for spreading modified bits into two balanced subsets. In general, a good mapping function should utilize address bits with low λ.

In Figure 7, λ for each of the eleven address bits (256-byte write request) is shown for three representative applications: *astar*, *lbm* and *libquantum*. *Astar* has a strong cluster pattern, *libquantum* has a cyclical pattern, and *lbm* has both patterns. In the figure, it is apparent that each application has a unique λ profile. To get a balanced distribution of modified bits, different applications should ideally use different address bits in the mapping of bits to cell groups.

The figure also shows the average λ over fourteen applications. From the figure, two address bit regions, on average, have low λ. One region is $a_6...a_{10}$, which reflects cyclical patterns and the other region is $a_0...a_2$, which reflects clusters at byte granularity. This observation is consistent with the two patterns described in the previous section. To design an effective mapping function, address bits from both regions should be used.

3.3 PROPOSED SOLUTION: BIT REMAPPING

3.3.1 XOR Mapping Function

From the distribution imbalance characterization, two mapping functions can be directly derived. As shown in Figure 8(a), to get an m-bit cell group index, high address bits can be used for the mapping function such that a data bit $b_i, i = a_{n-1}a_{n-2}...a_0$, is mapped to $Group_j$ where $j = a_{n-1}a_{n-2}...a_{n-m}$. When high address bits are used, adjacent data bits are mapped to the same cell group and the mapping function exploits the cyclical pattern. Alternatively, as shown in Figure 8(b), low address bits can be used, that is, a data bit $b_i, i = a_{n-1}a_{n-2}...a_0$, is mapped to $Group_j$ where $j = a_{m-1}a_{m-2}...a_0$. When low address bits are used, adjacent data bits are mapped to different cell groups, and the mapping function exploits the cluster pattern. However, each of

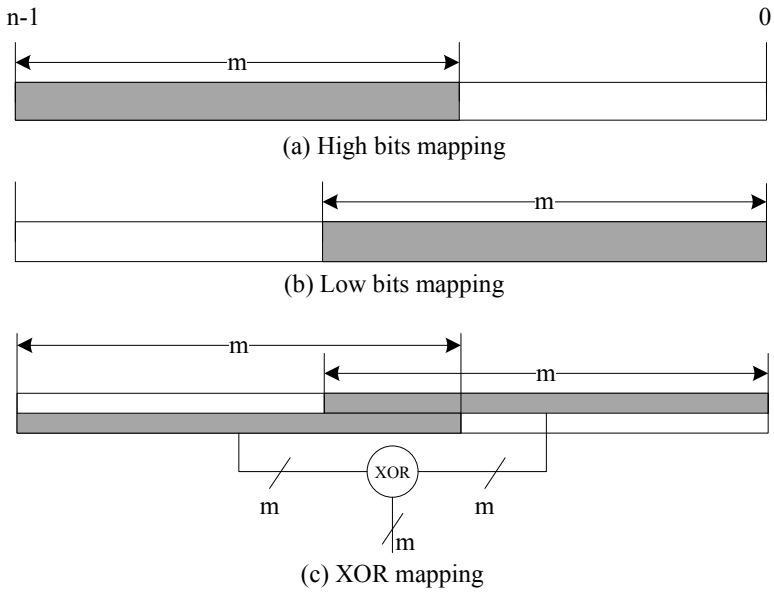


Figure 8: XOR mapping function.

these mapping functions has a limitation that only m bits out of n address bits are utilized.

As shown in Figure 8(c), I devised an *XOR mapping function* that combines low address bits with high address bits. This way, either pattern is incorporated seamlessly into the mapping. Similar to the high address and low address mappings, this new mapping distributes an equal number of data bits to each cell group. This is a necessary feature of any valid mapping function, which can be verified by calculating the cell group index of each bit position.

3.3.2 Support for Multiple Cell Group Sizes

An XOR mapping function is optimized for a specific cell group size. There are scenarios where a single mapping function is needed for multiple cell group sizes. For example, server memory configurations typically map each request to more memory chips (more cell groups) to improve reliability [17]. Also, a PCM chip can have a wider programming width (smaller cell group size) in the SET phase than in the RESET phase [22, 67] because the programming current of a SET pulse is much smaller than the programming current of a RESET pulse.

I use an example to illustrate why a single XOR mapping function is not the optimal solution

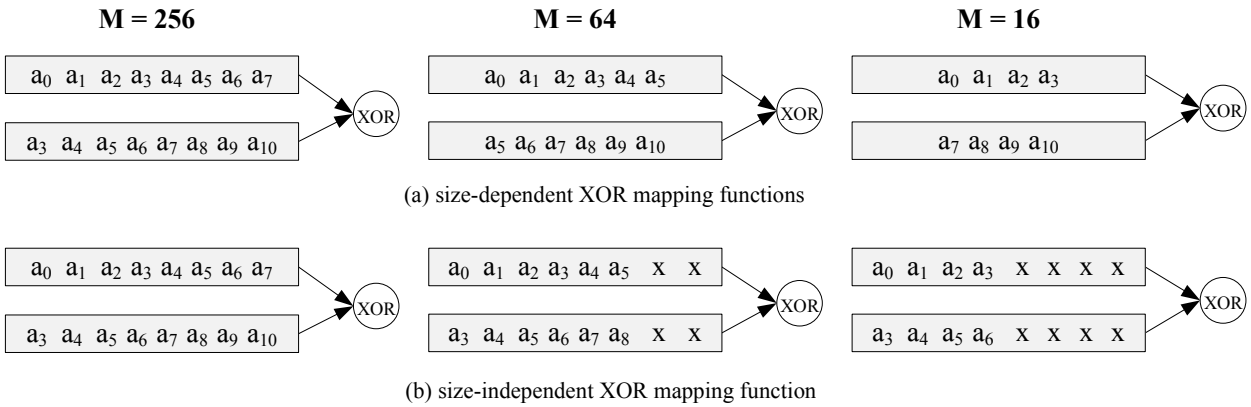


Figure 9: XOR mapping functions for different number of cell groups.

in these scenarios. Figure 9(a) shows XOR mapping functions that are size dependent. For each unique number of cell groups, M, a specific implementation of XOR mapping is used. Figure 9(b) shows how a single function can be used to support mapping different number of cell groups. The XOR mapping function is optimized for 256 cell groups. To map to fewer number of cell groups, the low bits of the mapping function are masked. The masking process is achieved by treating data bits in adjacent cell groups as a larger single cell group. Masking k bits of the mapping function means that every 2^k adjacent cell groups are treated as one group. Given that the order of data bits is unchanged, the masking process does not require any additional hardware support. The problem for masking is that fewer address bits are used in the mapping when M becomes smaller, which will degrade effectiveness.

To address this problem, I propose *double XOR mapping (D-XOR)*. In D-XOR, in addition to the XOR between low and high address bits, the high half of the high address bits is XORed with the low half of the high address bits. Figure 10 shows an example with eleven address bits. The mapping function does one XOR between $a_0...a_7$ and $a_3...a_{10}$ and a second XOR with $a_7...a_{10}$ padded with zeros. For M=16, all address bits still participate in the mapping function, although the low four bits of the function are masked.

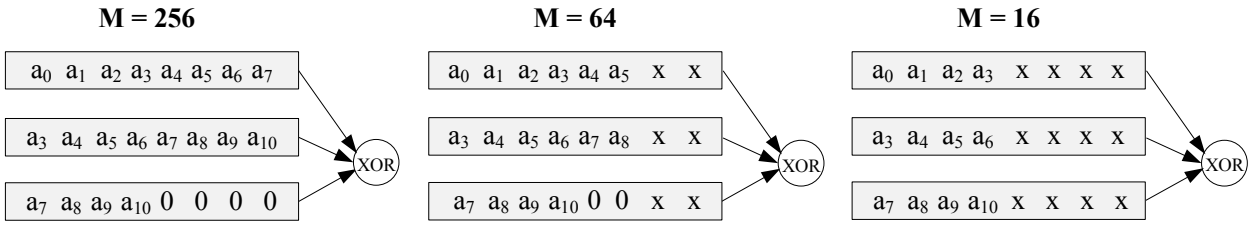


Figure 10: D-XOR mapping function.

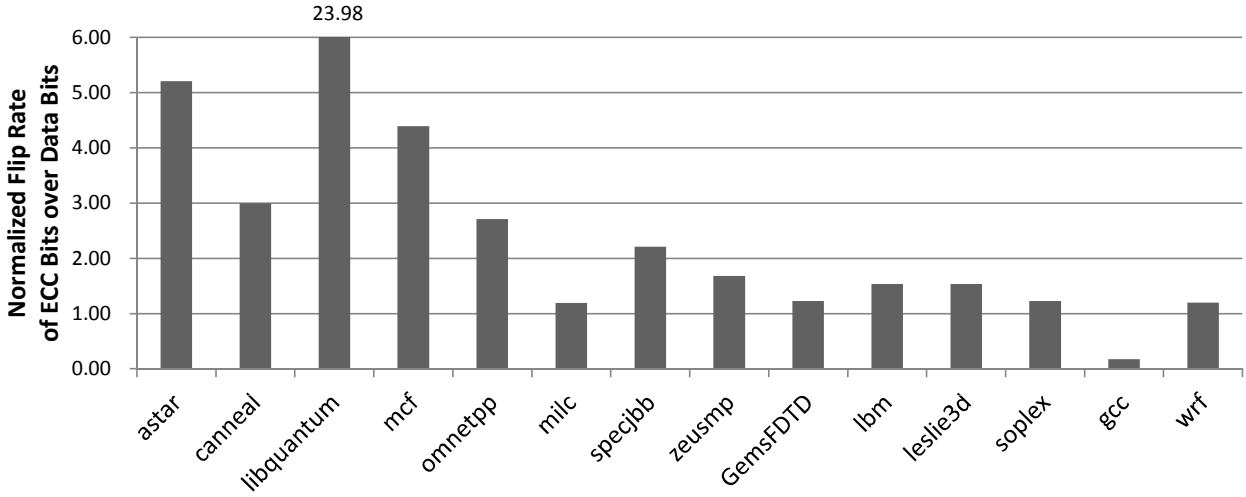


Figure 11: Flip rate of ECC bits normalized to data bits.

3.3.3 Support for PCM with Redundant Bits

To protect memory from data corruption due to transient errors, error correcting codes (ECC) are typically used in DRAM and PCM memory subsystems. Furthermore, PCM main memory typically uses a scheme to tolerate not only transient errors but also permanent errors in weak cells. To correct these errors, redundant bits are used. For typical DRAM solutions, extra DRAM chips store ECC bits [27], but for PCM, dedicated ECC chips can degrade write performance if ECC chips have more modified bits to write.

Figure 11 shows the flip rate of ECC bits normalized to the flip rate of regular data bits. The ECC code in this example is a (72, 64) SEC-DED ECC code with 8 ECC bits per 64-bit data block [27]. Since ECC is a checksum for data bits, ECC bits tend to have a higher flip rate than normal data bits. From the figure, in all applications, except *gcc*,[2] the ECC bits have high flip rates. For *astar*, *canneal*, *libquantum*, *mcf* and *omnetpp*, the ECC bit flip rates are much higher because most data bits are rarely modified. With a high flip rate, the PCM chips that store ECC bits have many more modified bits, which causes these redundant chips to become a write bottleneck.

Figure 12 shows my mapping function for redundant PCM storage. Either XOR or D-XOR mapping is used for data bits and redundant bits. To avoid the situation where the cell groups for redundant storage have many more modified bits than the cell groups for data storage, an extra bit swap function (BS) between regular data bits and redundant bits (e.g., ECC) is added to disperse the redundant bits (that have a high flip rate) among the data bits. By swapping some regular data bits with redundant bits, redundant bits will not be clustered in a single cell group to become a write bottleneck.

For example, if ECC is used with a capacity ratio of 8:1 between data bits and ECC bits, then for PCM with 256-byte line size and 256-bit ECC, one bit from every data byte is picked to swap with ECC bits. To avoid selecting frequently-modified bits with a cyclical pattern, an XOR between $a_5 ... a_7$ and $a_8 ... a_{10}$ is used to determine the bit to be swapped in each data byte.

[2] *Gcc* has many program data updates, which flips many data bits but few ECC bits.

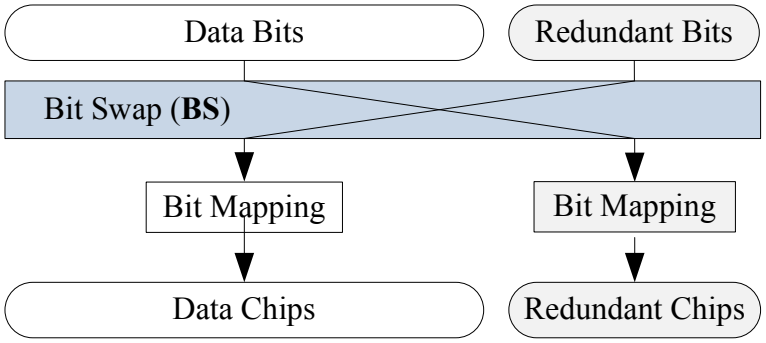


Figure 12: Mapping function for ECC memory swaps between selected data and ECC bits.

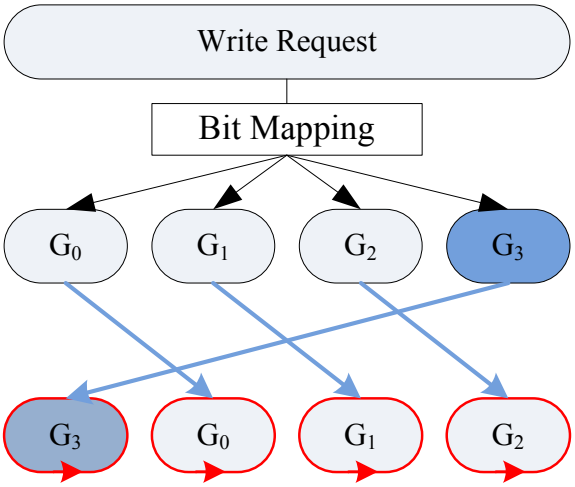


Figure 13: Revised intra-line wear leveling.

3.3.4 Support for Intra-line Wear Leveling

There are two types of wear leveling to evenly distribute memory writes over the entire memory storage: inter-line and intra-line. Inter-line wear leveling balances memory writes among memory lines. Intra-line wear leveling balances memory writes within a memory line.

My bit mapping functions change the bit ordering only within a memory request. Therefore, the functions are not impacted by inter-line wear leveling. For intra-line wear leveling, the conventional row shifting scheme [70] is not compatible with XOR or D-XOR mapping. Figure 13 shows two-level row shifting which is compatible with my mapping functions. First, a global row shift moves data bits at cell group granularity. Second, a local row shift moves data bits inside each cell group. With two-level row shifting, the data bits mapped to a single cell group are still mapped to a single cell group after the row shift. Also, the overhead of two-level row shifting is the same as conventional row shifting.

Intra-line wear leveling only allows for the even distribution of modified bits over the lifetime. For each write request, distribution of modified bits can still be imbalanced with intra-line wear leveling. By contrast, my mapping functions are effective in balancing the distribution of modified bits and reducing write service time for each write request.

3.3.5 Hardware Implementation

As will be shown in Section 3.5, a single mapping function can provide good performance for different number of cell groups. Since a mapping function is a static rearrangement of the data bits in a write request, it has negligible implementation overhead. For PCM writes, a fixed redistribution network is needed to map the data bits to the desired positions before they are sent to PCM. For PCM reads, a corresponding reverse redistribution network can restore data bits to their original positions. Changing the representation of memory data is common for PCM. For example, Zhou et al. [70] proposed a dynamic row shift mechanism to even out the writes to the cells inside a memory line. The delay and power of their 1KB row shifter are 400ps and 795μW. my mapping functions have much lower hardware cost because they statically change only the order of data bits.

The mapping functions do not require changes to the existing memory interface. When data bits are reordered, the memory controller divides the bits into chunks of adjacent bits and delivers

each chunk to a PCM chip in the DIMM. Each chip divides the bits into cell groups of adjacent bits.

The mapping function can be implemented in the memory controller, which exposes all bits, including redundancy bits (e.g., ECC). The mapping function can also be implemented inside the PCM chips themselves, especially for redundant bits that are not visible at the memory controller (i.e., when the chip itself has a remapping or redundant bits).

3.4 EVALUATION ENVIRONMENT

3.4.1 Configuration

I use Virtutech Simics [36] to collect main memory traces, which contain a command identifier (read or write) and address of each memory request. To accurately evaluate PCM write service time, the memory trace file also includes the data before and after memory writes. The trace files are input to a trace-driven simulator. The parameters of my simulator are detailed in Table 1.

I assume an 8-core 2GHz CMP with in-order cores. Each core has a 512KB private L2 cache and a shared 16MB L3 cache (2MB quota per core). After L3 cache, there is a 256MB DRAM cache (32MB quota per core) before the PCM main memory [44]. To alleviate cache miss penalty, a next line sequential prefetcher is incorporated for the DRAM cache.

I model a 128GB PCM main memory with two channels; each channel has two DIMMs and each DIMM has 8 chips and 16 banks. I assume 32 bits per cell group (64 cell groups per 256-byte request) and 2X division program operation (Up to two cells are concurrently programmed per 32-bit cell group). A memory controller configuration similar to Qureshi et al. [45] is used, where each bank has a 32-entry write queue. Read requests are given highest priority, as long as the write queue is not full. For PCM memory scheduling, write pausing [45] is used, whereby the memory controller suspends an active PCM write at the beginning of programming next modified bit to schedule a higher priority read request to the memory bank. I use single-level cell (SLC) PCM to conservatively evaluate the mappings because multi-level cell PCM has much lower write bandwidth than SLC PCM. To model SLC PCM write service time, I use 100ns for cell RESET

CPU	8-core, 2GHz, 2-issue, in-order, 32KB L1 I/D
L2 Cache (private)	512KB, 8-way, 64-byte line size, write-back
L3 Cache	2MB per core, 32-way, 64-byte line size, write-back, 20-cycle local L3 hit
DRAM Cache	256MB total, 32MB per core, 32-way, 256-byte line size, write-back, 60-cycle (30ns) read latency, next line sequential prefetcher
Memory Controller	2 PCM channels 2 DIMMs per channel 16 banks per DIMM 32-entry write-queue per bank write pausing scheduling for PCM
Main Memory	128GB SLC PCM, 120ns read latency 100ns RESET pulse 150ns SET pulse 100ns pulse interval 32 bits per cell group, 2X division program operation

Table 1: System settings.

programming pulse (1→0), 150ns for cell SET programming (0→1) pulse and 100ns interval between two programming pulses [13].

Name	Read PKI	Write PKI	Description
Gems_r	5.35	2.14	8 copies of GemsFDTD
lbm_r	3.14	1.52	8 copies of lbm
leslie_r	3.48	0.79	8 copies of leslie3d
libq_r	8.45	1.50	8 copies of libquantum
mcf_r	11.27	6.42	8 copies of mcf
milc_r	13.42	2.58	8 copies of milc
wrf_r	0.62	0.21	8 copies of wrf
mix_1	1.34	0.37	gcc-mcf-zeusmp-canneal
mix_2	1.13	0.41	astar-gcc-Gems-wrf
mix_3	4.70	0.74	libq-mcf-milc-zeusmp
mix_4	2.94	1.02	Gems-leslie-mcf-zeusmp
mix_5	6.50	1.45	lbm-libq-mcf-canneal
mix_6	4.25	1.50	lbm-leslie-mcf-milc
mix_7	7.02	1.85	Gems-milc-omnetpp-soplex
mix_8	8.93	1.98	libq-mcf-omnetpp-canneal

Table 2: Simulated workloads and their request rates.

3.4.2 Workloads

Since my work addresses the write performance bottleneck for PCM, only write-intensive benchmarks are considered. Twelve write-intensive benchmarks from SPEC CPU2006 and *canneal* from PARSEC are selected. Only *canneal* is selected from PARSEC because most PARSEC benchmarks are computation-intensive or have a very small memory footprint. *Canneal* is executed in single-threaded mode and uses the native input with 940MB memory footprint. For SPEC CPU2006, the reference inputs are used. To evaluate system performance, eight representative multiprogrammed

workloads are selected, each containing two copies of four unique benchmarks; these are the mix_i workloads at the bottom of Table 2. I also ran experiments with seven applications in *rate mode*, where eight instances of the same benchmark are concurrently executed; see the top of Table 2. The other applications are not evaluated in rate mode because the 256MB DRAM cache effectively filters most main memory write requests. All benchmarks are 64-bit binaries, compiled with gcc 4.1.2. Table 2 shows the number of memory read and write requests per 1000 instructions (PKI) after the DRAM cache.

3.5 RESULTS

This section presents simulation results for my mapping functions, with and without redundant bits. I show how performance is improved in comparison to the conventional mapping, both in terms of write service time and IPC, for each benchmark and the average over all benchmarks. Some graphs are normalized, and in those graphs, The geometric mean is used instead of the average to avoid the case in which one benchmark dominates the results.

I evaluated several mapping functions. H*x* (L*x*) uses the high (low) order *x* address bits for bit mapping, where *x* is the number of bits for the cell group index. L*x*^H*x* is an XOR mapping function between the high *x* address bits and the low *x* address bits. L*x*^H*x*^H*y* is a corresponding D-XOR function.

Unless otherwise specified, each write request is mapped to 64 cell groups (M=64, x=6). When M=64, H6 is the baseline, which maps logically adjacent 32 bits to the same cell group.

3.5.1 Mapping for Data Bits

Figure 14 shows the average number of modified bits in the cell group that has the longest programming time for a PCM write; we call this group the *critical cell group*.[3] When modified bits are more evenly distributed among cell groups, the average number of modified bits in the critical cell group decreases, which reduces the write service time. From the figure, most applications do

[3]If there are multiple groups having the longest programming time, a group having the maximum number of modified bits is chosen as the critical cell group.

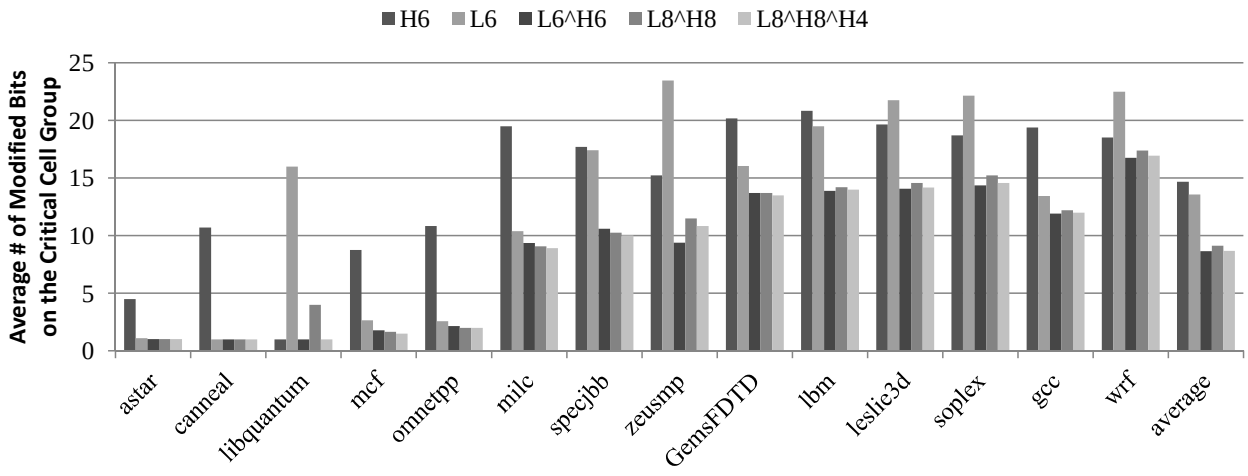


Figure 14: Average number of modified bits on the critical cell group (lower is better).

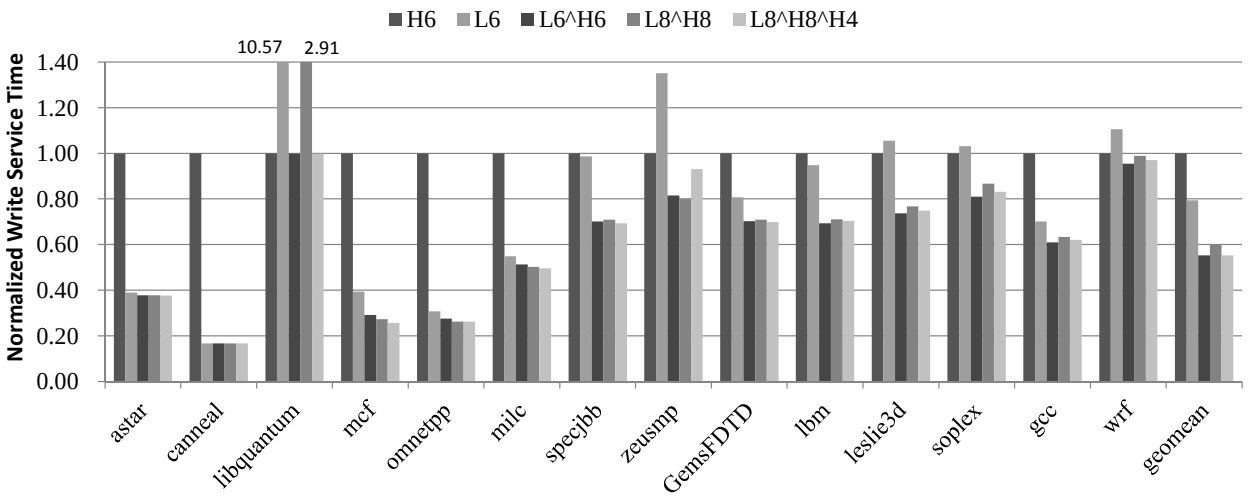


Figure 15: Average write service time of regular data bits normalized to H6.

better with low address bits as the mapping function than with high address bits. This indicates that the distribution of modified bits has a strong cluster pattern. In *libquantum*, *zeusmp*, *leslie3d*, *soplex* and *wrf*, the distribution of modified bits is dominated by a cyclical pattern. For these applications, it is more effective to use high address bits as the mapping function. Overall, L6 has 7.5% fewer modified bits in the critical cell group than the H6 baseline. Because XOR functions combine both low and high address bits, they are much better than H6. L6ˆH6 has 41% fewer modified bits in the critical cell group than the H6 baseline. L6ˆH6 is slightly better than L8ˆH8 because of its effectiveness for *libquantum*. For *canneal*, address bits a_9 and a_{10} are good candidates to use in the mapping function, but these address bits are masked by L8ˆH8 when M=64. The D-XOR mapping function, L8ˆH8ˆH4, has similar results as L6ˆH6. In a later section, the advantage of D-XOR mapping for a variable number of cell groups is shown.

Because divisions are sequentially programmed for each cell group, the number of modified bits in the critical cell group has a direct consequence on PCM write service time. Figure 15 shows the average write service time for each application normalized to H6. L6, L6ˆH6 and L8ˆH8 are much better (79%, 55% and 60%, respectively) than H6. The D-XOR (L8ˆH8ˆH4) service time is similar to L6ˆH6. The reduction in write service time also indicates an improvement in PCM write throughput. Based on these results, I conclude that, on average, L6ˆH6, L8ˆH8 and L8ˆH8ˆH4 can achieve 1.8X, 1.7X and 1.8X more write throughput than H6.

Figure 16 shows the average write service time with different XOR mapping functions (averaged over all 14 benchmarks). All results are normalized to L8ˆH8ˆH4.

As shown in the figure, each XOR mapping function is optimized for a specific number of cell groups. If the number of cell groups is small, some high address bits are masked in the mapping function. For example, with 16 cell groups, the average write service time of L8ˆH8 is 25% worse (higher) than L8ˆH8ˆH4. If the number of cell groups is large, there is unnecessary overlap between high address bits and low address bits, which will also degrade the effectiveness of an XOR mapping function.

Overall, L8ˆH8ˆH4 has similar write service time as an optimized XOR mapping function. However, L8ˆH8ˆH4 is a *single* mapping function that adapts to variable number of cell groups, and thus, it is simpler to implement.

I also compared D-XOR to the most natural shuffling, namely random bit mapping. A random

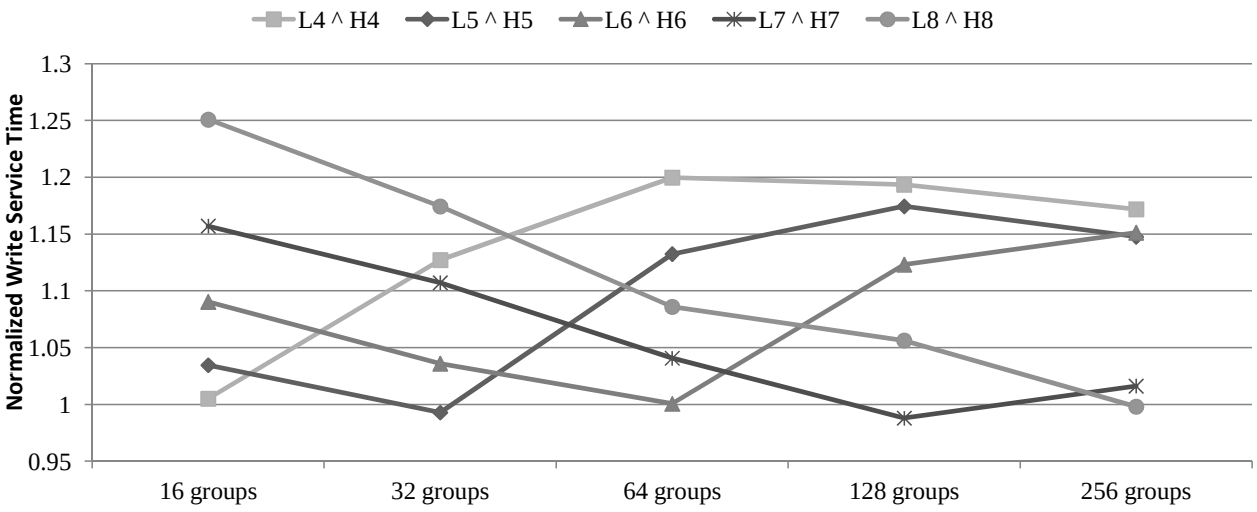


Figure 16: Average write service time of different numbers of cell groups normalized to L8ˆH8ˆH4 (averaged over all 14 benchmarks).

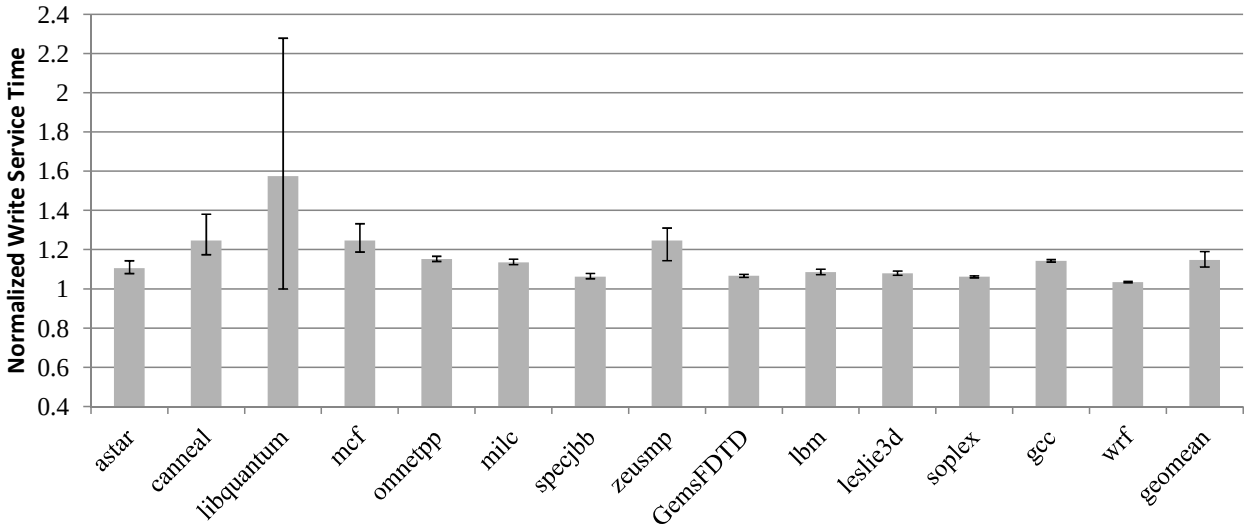


Figure 17: Average write service time of 20 random bit mapping functions normalized to L8ˆH8ˆH4.

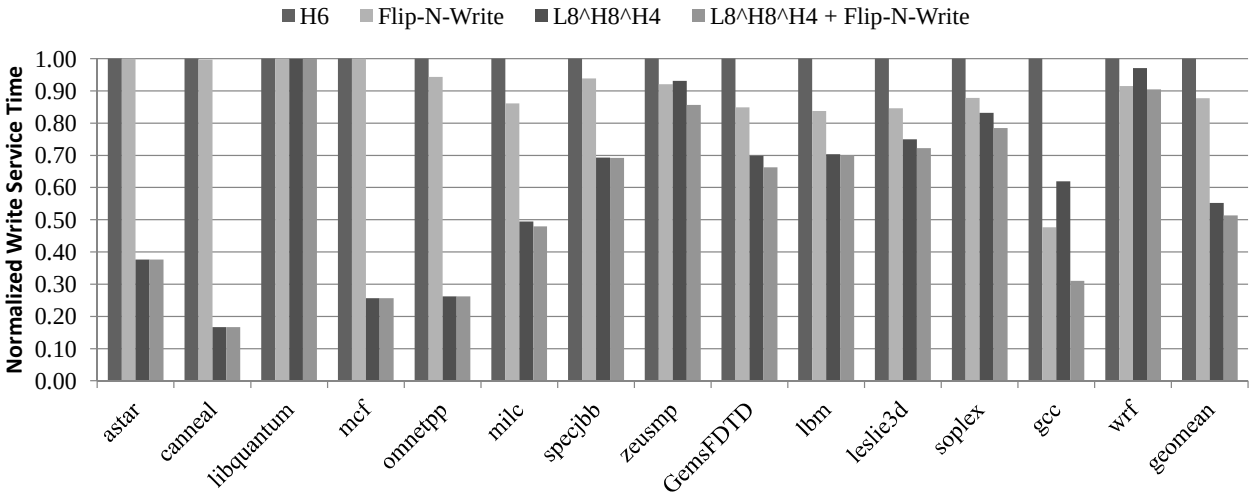


Figure 18: Comparison to Flip-N-Write.

mapping function is a *fixed random permutation* of data bits that eliminates any inherent distribution pattern to avoid clustering the modified bits in one cell group. Figure 17 shows the average write service time for 20 randomly-generated mapping functions. The results are normalized to L8ˆH8ˆH4 D-XOR for comparison. Because random mapping does not utilize cyclical and cluster patterns, the average write service time is 14.7% worse (higher) than L8ˆH8ˆH4. The error bars show the maximum and minimum write service time for different random instances. For most applications, the difference is small. *Libquantum* has a very large variation because the cyclical pattern allows a perfect distribution of modified bits, which cannot be achieved by most random mapping functions.

Adding Flip-N-Write. Flip-N-Write [12] is an effective technique to reduce PCM write service time. With an extra bit of storage, Flip-N-Write counts the number of bits to be written and changes the encoding of data bits to reduce the number of cells that must be programmed. Figure 18 shows the average write service time for Flip-N-Write and the proposed L8ˆH8ˆH4 D-XOR mapping function. Note that Flip-N-Write has an extra bit flip for every 32 data bits in each cell group. All results are normalized to H6. On average, Flip-N-Write and L8ˆH8ˆH4 reduce the average write service time by 12% and 45% respectively. Since both techniques are orthogonal, applying Flip-N-Write to L8ˆH8ˆH4 further reduces the average write service time by 7% over L8ˆH8ˆH4

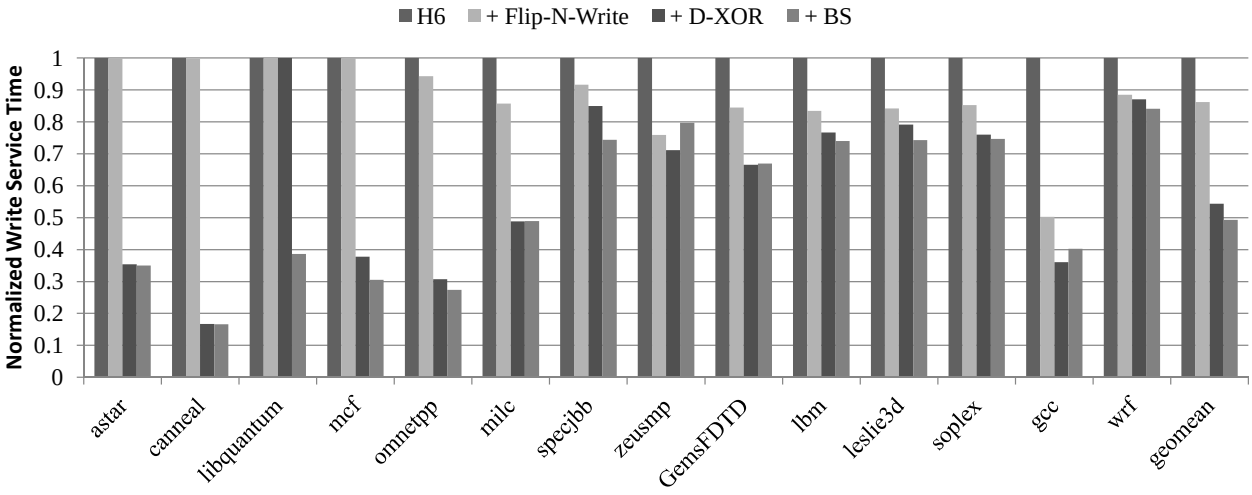


Figure 19: Average write service time for ECC memory normalized to H6.

alone.

3.5.2 Mapping with Redundant Bits

Figure 19 shows the average write service time for PCM with ECC redundant bits. The results are normalized to a DRAM-like ECC baseline with H6. I first apply Flip-N-Write, which reduces the write service time to 84% of H6, on average. Then, I apply D-XOR to both data bits and ECC bits. For 256-byte write requests with 32-byte ECC, the mapping functions for data bits and ECC bits are L8ˆH8ˆH4 and L5ˆH5ˆH2. The write service time is further reduced to 52% of the H6 baseline. Lastly, I apply the bit swap function (BS) to swap all 256 ECC bits with selected data bits. Compared to D-XOR mapping, BS reduces the write service time by 8% on average to 48% of H6. Specifically, given ECC bits as the bottleneck, BS reduces the write service time by 61% for *libquantum*. BS is effective only if the average flip rate of ECC bits is much higher than the flip rate of the data bits. In *gcc* and *zeusmp*, BS degrades write performance as a result. It may be possible to adaptively disable BS in this situation.

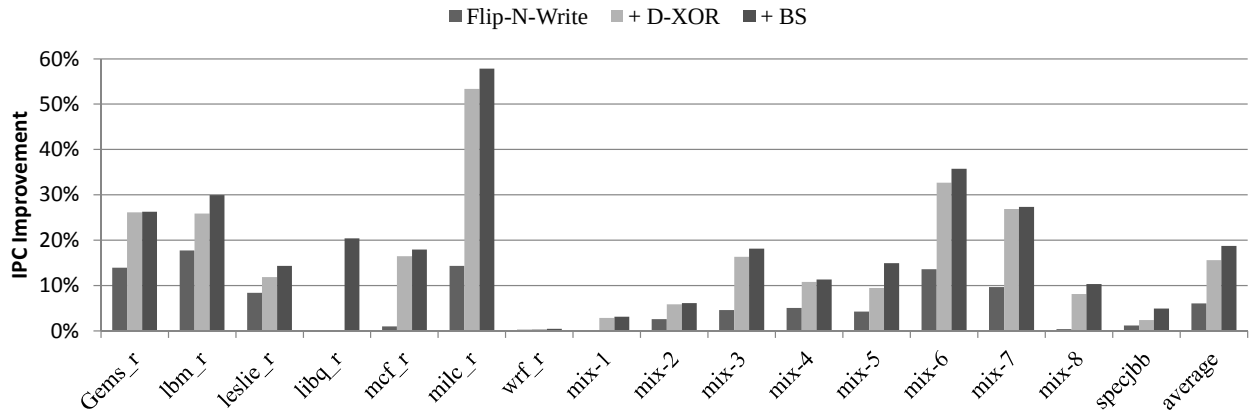


Figure 20: IPC improvement relative to H6 for ECC PCM with 64 cell groups.

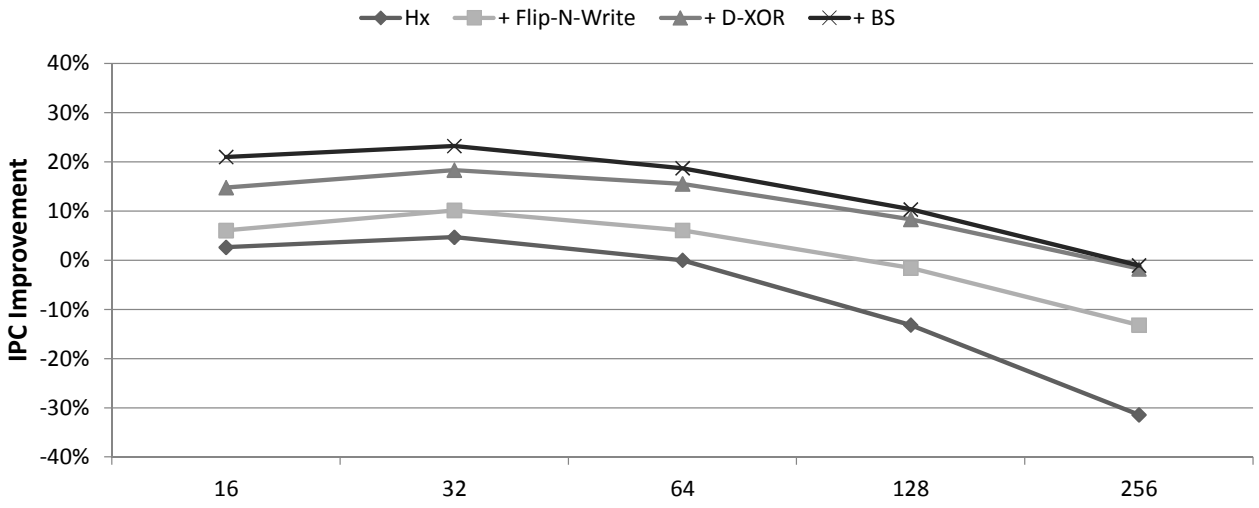


Figure 21: IPC improvement as the number of cell groups is varied (baseline is H6 with 64 cell groups).

3.5.3 Performance

Figure 20 shows IPC improvement over the H6 baseline. The graph shows the improvement for Flip-N-Write and my bit mapping functions for ECC PCM with 64 cell groups.

The results show that Flip-N-Write provides 6.1% performance improvement on average. Adding D-XOR mapping, the improvement increases to 15.6%. *libquantum* has a perfect stride pattern. Hence, D-XOR cannot improve over the baseline. BS provides an additional 3.1% improvement on average. Because *libquantum* has a few modified data bits, BS is particularly effective for *libq_r*, which has a 20% improvement in IPC. Combining D-XOR and BS achieves an extra 11.9% IPC improvement on average over Flip-N-Write. The IPC improvement is sensitive to PCM write traffic load. Workloads with a high WPKI tend to have higher performance improvement. *Wrf* and *mix-1* have very small improvement because their write traffic is limited.

Figure 21 shows the average IPC improvement with different number of cell groups. The maximum number of concurrently activated cell groups is kept fixed. If a memory request is mapped to more cell groups, fewer memory requests can be concurrently processed. D-XOR is consistently better than the H6 baseline and Flip-N-Write. BS provides an additional small improvement, particularly when the number of cell groups is small. The figure also shows that careful selection of the number of cell groups is needed to achieve the best performance. If the number of cell groups is too large, performance drops because fewer memory requests can be concurrently processed. If the number of cell groups is too small, the write service time becomes longer and the memory subsystem suffers a performance penalty from burst writes.

From the results in this section, we conclude that D-XOR mapping with BS and Flip-N-Write should be applied to achieve the best performance for write intensive workloads.

3.5.4 Intra-line Wear Leveling

To quantify the effectiveness of intra-line wear leveling, the number of writes on each bit position is measured. The hottest bit position has the maximum number of writes. Perfect inter-line wear leveling is assumed by simulating all memory write requests to a single memory line. The row shift offset is changed by one byte after every 256 writes [70].

Figure 22 shows the number of writes normalized to the scheme without intra-line wear lev-

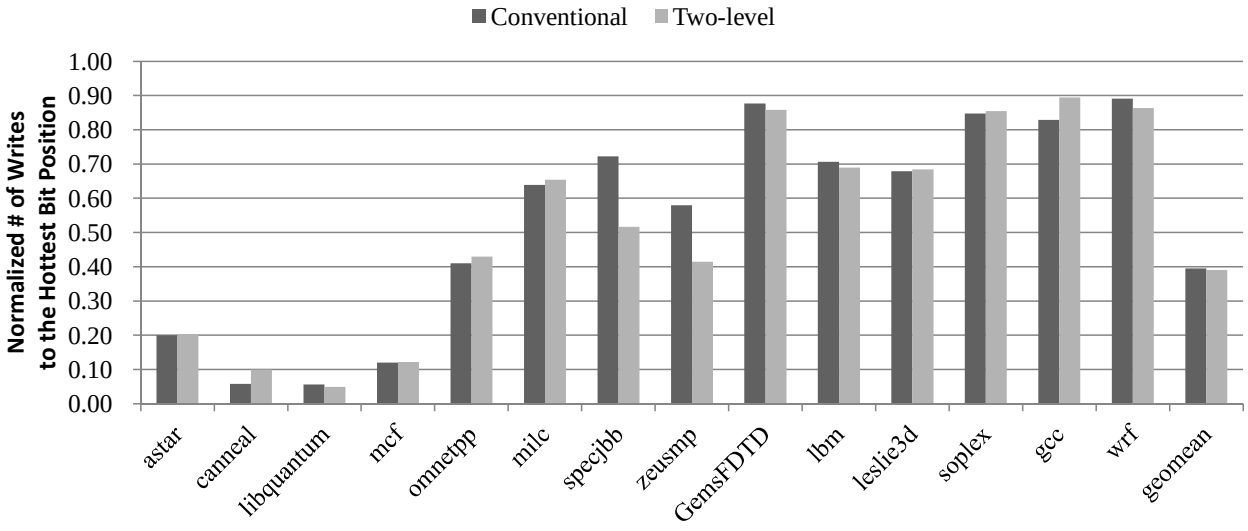


Figure 22: Comparison between conventional row shifting and two-level row shifting.

eling. The figure shows that two-level row shifting is as effective as conventional row shifting on reducing the number of writes to the hottest bit position.

3.5.5 Impact of Division Program Width

Figure 23 shows the average write service time for the proposed mapping function with different division program widths (averaged over all 14 benchmarks). The mapping function used in the evaluation is D-XOR + BS. All results assume that Flip-N-Write is enabled; the results are normalized to H6 with 1X division program width. The division program width is the maximum number of cells that can be programmed in one cell group. The figure shows that the proposed mapping function consistently reduces write service time for different widths. However, the potential benefit decreases (on average) as the width is increased. As shown in Figure 19, *gcc* and *zeusmp* have a worse write service time with BS.

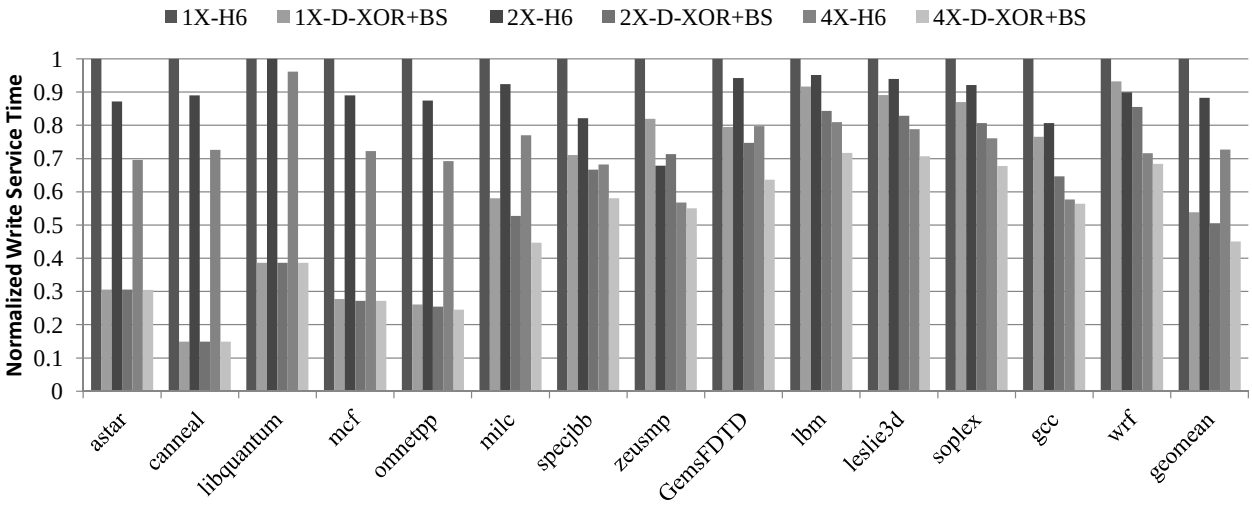


Figure 23: Average write service time for different division program widths normalized to H6 with 1X division program width (over all 14 benchmarks).

3.5.6 Impact of SET to RESET Ratio

In the experiments above, the SET pulse is 150ns and the RESET pulse is 100ns [13]. I analyzed the sensitivity of the proposed mapping functions to the SET:RESET latency ratio.

Table 3 shows the reduction of write service time with Flip-N-Write (one flip bit per 32 bits) and Bit Mapping (D-XOR + BS) for different SET latencies (RESET = 100ns). The proposed bit mapping technique is insensitive to the SET:RESET ratio. When SET:RESET ratio becomes larger, the improvement becomes slightly smaller because write service time is gradually dominated by SET programming time.

SET-to-RESET Latency	1.5:1	2:1	4:1	8:1
+Flip-N-Write	13.8%	13.3%	11.6%	10.3%
+Bit Mapping	50.7%	50.3%	49.2%	48.2%

Table 3: Reduction of write service time for different SET-to-RESET ratios.

3.6 CONCLUSION

The mapping between data bits and PCM cell groups has a significant impact on PCM write service time. I first uncovered stride patterns and spatial locality in the distribution of modified bits in memory write requests. Based on observations about how bit writes are distributed in a write request, I showed that a balanced distribution of modified bits can be achieved by XOR mapping, catering to both stride and locality. I also proposed a double XOR (D-XOR) mapping, which allows a single mapping function to be used for different PCM device programming widths. Finally, I extended my technique to PCM with ECC.

My results show that D-XOR mapping can reduce PCM write service time by 45% on average, which results in a 1.8 times improvement in write throughput. My best mapping function achieves an average 11.9% IPC improvement over Flip-N-Write for ECC-protected PCM. Data bit mapping is a simple and effective mechanism to increase PCM write throughput and program IPC.

4.0 DELTA-COMPRESSED DRAM CACHING FOR HYBRID MEMORY SYSTEMS

4.1 PROBLEM STATEMENT

To avoid performance bottlenecks caused by the limited write performance of non-volatile memory, DRAM can be used as a cache to offload memory traffic to non-volatile memory for both *performance-critical data* and *frequently modified data*. To cache more data in DRAM in hybrid memory systems, compression can be used to increase the effective capacity of the DRAM cache.

Figure 24 shows an organization which supports compressed DRAM caching in hybrid memory. The cache's capacity is divided into *uncompressed* and *compressed* regions. The amount of capacity allocated to each region can be dynamically changed. The uncompressed region has low access latency (it does not require decompression and compression), and thus, it is used to cache performance-critical data. The compressed region has higher access latency than the uncompressed region; it caches only frequently modified data that are not in the uncompressed region. Non-modified data are not cached in the compressed region because the potential gain to read data from the compressed DRAM region instead of PCM is limited due to the latency overhead to locate and decompress the data.

In a DRAM-only system, the design goal of memory compression is to reduce expensive disk accesses. In hybrid memory, however, the design goal of DRAM compression is to reduce the PCM write traffic. The problem is to find a new DRAM compression scheme which is optimized for hybrid memory systems to further improve the compression ratio and reduce the PCM write traffic.

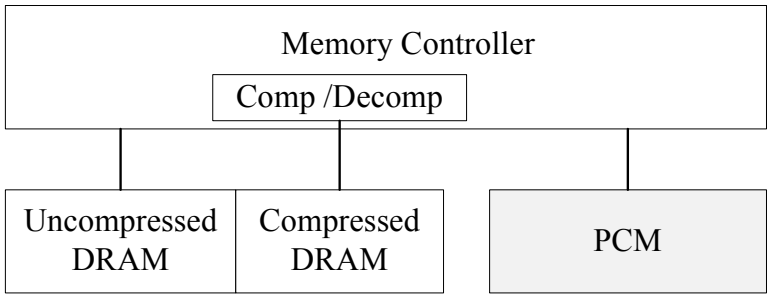


Figure 24: Compressed DRAM caching in hybrid memory.

4.2 PROPOSED SOLUTION: DELTA-COMPRESSED DRAM CACHING

4.2.1 Compressed DRAM Caching

To support compressed DRAM caching, three components are introduced to control the DRAM cache. A *page classification* module identifies memory pages that are suitable for the uncompressed DRAM region and the compressed DRAM region. A *partition adjustment* module determines how much DRAM capacity is used by each region. A *compression and decompression* module accesses data in the compressed region.

4.2.1.1 Page Classification Page classification is used to determine whether data should be kept uncompressed (frequently accessed), compressed (frequently modified) or uncached (infrequently accessed). The monitoring is done at page granularity instead of line granularity to amortize the storage overhead. Recent techniques for PCM data placement use variations of a Multi-Queue (MQ) algorithm to identify hot pages to cache in DRAM [69, 50]. In a MQ algorithm, a memory page's access frequency is monitored. Pages are placed into a queue based on their access frequency. Queues are classified into *hot* and *cool*, and pages in the hot queues are cached in DRAM.

In the proposed technique, the DRAM cache is partitioned into uncompressed and compressed regions. Therefore, an additional classification category is introduced to the MQ algorithm: *warm queues*. Using this classification, pages in hot queues are cached in the uncompressed region, pages

in warm queues are cached in the compressed region, and pages in cool queues are not cached in DRAM. Because the compressed region is suitable for storing frequently-modified pages, page write frequency should be used to identify warm pages, while page access frequency should be used to identify hot pages. However, this increases hardware cost to monitor multiple metrics; In this work, only page access frequency is used to determine page type.

4.2.1.2 DRAM Partition Adjustment When data is cached in the compressed region, its access latency inevitably increases due to compression and decompression operations and extra memory accesses to the metadata. Therefore, it is beneficial to control the capacity of the uncompressed and compressed cache regions. For workloads with low PCM write traffic, the capacity allocated to the uncompressed region should be increased to get the benefit from low access latency. When PCM write traffic is high, the compressed region's size should be increased to cache more modified data in DRAM (avoiding writebacks and hitting the "write bandwidth wall" of the PCM devices). The capacity allocated to the uncompressed and compressed regions are parameters that are typically set at system boot-up (e.g., set in the BIOS). The trade-off to determine how much capacity to allocate to each region is investigated in the experimental results. The allocation decision could be made online, according to monitored workload behaviour (the write throughput to the non-volatile memory and the compression ratio of the data). This work only addresses the detailed design of incorporating compressed DRAM cache in hybrid memory.

4.2.1.3 Selective and Predictive Compression Any memory compression scheme can be used for the compressed region of the DRAM cache. In my scheme, I choose to use conventional and delta-compression together. Delta-compression leads to a higher compression ratio and fewer PCM writes, but it may not improve performance because delta-compression can generate extra PCM read traffic (to read the old reference data) for both memory reads and writes. Due to this additional cost, I propose *selective compression*, which sets a threshold on when delta-compression should be enabled. Specifically, for each memory line, the memory controller selectively enables delta-compression only if it leads to a minimum gain in storage (which I call a *Gain Threshold*, GT) over conventional compression. The GT captures a tradeoff between gain in memory capacity due to compression and extra memory traffic to access delta-compressed data. Also, a line is kept

uncompressed if neither compression nor delta-compression can reduce the storage needed for the line.

When writing a memory line, selective compression needs to read the old data in PCM to evaluate whether the line should be delta-compressed. If the line is evaluated as not suitable for delta-compression, the extra PCM read becomes an unnecessary overhead. I propose *predictive compression* to address this problem. Predictive compression relies on the observation that delta-compression is probably not advantageous for a line if delta-compression was not advantageous the last time that the line was written. In other words, the memory controller applies delta-compression to lines that can be consistently delta-compressed. For a line that was not delta-compressed last time it was written, the memory controller typically stores it not delta-compressed to avoid unnecessary PCM read cost. To give the line a chance to recover delta-compressed status, with a small probability (which I call *Recovery Probability*, RP) the memory controller will evaluate whether the line should be delta-compressed.

4.2.2 Delta-compression for Written Data

For compression in a DRAM-only system, the data in a *memory line* should be compressed[1]. A higher compression ratio can be achieved with delta-compression. Delta-compression requires the data having a reference copy. The modification to the reference copy is compressed instead of the data itself. Figure 25 shows how to apply delta-compression in hybrid memory. First, the old data is read from PCM. Then the difference between the new and the old data (called *diff data*) is computed using an XOR operation. Lastly, the diff data are compressed and stored in DRAM. Delta-compression tends to have a higher compression ratio than conventional compression because delta-compression converts unmodified data bits to zeros before compression.

The challenge is reading delta-compressed data (i.e., the *frequently modified data*): it is more time consuming because the decompressed diff data cannot be accessed directly from DRAM. Instead, the old PCM data must be read from main memory (PCM) and XORed with the decompressed diff data. Delta-compression in DRAM provides a way to trade off more PCM reads against reduced PCM writes (caching more modified data in a more compressed format).

[1] A memory line is defined as the minimum unit for memory compression.

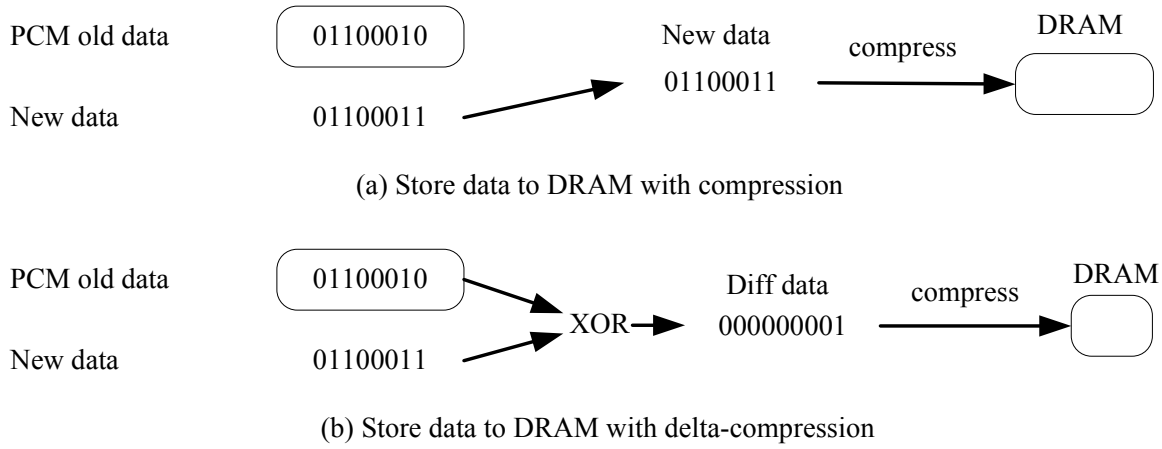


Figure 25: An example to illustrate delta-compression.

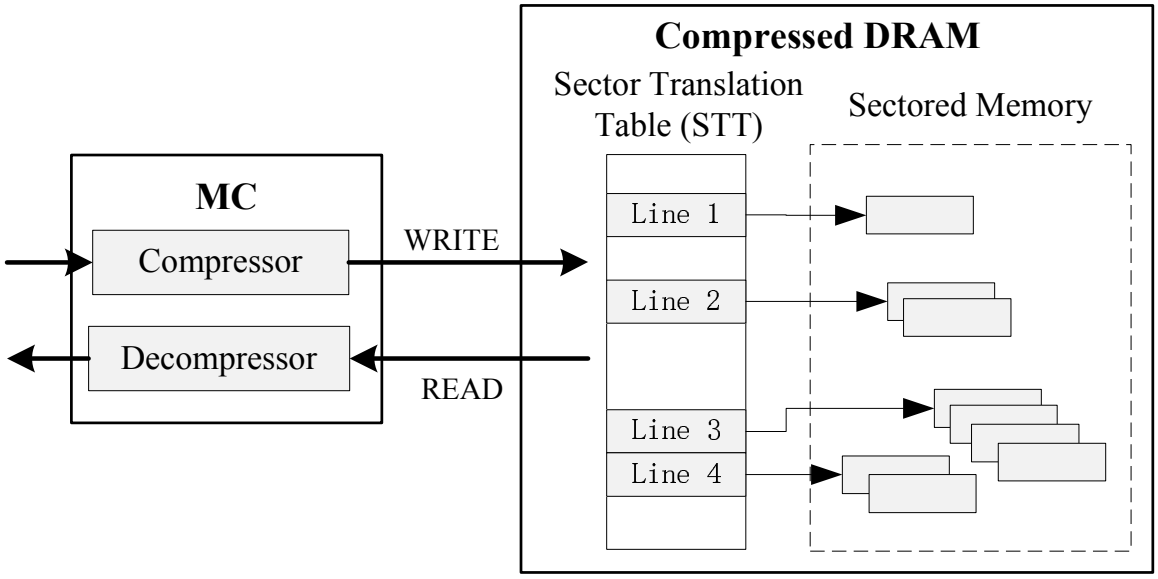
In summary, for delta-compression in hybrid memory, frequently modified data are compressed and partially stored in both DRAM and PCM. The memory controller splits and combines data to/from DRAM and PCM to provide correct data. Given this unique requirement on memory compression in hybrid memory, I designed a compression scheme specially tailored to the DRAM cache in hybrid memory, as described next.

4.2.3 IBM MXT Compression for DRAM-only Systems

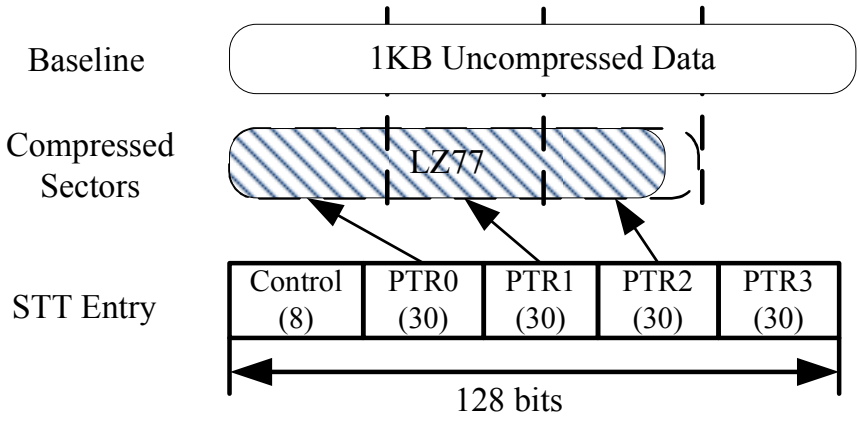
Since my work is based on IBM MXT, I will make a brief introduction to IBM compression framework. IBM MXT [62] is a high performance hardware-based memory compression scheme for DRAM-only systems. Figure 26(a) shows MXT's architecture; 4KB pages are stored in DRAM in a compressed form. On every memory access, the memory controller (MC) compresses or decompresses the data. MXT offers a framework for my D-COMP algorithm and its variants. Other compression frameworks could also be used.

MXT partitions DRAM into two regions: the sector translation table (STT) and the sectored memory[2]. STT contains the metadata to locate compressed lines. Sectored memory is a collection of 256-byte memory sectors that hold compressed data. Each 4KB page is divided into 4 *memory*

[2]This is a common design for a memory compression scheme.



(a) MXT architecture



(b) an example of a line compressed to three sectors

Figure 26: IBM MXT compression.

lines. Figure 26(b) shows how each memory line is compressed in MXT. A 16-byte STT entry is associated with each 1KB memory line and contains up to 4 pointers to memory sectors. The memory controller uses the pointers in the STT entry to access the data. The control field of the STT entry contains a line state to indicate whether the line is Uncompressed (U), Compressed (C) or Invalid (I). If a line is Uncompressed, the STT entry uses all pointers to 4 memory sectors. If a line is Compressed, the STT entry may have fewer than 4 valid pointers to compressed memory sectors. The control field also contains the compressed size of the line. To avoid repeated accesses to STT entries, a small dedicated on-chip storage in the memory controller (*STT Cache*) contains recently accessed STT entries.

4.2.4 Implementation of Delta-compressed DRAM Caching

Delta-compression is not suitable for traditional memory compression because expensive disk read operations are needed to restore delta-compressed data using the reference data on the disk. Delta-compression is possible for hybrid memory because the reference data is accessed from PCM which is slightly slower than DRAM.

In this section, I choose to use a DRAM-only compression scheme, namely MXT [62], as my baseline and extend it to support delta-compression for hybrid memory. This approach allows me to focus my efforts on enabling the unique delta-compression without reinventing the non-trivial mechanisms that are germane to any memory compression scheme. My proposed techniques can be tailored to other compression schemes that are used for hybrid memory systems.

4.2.4.1 Hierarchical Compression Metadata The most important change to MXT for hybrid memory involves the compression algorithm. MXT uses a parallel derivative [24] of the Ziv-Lempel (LZ77) algorithm. Because LZ77 is a dictionary algorithm, the memory line size is 1KB to achieve a good compression ratio. The whole 1KB memory is stored in DRAM. It takes 128 CPU cycles for MXT to decompress a memory line.

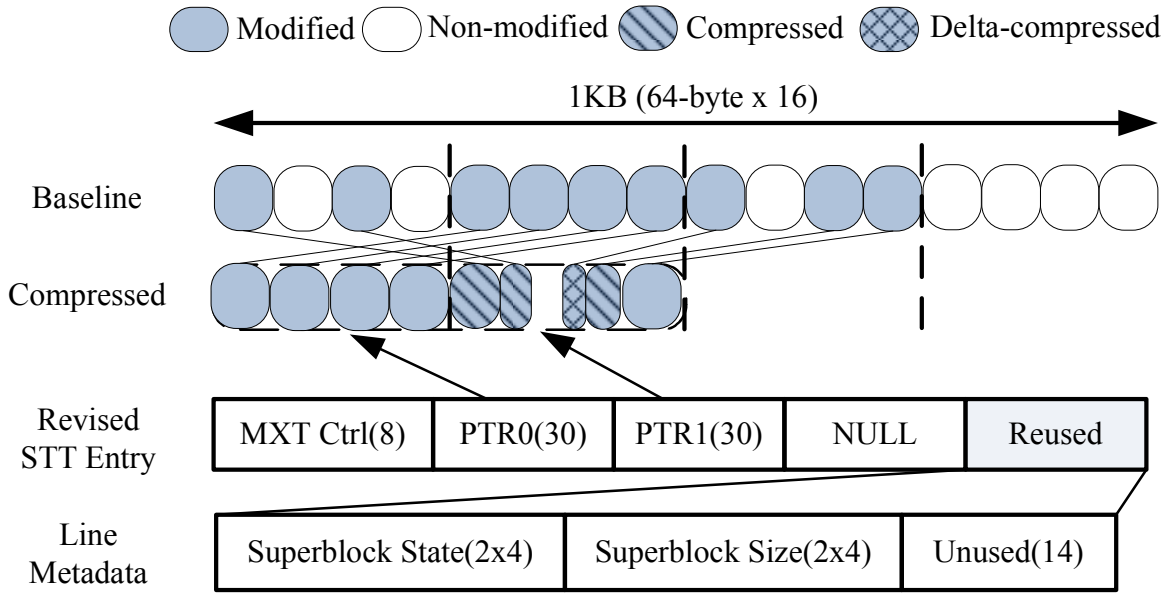
In contrast, my new approach for hybrid memory needs to cache only modified data in the compressed region of DRAM. Therefore, the 1KB line is divided into sixteen 64-byte *blocks* to allow a finer granularity of compression (i.e., 64 bytes rather than 1KB). The memory controller

uses *Frequent Pattern Compression* (FPC) [2] to compress memory blocks. FPC is the underlying compression algorithm for both conventional and delta-compression. FPC compresses data on a word-by-word basis by storing common patterns in a compressed format accompanied by an appropriate prefix. FPC has a low-cost hardware implementation [2] and takes only 5 CPU cycles to encode/decode a 64-byte block [1]. In my implementation, the sixteen blocks in a memory line can be compressed or decompressed in parallel.

Figure 27(a) shows an example in which nine modified blocks are compressed and stored in DRAM. For each block, both conventional compression and delta-compression are tried. The compression method that achieves the smallest compressed block size is selected. If both schemes give a compressed block size equal to or larger than the uncompressed size, neither scheme is selected and the block is stored in an uncompressed form. Each block has a state to indicate its format. There are four block states: Invalid (I), Uncompressed (U), Compressed (C) and Delta-compressed (D). If a block state is Invalid, data is written and read from PCM. If a block state is Uncompressed or Compressed, data is written and read from DRAM. If a block state is Delta-compressed, then the DRAM stores only the diff data. To read the content of the block, the PCM data is read and XORed with the diff data.

FPC-based compression allows a line to be partially decompressed to access some of its data blocks. To achieve this capability, additional metadata are stored in the STT entry of each compressed line. At a minimum, each 64-byte block requires 5-bit metadata: a 2-bit block state and a 3-bit block size (if data block is aligned at 8-byte boundary). Hence, for all 16 blocks in a line, there is an extra 80-bit metadata in each STT entry versus the original MXT scheme. To avoid actually increasing the STT entry size, the PTR3 field of the STT entry is reused, as described next.

For my compression scheme, a 1KB line is not compressed if compressed data needs more than three memory sectors. Therefore, for compressed lines, the PTR3 field of the STT entry is unused. However, the PTR3 field is only 30 bits, which is not enough to store the 80-bit metadata. To work around the problem, I use a hierarchical design for the metadata. Every 4 blocks in a line are treated as a *superblock*. Each 256-byte superblock requires 4-bit metadata: a 2-bit superblock state and a 2-bit superblock size (superblocks are aligned at 64-byte boundary and the superblock size field is valid only for compressed superblocks). The full 16-bit metadata for 4 superblocks



(a) an example of a compressed line

Superblock	Block
Invalid (I)	I
Uncompressed (U)	U
Compressed (C)	U, C
Delta-compressed (D)	I, U, C, D

(b) Block and superblock states

Figure 27: FPC-based delta-compression algorithm.

Superblock	State	Size
0	C	128 (80)
1	U	256 (256)
2	D	128 (114)
3	I	0

Block	State	Size
0	D	18
1	I	0
2	C	31
3	U	64

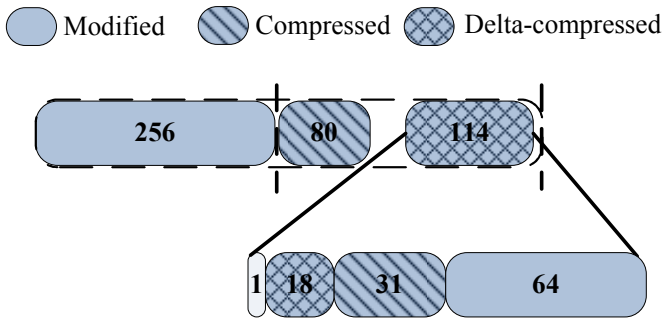


Figure 28: Data layout of an example memory line.

can be stored in the PTR3 field.

The *superblock state* is a summary of the states of the associated blocks (see Figure 27(b)). For an uncompressed (U) superblock, all of its blocks are uncompressed and there is no need to store block states. Similarly, for an invalid (I) superblock, all of its blocks are invalid and the block states are unnecessary. If all blocks in a superblock are stored in DRAM and at least one block is compressed, the state of the superblock is compressed (C). If the state of a superblock is not U, I and C, the state of the superblock is delta-compressed (D). For compressed or delta-compressed superblocks, the states of constituent blocks are stored along with the compressed data (described in the next section). By keeping superblock states rather than block states, we trade off the granularity of state information against the storage overhead of the metadata.

4.2.4.2 Compressed Data Layout In MXT, compressed data are stored contiguously in the logically linear storage implemented by the pointers in an STT entry. To avoid accessing two memory sectors for each uncompressed superblock, a revised compressed data layout in the logi-

cally linear storage is used. In the layout, uncompressed superblocks are placed before compressed superblocks. By doing so, each uncompressed superblock can be aligned to a 256-byte boundary and stored in a dedicated sector[3]. Compressed superblocks are sequentially arranged after uncompressed ones. Because my design uses a 64-byte read size, a compressed superblock is aligned to a 64-byte boundary to avoid extra memory reads for crossing a 64-byte boundary.

Figure 28 shows the compressed data layout of an example memory line. Uncompressed superblock 1 is placed at the beginning of the logically linear storage, followed by two compressed superblocks. There is a gap between the two compressed superblocks because superblocks are aligned to a 64-byte boundary. The example also shows the detailed layout of superblock 2. Since this block is delta-compressed, it has a block state header before the compressed data. The 1-byte header contains the block states of the superblock. The data of the blocks are compressed by the FPC algorithm. The size of each compressed block is indicated by the metadata inside the FPC-compressed data.

4.2.4.3 Memory Read To read data from the cache, the memory controller needs to determine the location of the data. Figure 29(a) shows how to determine the data location from the line state and superblock state. If the line state is invalid or uncompressed (first two rows of Figure 29(a)), the data will be read from PCM and DRAM only, respectively. If the line state is compressed, the location of data depends on the superblock state as follows. If a superblock X is invalid (I), the data will be read from PCM. If a superblock X is uncompressed (U) or compressed (C), the data will be read from DRAM. If superblock X is delta-compressed (D), the data will be read from both DRAM and PCM. The data from the DRAM and PCM also need to be merged to get actual memory data. The merge operation is performed at block granularity based on the state of each block. Figure 29(b) shows the merging rules. For example, if a block is delta-compressed, the data from the DRAM is decompressed and XORed with the data from the PCM.

To read PCM data, the address is the same as the address of the memory request. To read DRAM data, the memory controller needs to calculate the start position of the superblock from the superblock state and superblock size in the STT entry.

If superblock state is U, its relative position is the sum of the sizes of the uncompressed su-

[3]Recall that the logically linear storage space is represented by multiple 256-byte sectors.

Line State	Superblock State	DRAM	PCM
I	-	No	Yes
U	-	Yes	No
C	I	No	Yes
C	U	Yes	No
C	C	Yes	No
C	D	Yes	Yes

(a) Determine data location

Block State	Action on Data
Invalid (I)	Read PCM
Uncompressed (U)	Read DRAM
Compressed (C)	Read Decompressed DRAM
Delta-compressed (D)	Read Decompressed DRAM XORed with PCM

(b) Merge block data

Figure 29: Rules for reading data from compressed hybrid memory.

perblocks stored before it, that is

$$StartPos(X) = \sum_{j=1}^{X-1} Uncomp(j) \cdot Size(j) \tag{4.1}$$

If a superblock state is C or D, its relative position is the sum of the sizes of all superblocks stored before it and all other uncompressed superblocks, that is

$$StartPos(X) = \sum_{j=1}^{X-1} Size(j) + \sum_{j=X+1}^{4} Uncomp(j) \cdot Size(j) \tag{4.2}$$

To implement this calculation, simple logic is needed to conditionally sum the size of four superblocks.

4.2.4.4 Memory Write To store a compressed superblock, the memory controller first evaluates the different sizes of the data when the superblock is stored as uncompressed (U), compressed (C) or delta-compressed (D). To calculate the size of a delta-compressed superblock, the current PCM data of the superblock are read. After the evaluation, the memory controller chooses the superblock state that can achieve the smallest compressed size. If the storage size of the superblock is unchanged (aligned on a 64-byte boundary), then the new data will be directly written to DRAM. If the storage size of the superblock is changed (overflow/underflow), which is a common condition for compressed memory, the memory controller will read from DRAM all superblocks stored

after the requested superblock. The memory controller writes all affected superblocks to their new DRAM location. I evaluated this extra data movement cost in my experiments. With a write buffer, the data movement operation is not on the critical path, and accounts for only a small portion of total DRAM memory traffic. Similarly to reading PCM, writing PCM is straightforward. The data is directly written to PCM using the address from the write request. The memory controller needs to write data to PCM in two cases: a memory line is written but not cached in DRAM or a memory line is evicted from DRAM.

4.2.4.5 Cache Replacement Policy For MXT, unused memory sectors are organized as a linked list. When extra memory storage is needed, unused sectors are allocated from the list by a small hardware circuit [62]. The memory controller tries to maintain a minimum quota of unused DRAM sectors (16MB in the experiments). Once the quota is below this threshold, the memory controller walks the STT to evict rarely accessed pages.

The proposed design uses a MQ algorithm to monitor page access frequency. Recall that hot pages are placed in the uncompressed region, only warm pages are candidates for cache replacement in the compressed region. The memory controller tracks the number of pages in each warm queue and divides warm queues into high-rank queues and low-rank queues. A high-rank warm page is guaranteed to be cached in the compressed region. A low-rank warm page with low access frequency is evicted first. To avoid thrashing, modified data of a low-rank warm page is inserted into the compressed region with a throttled insertion policy [46].

4.2.4.6 Hardware Cost and Overhead The hardware cost to support compressed DRAM depends on the underlying compression algorithm and page classification mechanism. Delta-compression by itself only requires implementing a tailored delta-compression algorithm for both memory reads and writes in the memory controller. I use FPC as the building block for delta-compression. FPC is easily implemented in hardware and is used for on-chip cache compression [1].

For MXT, the STT is the major memory data structure. The size of the table can be dynamically adjusted based on the size of the compressed region. For every 2GB of compressed capacity, 32MB DRAM is needed for the STT. I assume a 16KB STT cache to hold the 256 most recently accessed

STT entries. I use Rank-based Page Placement (RaPP) [50], a hardware-assisted variation of MQ, for page classification. RaPP requires 126KB on-chip storage and 12MB DRAM to maintain data structures for the MQ algorithm for a 2GB DRAM and 16GB PCM hybrid memory.

When data is stored in a compressed form, its access latency is increased. There are two reasons for the increase: STT translation delay and decompression latency. I assume 5 CPU cycles for STT translation delay, for a hit in the STT cache. I assume 15 CPU cycles to decompress a 256-byte superblock and merge the data from both DRAM and PCM if the data is delta-compressed. I also model the extra memory delay due to STT misses. For uncompressed data, the memory latency is the time to read the critical 64-byte block. For compressed data, all compressed blocks of a superblock are read before decompressing the superblock.

Because FPC is simple, an on-chip FPC compression and decompression engine has low power overhead. The power of the implementation is estimated to be 0.28W [15]. I add 0.3W for the compression and decompression logic. I assume 0.3W for STT logic and 0.3W for buffering.

4.3 EVALUATION ENVIRONMENT

4.3.1 Configuration

I use Virtutech Simics [36] to collect memory traces. To evaluate memory compression, I use a trace-driven simulator that takes traces as input files with the command and address of each memory request, and the data before and after every memory write.

I model an 8-core 2GHz CMP with in-order cores and a cache hierarchy similar to the IBM Power7 [56] with a 32MB L3 cache. Since multiprogrammed workloads are executed with one program per core, I assume that each core only uses its local 4MB L3 cache region. To alleviate the miss penalty, I also model a simple sequential data prefetcher for the L3 cache.

I model a hybrid memory system with 2GB DRAM and 16GB PCM. Similar to [50], a small DRAM size is used to match the memory footprint of the workloads. The hybrid memory has four 12.8GB/s memory channels. Based on the measured bandwidth requirements, DRAM and PCM have two dedicated memory channels, each with two DIMMs, and each DIMM with eight

banks. A memory controller configuration similar to [45] is used, where each bank has a separate 32-entry read queue and a 32-entry write queue. Read requests are given higher priority, as long as the write queue is not full. For PCM memory scheduling, Write Pausing [45] is used: a PCM write is divided into multiple 50ns epochs and the memory controller can suspend an active PCM write at the beginning of an epoch to schedule a read request to the memory bank. In the baseline, I assume 0.75GB/s PCM write bandwidth per channel, which is equivalent to PCM write service time of 2600 cycles per write.

I calculate memory power as in [16], adding background power and operation power. Background power is determined by memory type, capacity and power state. Operation power is assumed to be proportional to memory bandwidth. Memory power is estimated using the parameters in [11]. The simulation parameters are summarized in Table 4. My experiments confirmed the results in [48] that a hybrid memory can have better performance and energy than a DRAM-only system due to the increased capacity enabled by better cell density from PCM.

4.3.2 Workloads

Because my work focuses on memory, I use only memory-intensive benchmarks that have large memory footprints from the SPEC CPU2006 [14] and PARSEC [9]. All benchmarks are 64-bit binaries, compiled with gcc 4.1.2. Most PARSEC benchmarks are computation intensive or have a very small memory footprint. For SPEC CPU2006, the reference inputs is used. The memory footprints of benchmarks are scaled when it is possible to change the input parameters to avoid results that are skewed by *mcf*, which has a 1.6GB memory footprint. Table 5 gives the detailed parameters of the scaling. For *canneal* from PARSEC, I run it in single-threaded mode and use native input with 940MB memory footprint.

Table 6 shows the memory footprint (size, in GB), the number of memory read requests per 1000 instructions (Read PKI) and the number of memory write requests per 1000 instructions (Write PKI) of each workload. Ten representative multiprogrammed workloads are selected, each containing two copies of four unique benchmarks. Four single applications are selected and run in a rate mode, where eight instances of the same benchmark are concurrently executed. The simulator requires a long warm-up phase to populate the large 2GB DRAM cache. The simulation

CPU	8-core, 2GHz, 2-issue, in-order, 32KB L1 I/D
L2 Cache	512KB private, 8-way, 64-byte linesize, write-back.
L3 Cache	32MB total, 4MB per core, 256-byte linesize, 20-cycle local L3 hit, write-back, sequential data prefetcher
Memory Controller	2 PCM channels + 2 DRAM channels, 12.8GB/s per channel, 2 DIMMs per channel, 8 banks per DIMM, 32-entry read- and write-queue per bank, read priority scheduling for DRAM, write pausing scheduling for PCM, FPC decoding latency 15 cycles STT translation latency 5 cycles
DRAM	2GB, 50 ns (100 cycles) for first 64-byte read, 15 ns (30 cycles) for each consecutive 64-byte read. Background Power: 0.93W/GB (leakage + refresh) Read Energy: 0.8J/GB, Write Energy: 1.2J/GB
PCM	16GB, 80 ns (160 cycles) for first 64-byte read, 15 ns (30 cycles) for each consecutive 64-byte read, 0.75GB/s write bandwidth per channel, (2600-cycle write service time per 64-byte write), 50ns epoch, 26 epochs per PCM write Background Power: 0.10W/GB (leakage) Read Energy: 1J/GB, Write Energy: 6J/GB

Table 4: System settings.

Name	Ref Problem Size	New Problem Size
libquantum	1397	5953 (1GB)
milc	20x20x20x20	26x26x26x26 (1.5GB)
leslie3d	121x121x15	331x331x35 (1.2GB)

Table 5: Benchmarks with scaled problem sizes.

is switched from the warm-up phase to the timing phase after 160 million write references are simulated or after one benchmark completes 30 million write references. The simulation stops when one of the benchmarks completes 60 million write references.

4.4 RESULTS

The baseline for my evaluation is a hybrid memory system using *Rank-based Page Placement* [50] without compression (RaPP-RW). RaPP-RW uses a tailored MQ algorithm to identify and store *frequently-accessed* pages in DRAM. I also evaluated a variation of the MQ algorithm (RaPP-WO), which only identifies and stores *frequently-modified* pages in DRAM.

I compare the baseline with four schemes that use compression: COMP applies FPC compression only on the written data; D-COMP is a delta-compression scheme that applies FPC on the difference between the new and old data; SD-COMP applies selective compression and PSD-COMP adds predictive compression to SD-COMP (both described in Section 4.2.1.3). Unless explicitly specified, COMP, D-COMP, SD-COMP and PSD-COMP all have a 1.5GB compressed region and a 0.5GB uncompressed region.

For SD-COMP and PSD-COMP, I performed a sensitivity study on Gain Threshold (GT, defined in Section 4.2.1.3). If GT is small, more lines are delta-compressed, and vice versa. I find optimal performance results when GT is between 48 bytes and 96 bytes. In the evaluations, I choose 64 bytes as the value of GT. For PSD-COMP, I also did a sensitivity study on the value of

Name	Size (GB)	Read PKI	Write PKI	Description
Gems_r	6.3	5.34	2.40	8 copies of GemsFDTD
leslie_r	9.2	3.71	1.12	8 copies of leslie3d
mcf_r	12.8	15.10	7.24	8 copies of mcf
milc_r	11.8	6.18	1.84	8 copies of milc
mix_1	8.3	8.10	4.09	lbm-libq-mcf-milc
mix_2	8.2	5.43	1.66	libq-mcf-milc-zeusmp
mix_3	7.9	6.60	3.39	lbm-leslie-libq-mcf
mix_4	7.7	8.52	4.57	lbm-libq-mcf-canneal
mix_5	5.7	8.59	3.27	gcc-mcf-zeusmp-canneal
mix_6	5.4	5.14	1.98	leslie-omnetpp-zeusmp-canneal
mix_7	5.2	5.65	3.16	lbm-libq-zeusmp-canneal
mix_8	5.0	13.99	5.97	astar-libq-milc-omentpp
mix_9	3.6	3.91	2.06	astar-gcc-milc-zeusmp
mix_10	2.5	3.23	2.06	astar-gcc-Gems-wrf

Table 6: Simulated workloads.

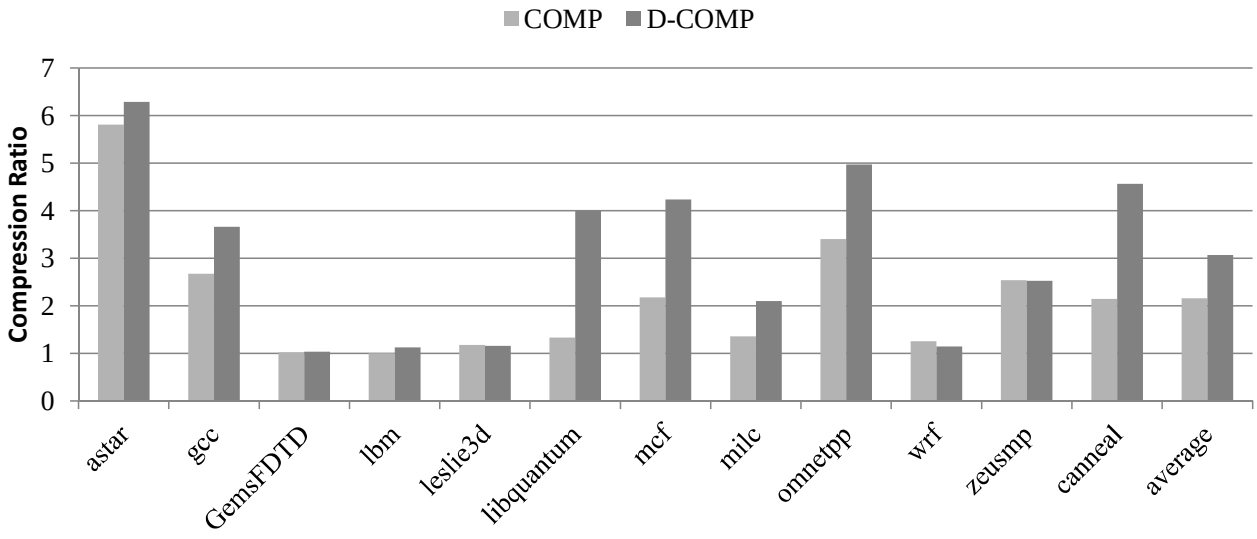


Figure 30: Compression ratios (original size / compressed size) for FPC compression (COMP) and delta-compression (D-COMP)

the recovery probability (RP, defined in Section 4.2.1.3). I found that the workloads are not sensitive to the value when RP is small. Consequently, I choose RP = 1/32, which means that for every 32 memory writes (lines are not delta compressed), the memory controller evaluates one write to check whether the corresponding line should be delta compressed.

4.4.1 Compression Ratio

First, I evaluate the compressibility of the written data in the benchmarks. Figure 30 shows the compression ratio for FPC compression (COMP) and delta-compression (D-COMP). For the twelve studied benchmarks, while COMP achieves an effective average compression of 2.2X, D-COMP can achieve an average of 3.1X. For six out of twelve benchmarks, D-COMP achieves at least 35% more effective capacity than COMP because delta-compression converts unmodified data bits to zeros, which are more compressible. The results also show that, some floating-point benchmarks, like *GemsFDTD* and *leslie3d*, are difficult to compress, even with delta-compression.

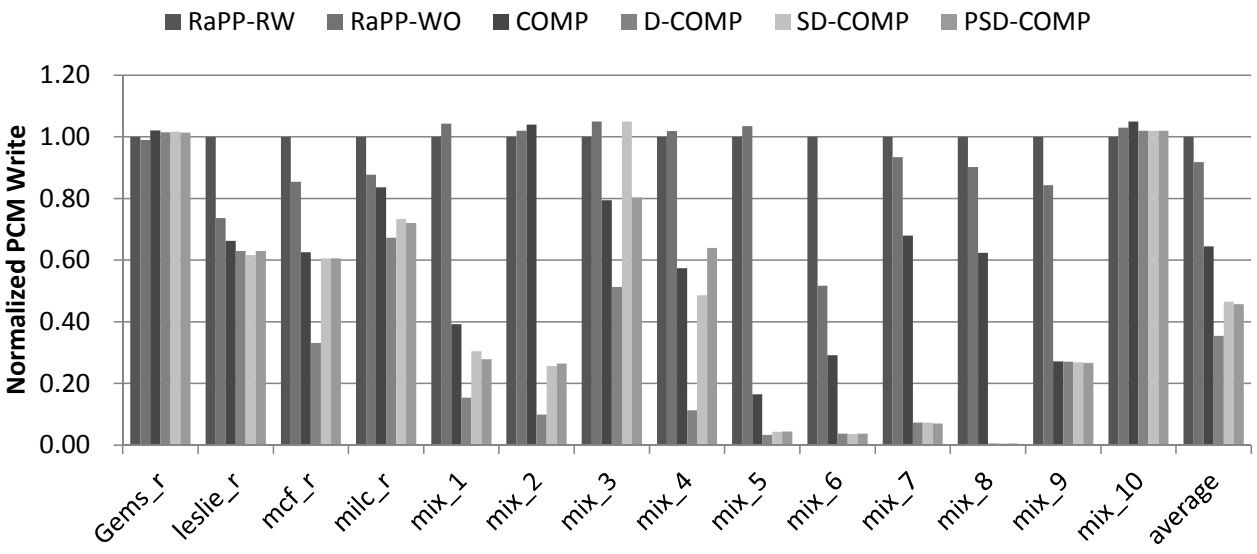


Figure 31: PCM writes normalized to RaPP-RW.

4.4.2 PCM Write

Figure 31 shows the number of PCM writes normalized to RaPP-RW baseline. On average, RaPP-WO, COMP and D-COMP reduce the number of PCM writes by 8.2%, 35.5% and 64.5%, respectively. It turns out that using only D-COMP more than 90% PCM write requests are absorbed by the DRAM cache for five out of fourteen workloads. This is because D-COMP can achieve a much higher compression ratio than COMP.

For *GemsFDTD*, compression is not effective and cannot reduce the number of PCM writes For *leslie3d*, the number of PCM writes will be reduced significantly if only modified data are cached in DRAM. In general, SD-COMP results in more PCM writes than D-COMP because SD-COMP only enables delta-compression on lines that have a minimum 64-byte storage gain (GT). PSD-COMP has almost the same number of PCM writes as SD-COMP because PSD-COMP only disables delta-compression for lines that are difficult to be delta-compressed, which will not change overall compression ratio. When DRAM compression is enabled, PCM endurance is proportionally improved with the reduction in PCM writes. On average, COMP, D-COMP, SD-COMP and PSD-COMP achieve 1.6X, 2.8X, 2.2X and 2.2X improvements in PCM lifetime over RaPP-RW[4].

[4]Assuming an ideal wear leveling mechanism.

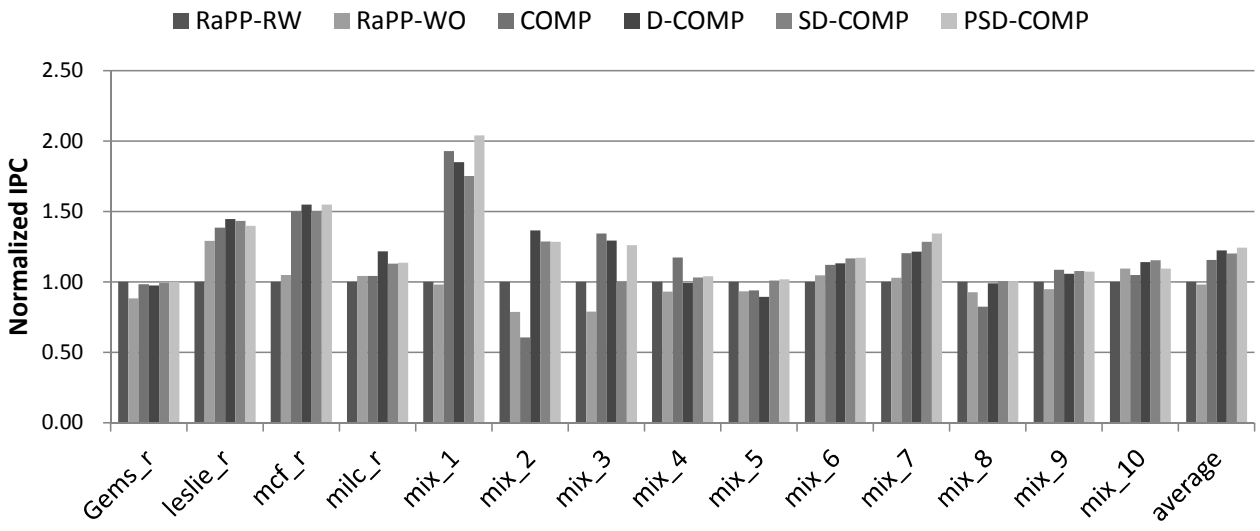


Figure 32: IPC normalized to RaPP-RW.

4.4.3 Performance

Figure 32 shows IPC improvement of 15.6% and 22.3% for COMP and D-COMP, respectively, when normalized to RaPP-RW. The difference shows the importance of enabling delta-compressed caching for hybrid memory. SD-COMP and PSD-COMP have similar improvement as D-COMP, but they bring higher power and energy savings (see next section).

Notice that blindly enabling compression will not always improve performance. This is because the performance gain on reducing PCM writes can be offset by the performance loss of increased accessing latency. A typical example is *mix-2*. With COMP, *mix-2* has no reduction on PCM writes but 25% performance penalty over RaPP-WO due to the extra read latency.

My results also show that on average RaPP-WO has similar performance as RaPP-RW. For *mix-2* and *mix-3*, RaPP-RW is better because more frequently-accessed data are cached in the DRAM cache. For *leslie_r*, RaPP-WO is better because more frequently-modified data are cached in the DRAM cache.

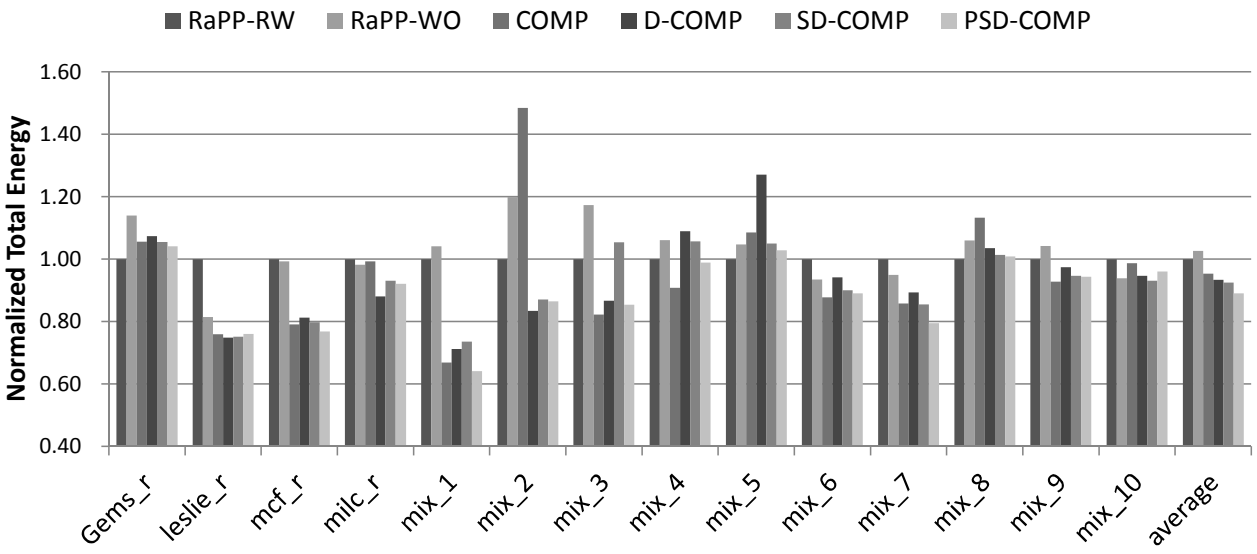


Figure 33: System energy consumption.

4.4.4 Energy Consumption

Although compression increases energy consumption, it reduces system total energy consumption due to the reduction in execution time. To be conservative in estimating system power, I modelled the 8-core processor with an average processor power of 25W. The energy consumption is computed by multiplying the execution time with system power. Figure 33 shows that system total energy consumption is reduced 4.6% and 6.6% with COMP and D-COMP, respectively. The reduction further goes to 7.5% and 11.0% with SD-COMP and PSD-COMP, respectively. *Mcf_r* and *mix-1* have large energy savings with compression, which is consistent with their IPC improvements, even though the power is much higher.

4.4.5 Impact of PCM Write Bandwidth

To analyze the impact of the PCM write bandwidth on the DRAM compression schemes, I carried out experiments with different PCM write bandwidth values. Figure 34 shows IPC normalized to the RaPP-RW baseline. As expected, the benefits of compression are higher when PCM write bandwidth is lower. When PCM write bandwidth is decreased to 1GB/s, IPC performance benefit

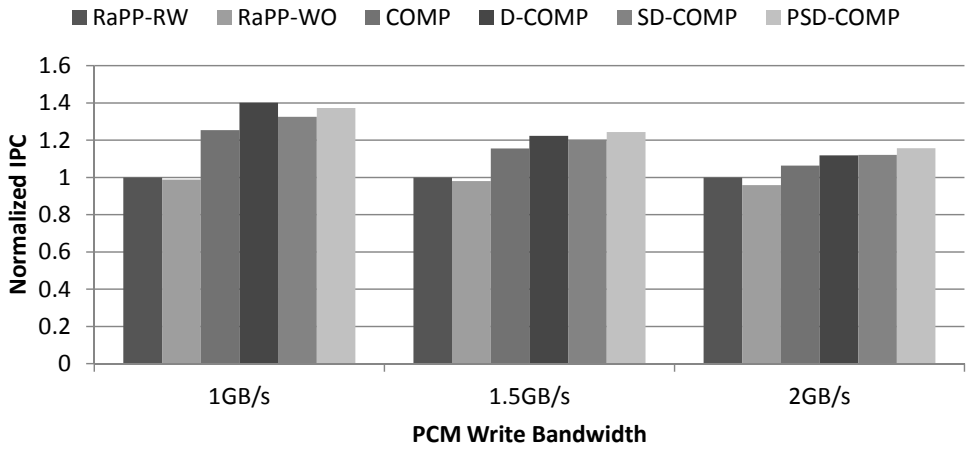


Figure 34: Sensitivity analysis of varing PCM write bandwidth on IPC

of PSD-COMP is increased to 37.3%. When PCM write bandwidth is increased to 2GB/s, IPC performance benefit of PSD-COMP is reduced from 24.4% to 15.7%. The performance gain from the reduction of PCM writes is offset by the increased latency of compressing memory. If PCM write bandwidth is the only bottleneck of the system, D-COMP will always get the best performance, because it maximize the number of PCM writes that can be reduced. These results show that the write bandwidth gap between DRAM and PCM is an important factor to determine whether or not memory compression should be enabled for hybrid memory.

4.4.6 Impact of Size of Compressed DRAM Region

I evaluate PSD-COMP with different sizes of the compressed region. The total DRAM capacity is fixed at 2GB. From Figure 35, the optimal partition size highly depends on the workloads. For *mix-5* and *mix-8*, it is better to use 1GB DRAM as compressed memory. For *milc_r* and *mix-3*, it is better to use 2GB DRAM as compressed memory. To achieve better performance, the memory controller should dynamically adjust the sizes of compressed and uncompressed regions of the DRAM (about 7% potential for IPC improvement), which is an aspect of my future work.

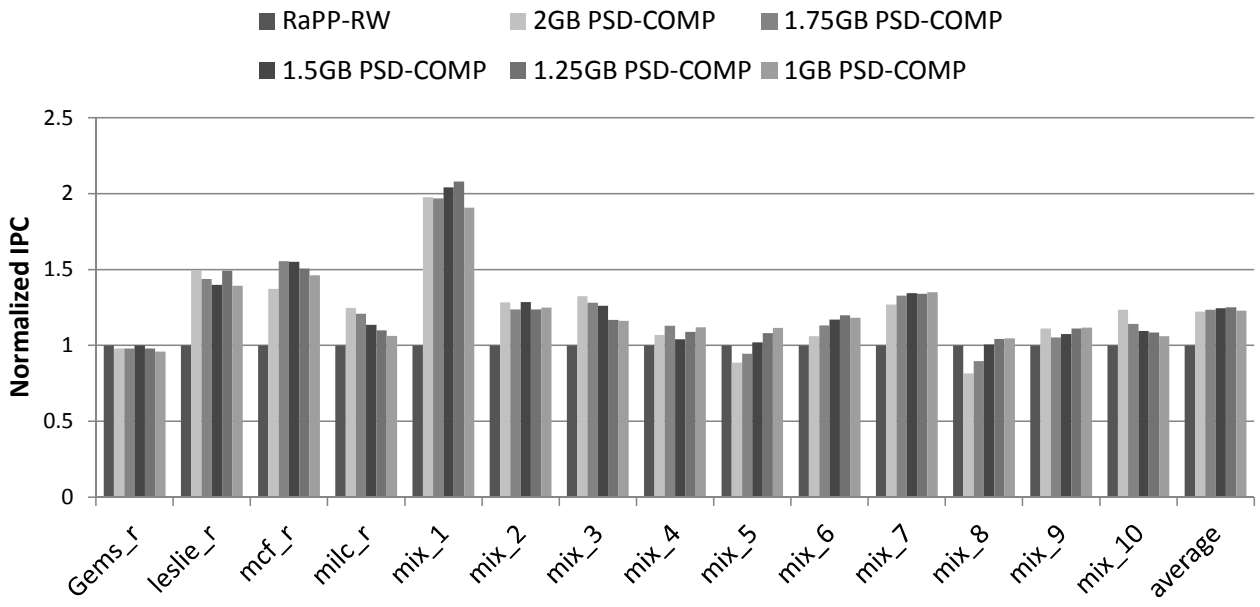


Figure 35: Speedup with different compressed region capacities (normalized to RaPP-RW)

4.5 CONCLUSION

Given that PCM write bandwidth is a major performance bottleneck for hybrid memory, compression has been used to reduce PCM write traffic by increasing DRAM effective capacity. I designed a novel DRAM cache compression scheme that is flexible and tailored to the specific challenges of hybrid memory. The proposed *delta-compression* algorithm stores a compressed version of the modified bits of the updated data in DRAM. I also proposed two extensions to further improve performance and efficiency of my compression scheme, namely *selective compression* and *predictive compression*. My results demonstrate that compression can significantly improve DRAM cache effective capacity by up to 6.3X and improve PCM lifetime by 2.2X on average. I observed 24.4% IPC improvement from my compression scheme over a non-compression design. I conclude that compression for the DRAM cache is an effective mechanism for improving the performance and increasing the endurance of hybrid memory.

5.0 SUPPORTING SUPERPAGES IN NON-CONTIGUOUS PHYSICAL MEMORY

5.1 PROBLEM STATEMENT

Non-volatile memories are being introduced in hybrid main memory systems to reduce memory static power [49, 70, 50]. These non-volatile memories, however, have limited write endurance [47], and cells gradually become non-programmable "bad" cells. Mechanisms have been proposed for error correction [52], but their limited error correction resources can tolerate limited number of errors. When uncorrectable memory errors occur, the affected memory page must be retired. Retired pages are marked as unusable [61] and prevented from being allocated in the future. Retired pages create many unusable holes in the physical address space and render the space non-contiguous. On the other hand, traditional superpages can only be constructed with contiguous physical memory blocks. As shown in Section 5.3, even a small number of retired pages can make it very difficult to find enough contiguous physical memory blocks to support traditional superpages.

The goal of the chapter is to find new storage-efficient page table formats to accommodate superpages in the context of non-contiguous physical memory. This is important because without superpages the performance overhead of virtual memory can be significant, specifically for memory-intensive workloads with large memory footprints and random access patterns. Four requirements are identified to achieve this goal:

1. Allow mapping a superpage to multiple non-contiguous memory area;
2. Use superpage tables that are of the same size as traditional superpage tables;
3. Guarantee that address translation is completed in a fixed number of steps; and,
4. Allow mixing superpages and traditional pages.

For backward compatibility and deployment, the new page table format is an optional extension to allow a portion of the non-contiguous memory to be mapped as superpages and the rest of memory to follow a traditional page table format.

5.2 UNDERSTANDING ADDRESS TRANSLATION OVERHEAD

There are three performance advantages to superpages. First, superpages increase TLB reach by the ratio of the size of a superpage to the size of a normal base page (i.e., TLB can cache 2-3 orders of magnitude larger address space), thus reducing TLB miss rate. Second, superpages reduce the number of levels during page walks, consequently reducing the latency of a TLB miss. Third, superpages significantly reduce the size of the page table; Table 7 shows page table sizes for different workload memory footprints and page sizes assuming an 8-byte page table entry. Note that for a workload with a 16GB memory footprint, the page table size for a traditional 4KB page is 32MB, which is already beyond the capacity of the Last Level Cache (LLC) for most processors. With 2MB superpages, the size of the page table is reduced to just 64KB and can easily fit in the cache.

Memory Footprint	1GB	16GB
4KB page	2MB	32MB
2MB superpage	4KB	64KB
1GB superpage	8B	128B

Table 7: Page table sizes for different workload memory footprints and page sizes.

To further understand the overhead of address translation, I characterized the performance of different problem sizes and TLB configurations for the GUPS workload [18], which is memory-intensive with a random access pattern. I use cycles per instruction (CPI) to measure performance. I study three TLB configurations: a 512-entry 4KB-page TLB, a 512-entry 2MB-page TLB and an 256K-entry 4KB-page TLB, which is much larger than any practical TLB design. Recent work shows that TLB reach can be improved by coalescing multiple TLB entries with similar con-

tents [42, 41]. The effective TLB reach of a 256K-entry 4KB-page TLB is 1GB, which is an upper bound that can be achieved by TLB coalescing.

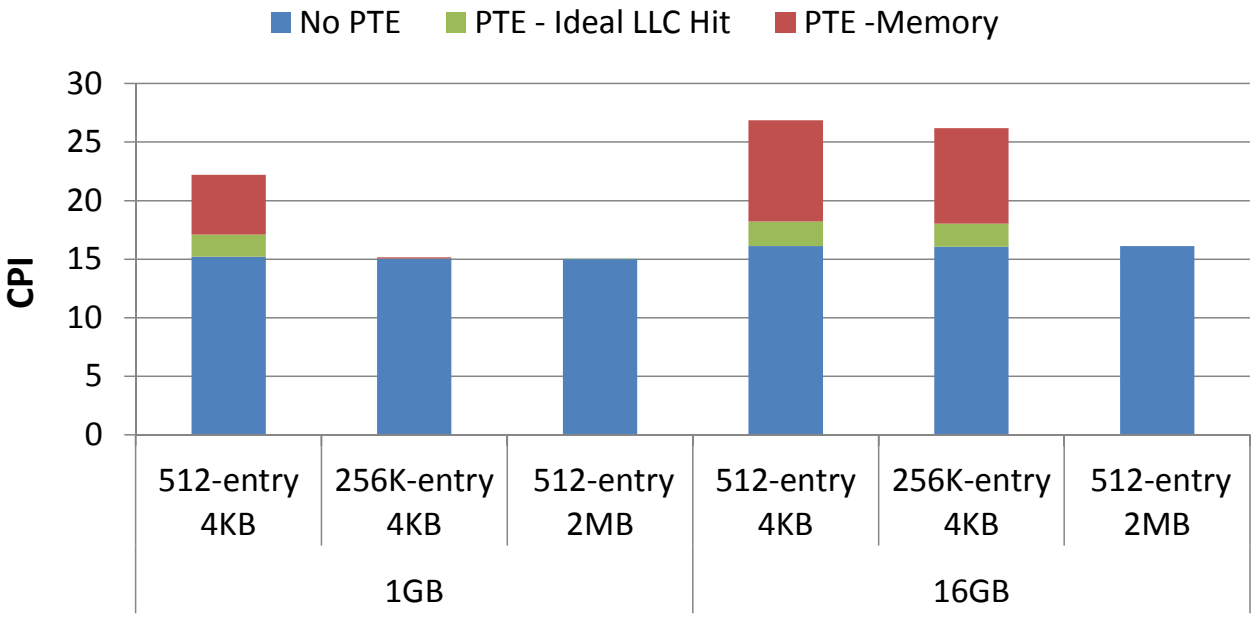


Figure 36: CPI breakdown with different problem sizes and TLB configurations for the GUPS workload.

As shown in the left side of Figure 36, when the memory footprint is 1GB, a 256K-entry 4KB TLB has similar performance as a 512-entry 2MB TLB, since the memory footprint is not larger than the 1GB TLB reach. When the memory footprint increases to 16GB (right side of Figure 36), which is much larger than the 1GB TLB reach, the performance improvement from increasing TLB reach becomes very small, but superpages perform very well.

Figure 36 characterizes the CPU cycles for an instruction on average, to understand the sources of the address translation overhead. Approximately 15 cycles are needed to access data, execute the instruction and account for address translation overhead in the first 3 levels (up to PDE, labeled No PTE in the figure). Compared to a 512-entry 2MB-page TLB, the additional address translation overhead of a 4KB-page TLB mainly comes from accessing PTEs.

The number of cycles to access PTEs can be broken down into two components: the cycles to access PTEs assuming all PTEs always fit in the LLC (PTE-Ideal LLC hit, middle cycles), and the cycles to access main memory if a PTE is not cached (PTE-Memory, at the top). For workloads

with large memory footprints (16GB), the performance overhead of accessing PTEs is dominated by main memory accesses.

In conclusion, to avoid address translation overhead from becoming a performance bottleneck, it is critical for workloads with large memory footprints to support superpages, which avoids accessing PTEs.

5.3 PAGE RETIREMENT AND MEMORY FRAGMENTATION

Given that superpages need contiguous physical memory, the physical memory can become fragmented when there is even a small percentage of retired pages. Figure 37 shows the probability of finding a contiguous memory block (of sizes 2MB, 128KB, 64KB and 32KB) as a function of percentage of retired 4KB pages (retired pages are uniformly distributed). The probability of allocating a 2MB superpage quickly approaches zero if the number of retired pages increases (e.g., for 0.5% retired pages, the probability is less than 8%). This implies that a traditional superpage implementation will be ineffective in the presence of retired pages. Nevertheless, it is relatively easy to find **small contiguous memory areas** when the percentage of the retired pages is small. To ensure that at least 60% of the memory blocks are contiguous (do not contain retired pages), the threshold on the percentage of the retired pages for 128KB, 64KB and 32KB superpages is 1.6%, 3.1% and 6.1%, respectively.

In the next section I propose a new method to construct superpages from multiple small memory fixed-sized areas instead of a single large contiguous memory area.

5.4 PROPOSED SOLUTION: GAP-TOLERANT SUPERPAGE

5.4.1 Gap-tolerant Sequential Mapping

When the physical memory is littered with retired pages, it is problematic to find a large contiguous memory block to establish a mapping. I devised *Gap-tolerant Sequential Mapping* (GTSM)

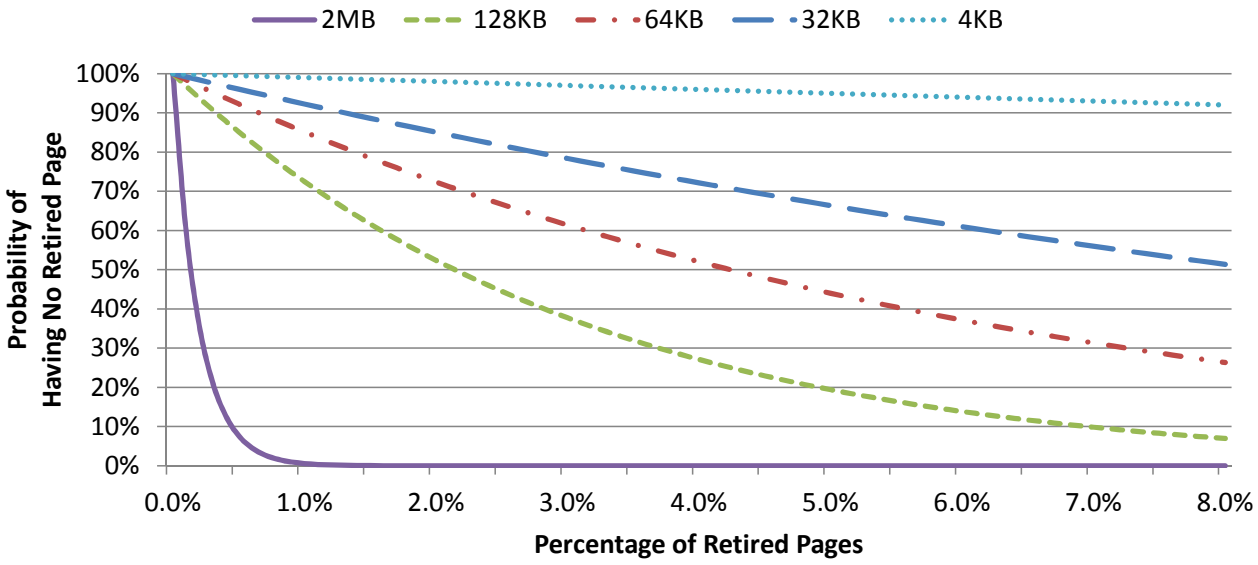


Figure 37: Probability of a memory block (of sizes 2MB, 128KB, 64KB and 32KB) to be contiguous (no retired pages) for different percentages of retired 4KB pages.

to support superpages in memory with retired pages. Figure 38 shows three ways to map a virtual memory space (VA) of the size of a superpage to physical memory (PA) that contains errors (marked with an X). Figure 38(a) shows traditional superpage mapping, that maps VA to contiguous PA (in this case, there is no contiguous physical space that can accommodate a superpage).

Figure 38(b) is the traditional page mapping, where each virtual page can be mapped to an arbitrary non-retired physical page. This flexibility is not free: the storage cost of fine-grained paged mapping is orders of magnitude higher than traditional superpage mapping. Figure 38(c) shows how GTSM divides a virtual superpage into multiple fixed-size smaller virtual blocks, which are sequentially mapped to memory (*B-blocks*, or building blocks). B-blocks are bigger than a regular page and together form a memory *slice*, whose size is twice the size of a superpage. Note that the utilization of memory is **not** only 50% with GTSM because remaining unmapped fragmented memory can still be used for traditional pages.

To maintain a one-to-one mapping between virtual blocks and B-blocks, exactly half of the B-blocks participate in the mapping, given the size of the memory slice. Any B-block that contains at least one retired page cannot be used for GTSM. Note that GTSM is a generalized form of

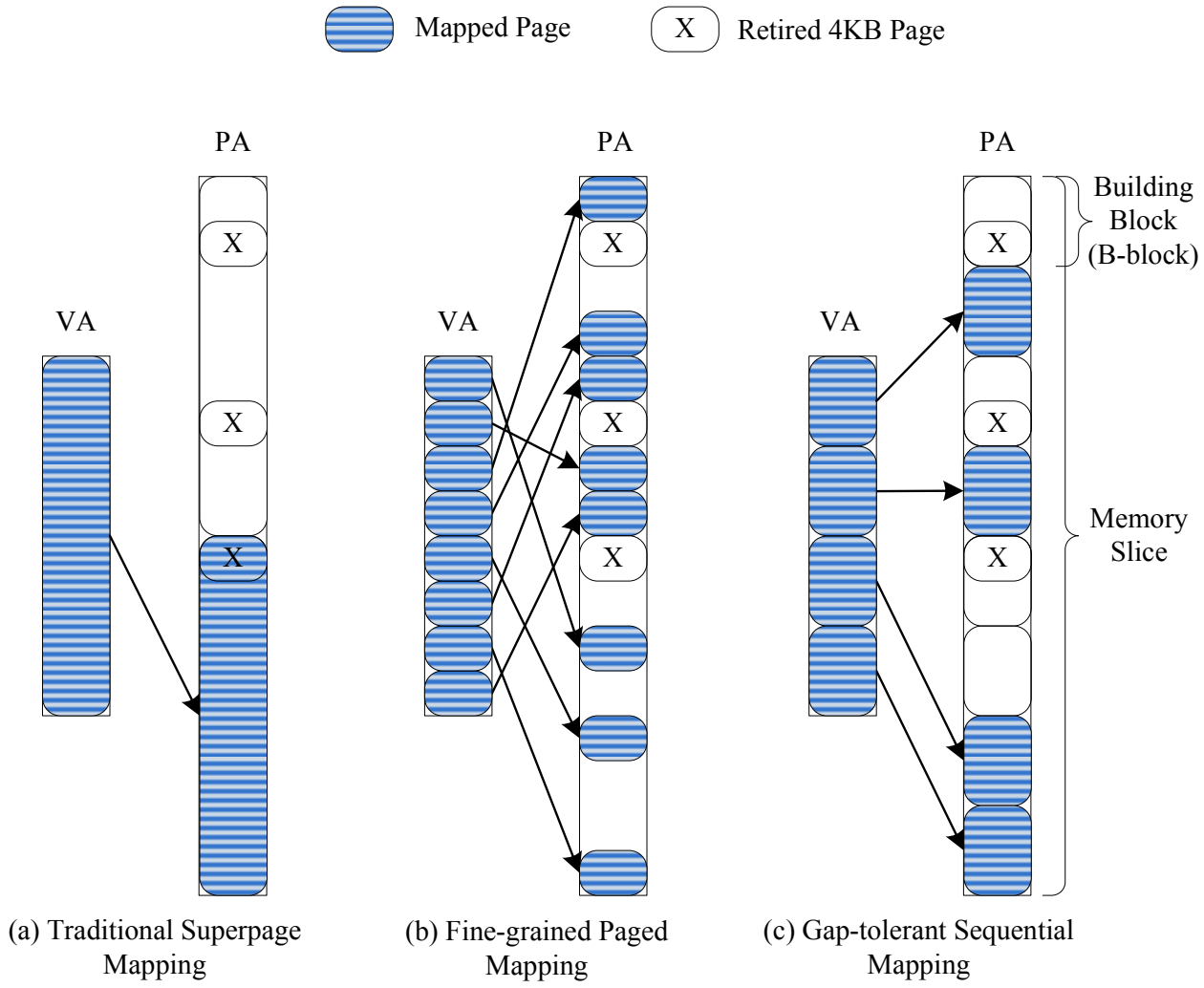


Figure 38: Examples of different address mapping schemes.

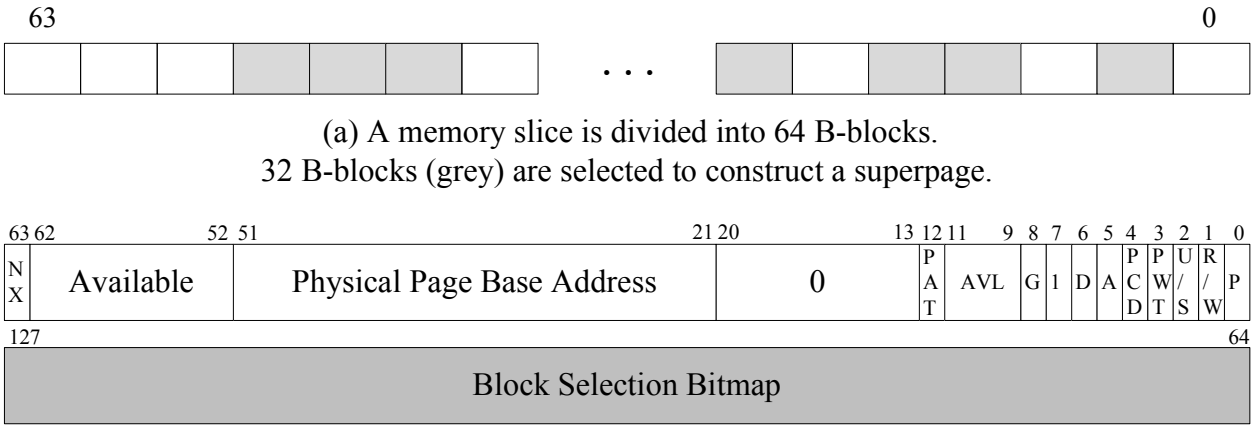


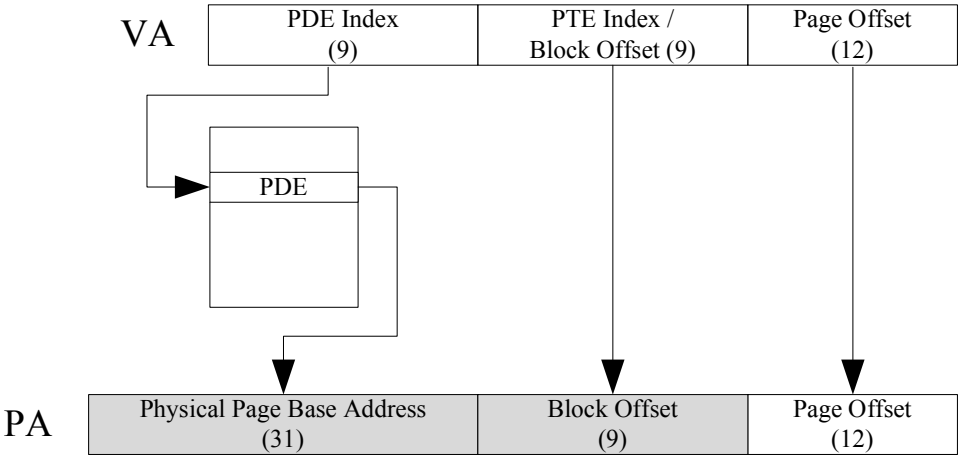
Figure 39: Gap-tolerant PDE (GT-PDE) format.

traditional superpage mapping, but it is more flexible to take into account retired pages. If there is no retired pages in a memory slice, GTSM creates the same (i.e., contiguous) memory mapping as traditional superpage mapping. By sacrificing flexibility of small/traditional page mapping, GTSM tolerates retired pages and maintains a page table that has similar storage efficiency as superpage mapping.

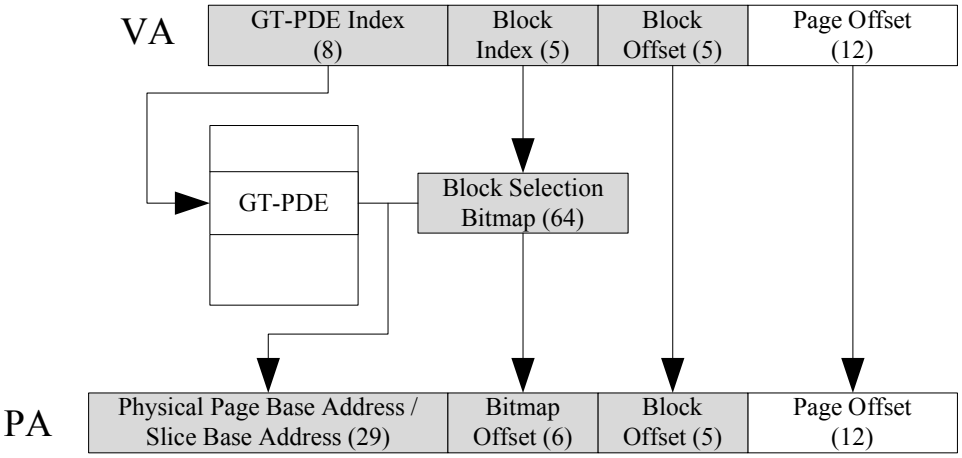
5.4.2 Gap-tolerant Page Directory Entry

For a 2MB superpage, page directory entry (PDE) is the last level of address translation; the PDE format contains the physical page frame base address and control flags of the superpage (present bit, access bit, dirty bit, etc.). To support GTSM, the 8-byte PDE is extended to a 16-byte Gap-Tolerant PDE (GT-PDE). Figure 39(a) shows a memory slice divided into 64 B-blocks with half of the B-blocks selected to construct a GTSM superpage. As shown in Figure 39(b), to minimize the impact on the OS, The first 64 bits of a GT-PDE is kept the same as a traditional 2MB-page PDE. An extra 64-bit B-block selection bitmap is appended for GTSM. The corresponding bit will be set to 1 if the B-block is selected in the mapping. Otherwise, the bit will be set to 0.

Figure 40(a) shows address translation of a traditional 2MB-page PDE. The 9-bit PDE index/block offset is used to select a PDE among 512 regular PDEs. The 9-bit PTE index will be



(a) Address translation of a 2MB-page PDE.



(b) Address translation of a 4MB-page GT-PDE.

Figure 40: Address translation using GT-PDE-4MB.

kept unchanged in the translated physical address. Since the default size of each page directory table is 4KB, it can hold 256 GT-PDEs instead of 512 regular PDEs. To avoid changing the size of the page directory table (4KB) and the size of the mapped memory range (1GB), each GT-PDE entry needs to map a 4MB superpage instead of a traditional 2MB superpage. Based on GTSM, a 4MB superpage is mapped to a 8MB memory slice. Since each memory slice has 64 B-blocks, the size of each B-block is 128KB.

Figure 40(b) shows the address translation of a 4MB-page GT-PDE. Only the upper 8 bits of the PDE index are needed to index a GT-PDE entry among 256 GT-PDEs. Since each B-block is 128KB, only the low 5 bits of the PTE index are used as block offset and kept unchanged during address translation. The remaining 5 bits between GT-PDE index and block offset are treated as block index. Block index is translated using the block selection bitmap of the selected GT-PDE. Same as a 2MB-page PDE, the physical page base address field of a GT-PDE entry is 31 bits. Because the mapped slice is aligned at an 8MB boundary, the low 2 bits of the physical page base address field are always zeros and ignored in the translated physical address. To translate block index K (0-31), the block selection bitmap is scanned to find the Kth selected bit, whose position in the bitmap (0 - 63) indicates the B-block that the virtual block is mapped to.

Because at least half of the B-blocks in the memory slice need to have no retired pages, Figure 41 shows the probability of constructing a valid mapping for different percentages of usable B-blocks, which are assumed to be randomly distributed in memory. A threshold of 60% is enough for most memory slices (93.3%) to find a valid mapping. Because only half of the B-blocks in a slice are used in GTSM, the memory capacity that can be mapped with GT-PDE is 46.6% with 60% B-blocks usable. Recall that the remaining memory capacity can be mapped as traditional pages.

5.4.3 Tolerating More Retired Pages

There is a trade-off between B-block size and number of retired pages allowed (robustness of mechanism). The B-block size used by a 4MB-page GT-PDE is 128KB. As shown in Figure 37, to tolerate more retired pages, a smaller B-block size should be chosen (to ensure enough usable B-blocks exist to find a valid mapping).

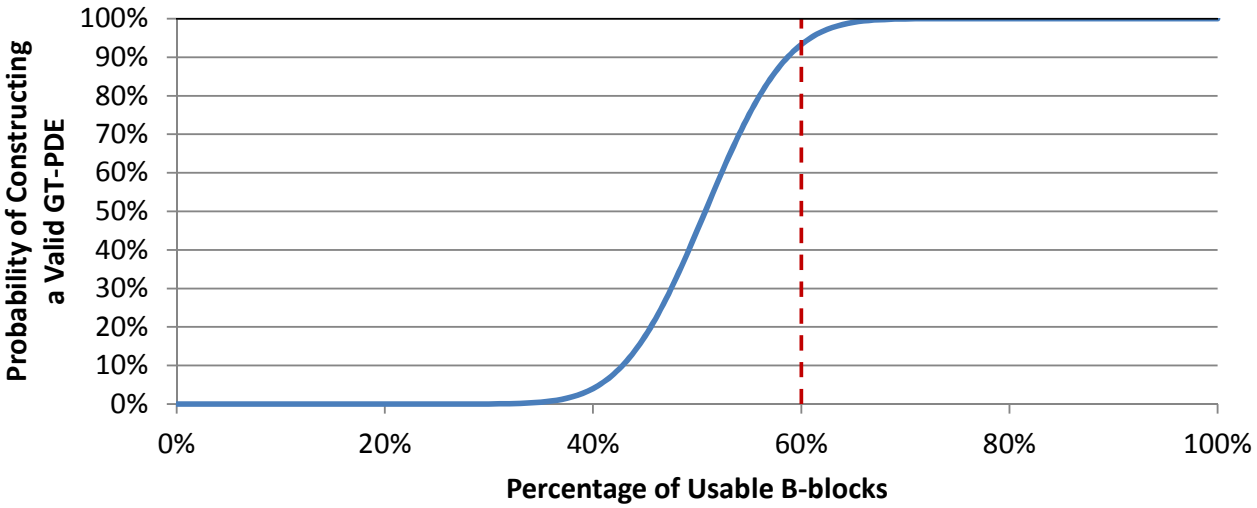


Figure 41: Probability to construct a valid GT-PDE mapping for different percentages of usable B-blocks.

When the bit of the block selection bitmap represents a smaller B-block, the format of the GT-PDE is not changed and the size of the superpage represented by a GT-PDE is reduced. To avoid changing the size of the mapped memory range (1GB), the page directory table needs to be expanded to hold more GT-PDEs. This change to use large pages (e.g., 16KB) as the page directory table is feasible. Some architectures, like ARM, already have this capability.

I limit the B-block size to 128KB, 64KB or 32KB. Table 8 shows the basic parameters of GT-PDEs. Supporting 16KB or smaller B-blocks requires changing the GT-PDE format to have more bits for the physical page base address. Although supporting 256KB or larger B-block size is possible, it is not considered for two reasons. First, 256KB or larger B-block size implies tolerating fewer retired pages ($< 1\%$), which is lower than the goal set for this work. Second, the size of the page directory table will be smaller than 4KB and become partially filled (i.e., wasted space) assuming a minimum page size of 4KB.

The translation process of GT-PDE with a smaller B-block size is similar to a 4MB-page GT-PDE. However, with a smaller B-block size, fewer address bits are used as block offset. At the same time, the page directory table is expanded to hold more GT-PDEs. More address bits are used to locate the GT-PDE in the page directory table.

GT-PDE Mode	B-block Size	Page Directory Table Size	Retired Page Threshold
4MB-page	128KB	4KB	1.6%
2MB-page	64KB	8KB	3.1%
1MB-page	32KB	16KB	6.1%

Table 8: Parameters of GT-PDEs with different B-block sizes.

5.4.4 Mixing Traditional Pages and Superpages

When GT-PDE is enabled, a page directory table can store both 4KB-page PDEs and GT-PDEs at the same time. Each 4KB-page PDE has the base address that points to the page table of PTEs. Each PTE will further point to each mapped 4KB pages. Each GT-PDE directly points to the physical base address of the mapped memory slice. 4KB-page PDEs need zero padding to fill the unused space if the page directory table is expanded. Figure 42 shows how to decode addresses using GT-PDE for various B-block sizes (see explanations below). When GT-PDE is enabled, the page directory table is accessed at aligned 16-byte granularity. Similar to a traditional 2MB-page PDE, the 7th bit of the accessed 16-byte is utilized to determine whether a PDE is a 4KB-page PDE or a GT-PDE. If the 7th bit is zero, the PDE should be decoded as a 4KB-page PDEs, otherwise the PDE should be decoded as a GT-PDE.

When B-block size is 128KB, every 16 bytes of the page directory table can store either two 4KB-page PDEs or one GT-PDE. There is no padding needed. When B-block size is 64KB, the size of the page directory table is doubled. To ensure 4KB-page PDEs are evenly distributed in the page directory table, each 4KB-page PDE needs to be padded with an 8-byte zero padding. When B-block size is 32KB, each 4KB-page PDE needs to be padded with a 24-byte zero padding. The storage overhead of the padding is small because the dominant storage cost of 4KB-page PDEs comes from their PTEs. When B-block size is 32KB, if the first access of a 4KB-page PDE points to the bottom-half 16 bytes, which are the zero padding, a second access to the top-half 16 bytes is needed. Similar to the storage cost, the performance overhead of the extra access is small because the dominant address translation overhead is from accessing PTEs.

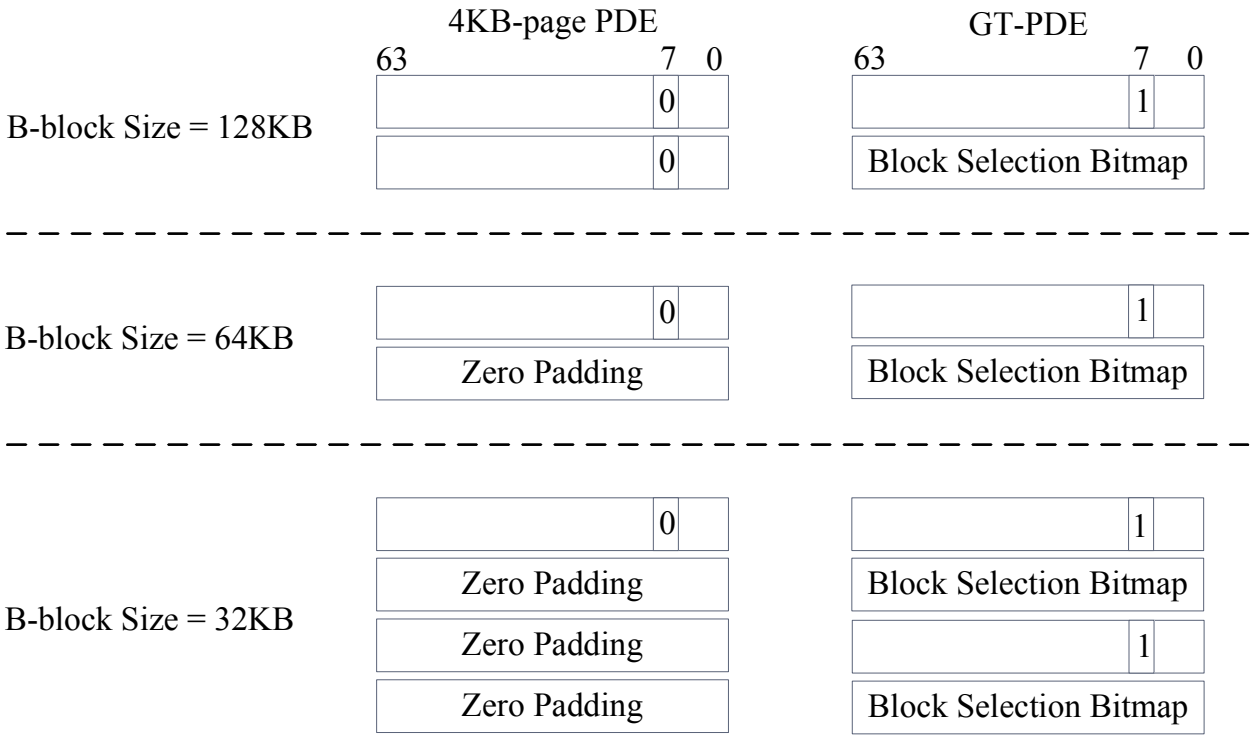


Figure 42: Decoding GT-PDE.

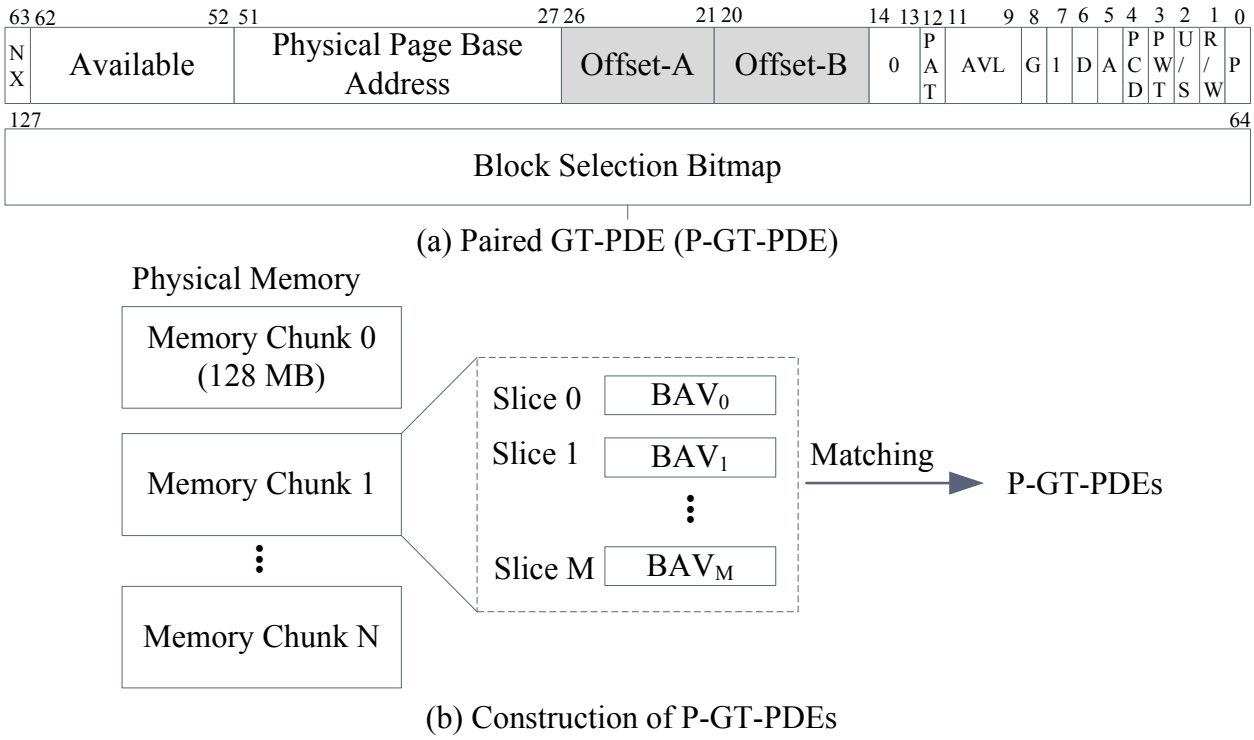


Figure 43: P-GT-PDE and its construction.

5.4.5 Compressing GT-PDEs

Recent research showed that TLB entries with similar contents can be coalesced to store more address translations in the TLB [42, 41]. Similarly, it is possible to halve the size of the page directory table by coalescing every two adjacent GT-PDEs. Figure 43(a) shows a Paired GT-PDE (P-GT-PDE) format to support GT-PDE coalescing. To coalesce two GT-PDEs with P-GT-PDE, two GT-PDEs use memory slices in a smaller range (128MB). Only the low 6 bits of physical page base address are different for the two slices. Also, the two GT-PDEs need to have the same block selection bitmap. Therefore, the low 6 bits of physical page base address of the second GT-PDE can be stored in the unused field (bit 15-20) of the first GT-PDE to form a P-GT-PDE. As shown in Figure 43(a), when a P-GT-PDE is accessed, it is simple to restore the coalesced GT-PDEs by masking the Offset-B field as zeros and overriding the Offset-A field with the value of the Offset-B field.

Figure 43(b) shows the matching process to construct P-GT-PDEs. First, the physical memory is divided into multiple aligned 128MB memory chunks. Each chunk is further divided into memory slices. Based on the distribution of retired pages, each slice has a 64-bit Block Availability Vector (BAV), which indicates which B-blocks of the slices can be used in the P-GT-PDE mapping. If a B-block is usable, the corresponding bit in BAV is set to 1, otherwise it is set to 0. A matching algorithm finds memory slices that should be paired.

Algorithm 5.1 is my matching algorithm to find GT-PDE pairs to construct P-GT-PDEs. First, for each memory chunk, a BAV can be constructed for each memory slice. A 128MB memory chunk has 64 2MB-size slices or 32 4MB-size slices. Then, BAVs are sorted based on the number of usable B-blocks in ascending order. The algorithm sequentially scans the remaining BAVs to find two unprocessed BAVs that can be paired using the bitwise-AND. If the result has at least 32 bits set, the slices can share a valid block selection bitmap. Matching continues until there are no more unprocessed BAVs. A special case is a memory slice which does not have any retired page, whose top half and bottom half slices can be *Self-Paired*. For a Self-Paired P-GT-PDE, the block selection bitmap is all ones.

Because the matching algorithm is done locally for each 128MB memory chunk, the time complexity of the algorithm is proportional to memory capacity. I tested my serial version of the

Algorithm 5.1 Construction of P-GT-PDEs in a Memory Chunk

Parameters:
$B_0 ... B_N$: a group of BAVs

Initialize the state of each BAV to *Unprocessed*.
Sort BAVs based on the number of usable B-blocks of each BAV in ascendant order.
Any BAVs with all B-blocks usable are marked as *Self-Paired*.
while the number of *unprocessed* BAVs ≥ 2 **do**
  Scan the BAV list to find the first *Unprocessed* BAV B_i.
  for each remaining *unprocessed* BAV B_j **do**
    B_{merged} = B_i bitwise AND B_j.
    if the number of usable B-blocks of $B_{merged} \geq 32$ **then**
      Record (B_i, B_j) as a valid P-GT-PDE.
      Mark B_i *and* B_j as *Paired*.
      Go to find next unprocessed BAV B_i to process.
    end if
  end for
  Mark B_i as *Discarded*.
end while

algorithm on a 2.8GHz Intel Xeon E5-2680v2 processor. It takes less than 0.1 second to find all BAV pairs for 128GB memory.

5.4.6 Hardware Implementation

To support GTSM, three hardware changes are introduced. First, a new 64-bit Gap-Tolerant Page Table Control Register (GTPTCR) is used to manage the parameters of GTSM. Second, the hardware page walker is extended to support loading missed TLB entries from GT-PDEs. Third, the PDE cache in the MMU is extended to hold 16-byte GT-PDEs. Next, we describe these hardware changes in detail.

As shown in Figure 44(a), GTPTCR has three fields: GT, P and BS. GT indicates whether GTSM is enabled. The page table of a process can use both 4KB-page PDE and traditional 2MB-page PDE. Alternatively, the page table of a process can use both 4KB-page PDE and GT-PDE. To avoid adding an extra flag in the PDE, the page table of a process cannot use both traditional 2MB-page PDE and GT-PDE. The P field indicates whether P-GT-PDE format is used. The BS field indicates the B-block size. If BS is n, the corresponding B-block size is $2^n \times 4$KB. The remaining unused bits in GTPTCR are reserved for future extension. Table 9 shows the PDE modes that are

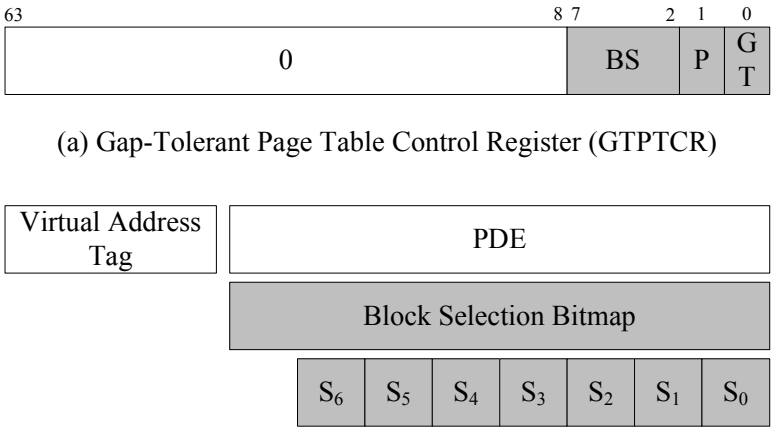
defined by the GTPTCR.

GT	P	BS	B-block Size	PDE Mode
0	0	0	-	2MB-page PDE
1	0	3	32KB	1MB-page GT-PDE
1	0	4	64KB	2MB-page GT-PDE
1	0	5	128KB	4MB-page GT-PDE
1	1	3	32KB	2MB-page P-GT-PDE
1	1	4	64KB	4MB-page P-GT-PDE

Table 9: List of all PDE modes in GTPTCR.

When a GT-PDE/P-GT-PDE is accessed, the hardware page table walker needs to translate the block index to a bitmap offset using the block selection bitmap. As shown in Figure 44(b), each PDE cache entry needs an extra 15-byte storage to store the block selection bitmap (8 bytes) and byte-granularity prefix sums (7 bytes). When a GT-PDE/P-GT-PDE is loaded into a PDE cache entry, byte-granularity prefix sums are calculated and cached to reduce translation latency. As shown in Figure 44(b), byte-granularity prefix sum, S_i, is the accumulated number of 1s of the first $i+1$ bytes of the block selection bitmap. Similar bit-counting logic is already implemented in modern processors and can be reused to reduce hardware implementation cost. For example, x86-64 POPCNT instruction counts the number of 1's of a 64-bit register in 3 cycles. S_7, which is the number of 1s of the whole block selection bitmap, is not stored because it is never used in address translation.

For each address translation using GT-PDE/P-GT-PDE, the byte-granularity prefix sums are compared with the value of $blockindex$. The $(i+1)^{th}$ byte of the block selection bitmap contains the matched bit position if $S_i > blockindex$ and $S_{i-1} \leq blockindex$. After the matched byte is determined, bit-granularity prefix sums are calculated for each bit position of the matched byte. The eight bit-granularity prefix sums are compared with the value of $blockindex - S_{i-1} + 1$, the bit position of the matched bit-granularity prefix sum is the translated bitmap offset. Also, address translation using the block selection bitmap can be done in parallel with other operations that are needed to fill a TLB miss (e.g., validating the access rights of the superpage). I assume that loading



(a) Gap-Tolerant Page Table Control Register (GTPTCR)

(b) PDE Cache Entry to support GT-PDE/P-GT-PDE

Figure 44: Hardware implementation of GT-PDE.

a TLB entry from a GT-PDE takes an three more cycles than a traditional PDE. I also carried out a sensitivity study on this penalty (see Section 5.6).

In this work, I assume that the baseline processor has a 32-entry PDE cache to store recently-accessed PDEs [3, 6]. The total storage overhead to support GT-PDE is 488 bytes:

$8\ bytes\ (GTPTCR) + 32 \times 15\ bytes\ (PDE\ cache\ entries)$.

To minimize the changes to the MMU, my design does not change the TLB hierarchy. The hardware page table walker is enhanced to support GTSM. When address translation is completed, a 4KB TLB entry is inserted into the TLB hierarchy for the translated address. Early x86-64 processors have also used a similar method to support traditional superpages. Alternatively, the TLB hierarchy can be enhanced to provide native support for GT-PDE.

5.4.7 Software Support

To enable GTSM, the OS needs to support functions to 1) configure GTPTCR; 2) determine whether to use traditional or GT-PDE superpages based on the setting of GTPTCR; 3) track memory slices that can be mapped as GTSM superpages; and, 4) install and release GTSM superpages.

To track memory slices that can be mapped as GT-PDE superpages, a BAV array can be used to store the usability information of B-blocks. Each memory slice has a dedicated 64-bit BAV. When

the B-block size is 128KB, the memory storage cost of the BAV array is 128KB for 128GB main memory. If the B-block size is halved, the memory storage cost of the BAV array will be doubled.

When the OS boots, it initializes the BAV array using a fault map of pages with errors. The fault map can be either stored in a permanent storage or constructed with memory built-in self-test (mBIST) during boot. The OS needs to keep the BAV array updated by using information from the kernel physical page allocator (e.g., Linux Buddy Allocator). Once a memory page of a B-block is allocated, the corresponding bit of the BAV needs to be set to 0. Once all the memory pages of a B-block are freed, the corresponding bit of the BAV needs to be set to 1. A memory slice can be used for GT-PDE memory allocation if more than half of its B-blocks are usable. To avoid scanning the BAV array for each GT-PDE memory allocation, all BAVs that can be used for GT-PDE allocations can be maintained in a dedicated list.

Unlike GT-PDE, P-GT-PDE should be used only for processes with very large memory footprints, and compressing GT-PDEs can further reduce TLB miss penalty. Because the matching algorithm described in Section 5.4.5 needs to be applied to the BAV array to find BAVs that can be paired, it is more expensive to make memory allocation with P-GT-PDE than GT-PDE. To utilize P-GT-PDE, physical memory should be allocated at the early stage of the process lifetime and released when the process is completed.

In this work, I assume that all processes use the same B-block size. To support per-process B-block size, the OS needs to track BAVs at multiple granularities.

5.5 EVALUATION ENVIRONMENT

5.5.1 Configuration

I use PTLsim [66], a cycle-accurate simulator, for performance evaluation. The simulation parameters are detailed in Table 10. The CPU is configured as a 2GHz out-of-order processor core with a 512KB L2 cache and a 2MB L3 cache slice. After L3 cache, there is a 256MB DRAM cache (32MB quota per core) before the PCM main memory [44]. Main memory capacity is 128GB PCM with 80ns access latency.

To evaluate different page table designs, I extended PTLsim with a TLB performance model. The L1 DTLB has 64 entries for 4KB pages and 32 entries for 2MB pages. The L1 ITLB has 64 entries for 4KB pages. The unified L2 TLB has 512 entries for both 4KB and 2MB pages. L1 TLB miss penalty is 7 cycles if it hits in L2 TLB.

Besides the two-level TLB, a MMU cache [3] is modeled. The MMU cache has 32 PDE/GT-PDE cache entries, 32 PDPE cache entries and 2 PML4E cache entries. Although the number of entries in the PDPE is larger than usual, I increased it to reduce TLB miss penalty for workloads with large memory footprints, as suggested in previous work [3, 6]. The MMU cache is indexed by virtual address and is concurrently looked up with L2 TLB [3]. I assume 5 cycles for the hardware page walker to access a PTE/PDE/PDPE/PML4E not including the cycles to load the entry from the cache/memory hierarchy. I assume that it takes an extra 3 cycles to access a GT-PDE due to the address translation latency using the block selection bitmap. The hardware page walker is not speculative (all configurations).

Since the B-block size is larger than a traditional page of 4KB (I experimented with 32KB, 64KB and 128KB), the selection of B-blocks in a memory slice has negligible impact on performance. For L1 and L2 cache, if a virtual page is mapped to different B-blocks, data at a given virtual address is still mapped to the same cache set. The selection of different B-blocks only affects the value of cache tags, the cache replacement sequence is kept unchanged. The memory pages for the page table are pre-allocated to simplify the simulation process. A similar reservation-based allocation policy has been used to allow MMU cache coalescing [6].

I use a Monte Carlo method to calculate the effective capacity of different GT-PDE/P-GT-PDE designs with different percentages of retired pages. To reduce the error introduced by the Monte Carlo method, I modeled randomly-distributed retired pages in a large physical memory sample (16PB capacity). I use Intel RdRand instruction [29] to generate different percentages of retired pages with a uniform random distribution.

5.5.2 Workloads

Since the work studies address translation overhead of virtual memory, only memory-intensive benchmarks with large memory footprints are considered. I use these applications because they are

CPU Core	2GHz, out-of-order, 32KB L1 I/D
L2 Cache	512KB, 8-way, 64-byte line size, 8-cycle latency
L3 Cache	2MB per core, 32-way, 64-byte line size, 20-cycle latency
DRAM Cache	256MB total, 32MB per core, 32-way, 256-byte line size, write-back, 60-cycle (30ns) read latency, next line sequential prefetcher
L1 DTLB	64-entry 4-way 4KB page 32-entry 4-way 2MB page
L1 ITLB	64-entry 4-way 4KB page
L2 TLB	512-entry 4-way 4KB/2MB page 7-cycle latency
MMU Cache	32-entry 4-way PDE/GT-PDE cache 32-entry 4-way PDPE cache 2-entry PML4E cache 5-cycle PTE/PDE/PDPE/PML4E access 8-cycle GT-PDE access
Main Memory	128GB PCM, 80ns read latency.

Table 10: System settings.

becoming prevalent and suffer the most from address translation overhead. I choose *GUPS* [18], *Canneal* from PARSEC [9] and 7 benchmarks from Problem Based Benchmark Suite [55]. *GUPS* is a popular benchmark to test random memory access performance. *Canneal* is a cache-aware simulated annealing kernel to minimize the routing cost of a chip design. *Dict* is a benchmark to test performance of batch insertion, deletion and search operations with a dictionary data structure. *BFS* runs a breadth first search in a directed graph. *SetCover* finds an approximate solution to the NP-hard set cover problem. *MST* finds the minimum spanning tree (MST) in an undirected graph. *SPMV* is multiplication between a sparse matrix and a dense matrix. *Matching* finds a maximal matching in an undirected graph. *MIS* finds a maximal independent set (MIS) in an undirected graph. With my current simulator, only single-threaded workloads are evaluated. Multi-threaded workloads should have similar results, given there will be even larger memory requirements by multiple applications or threads running concurrently. The pressure on the cache and sizes of page tables tend to be even bigger.

Name	Memory Read PKI	TLB Miss PKI		Memory Footprint(GB)	
		4KB	2MB	Touched	Total
GUPS	17.9	17.9	13.4	4.0	4.1
Canneal	24.4	21.2	2.9	3.7	4.0
dict	23.1	21.4	0.0	0.7	6.5
BFS	93.2	88.1	4.4	1.4	7.4
setCover	60.4	49.4	0.0	0.9	7.8
MST	50.0	43.2	0.0	1.0	13.0
SPMV	128.7	113.5	0.0	1.7	7.3
matching	119.1	109.7	0.0	0.9	6.2
MIS	144.4	124.3	0.0	1.3	7.3

Table 11: Simulated workloads and PKIs.

For the graph benchmarks, I use R-MAT graphs [10] as the input. For *Dict*, I use an uniform random distribution as the input. For each workload, I skipped the initialization phase and simulated 2 billion instructions. All benchmarks are 64-bit binaries, compiled with gcc 4.1.2. Table 11

shows the number of memory reads per 1000 instructions (PKI), TLB Miss PKI after a 512-entry L2 TLB (both 4KB pages and 2MB pages), and memory footprints of each workload (both the total footprint and the size of memory touched by the 2B-cycle simulation I ran). Most workloads can only touch a small portion of their total memory footprints during the simulation interval. A 512-entry 2MB-page L2 TLB can provide enough memory coverage (1GB). Most workloads have negligible TLB misses with ideal 2MB superpage.

Since accessing PTEs is a major source of address translation overhead for workloads with large memory footprints, Figure 45 shows the breakdown of PTE accesses based on whether the PTE is accessed in memory, LLC, or L2 cache. I assume that the L2 cache is the fastest cache large enough to cache PTEs; essentially, caching PTEs in the L1 cache could cause significant adverse cache pollution and severely harm performance. The breakdown of PTE accesses is an inherent characteristic of each workload, and is sensitive to the memory footprints of the workloads. The workloads that I studied can be divided into two categories. The slower page table access category are those applications that have a large portion (i.e., more than 10%) of PTE accesses to memory, given that the access to memory is 7.5 times slower than LLC: *GUPS*, *Canneal*, *dict* and *BFS*. The second category are those applications that have faster time to access PTs, with few PTE access to memory: *setCover*, *MST*, *SPMV*, *matching* and *MIS*.

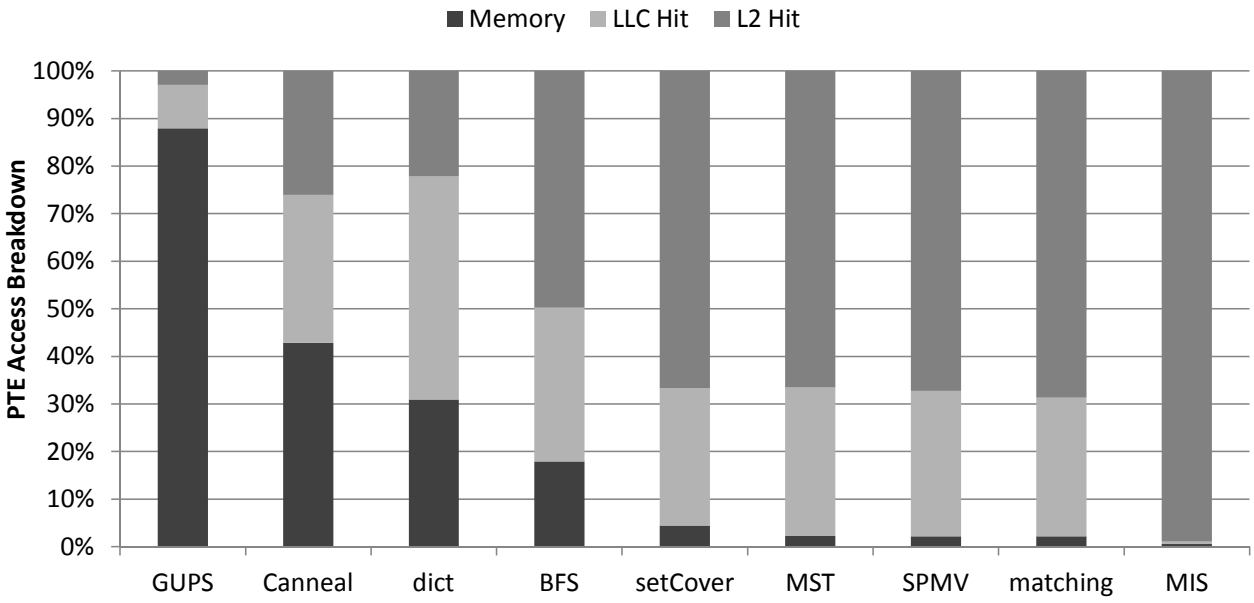


Figure 45: PTE access breakdown.

5.6 RESULTS

This section presents simulation results of GT-PDE/P-GT-PDE superpages. I show how performance is improved in comparison to traditional 4KB pages. I also show *Ideal* case, that is, traditional 2MB superpages with no retired pages (in other words, no errors occur in memory). Traditional 2MB superpage is only suitable for memory where retired pages are rare. In the figures, I use GT-PDE-*x*MB to denote *x*MB-page GT-PDE. Similarly, I use P-GT-PDE-*x*MB to denote *x*MB-page P-GT-PDE.

5.6.1 TLB Miss Penalty

Since my proposed design does not change the TLB hierarchy, it has the same TLB miss PKI as the traditional 4KB page baseline. As a superpage table format, GT-PDE does not need to access a PTE for each page table walk, which significantly reduces the TLB miss penalty. Figure 46 shows the average TLB miss penalty of the traditional 4KB page baseline and GT-PDEs. Compared with Figure 45, strong correlation between PTE access breakdown and the reduction of TLB miss

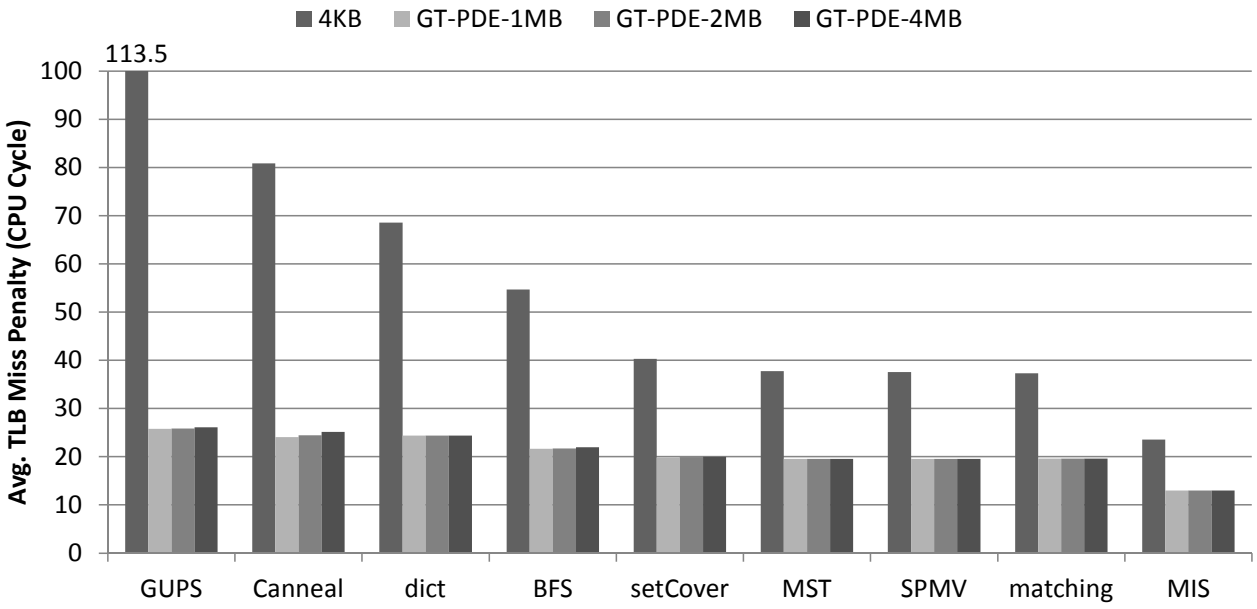


Figure 46: Average TLB miss penalty.

penalty is observed. For *GUPS*, *Canneal*, *dict* and *BFS*, average TLB miss penalty reduces by 30-90 CPU cycles because a significant portion of PTEs are accessed from memory for the traditional 4KB page baseline. For the workloads I studied, the average TLB miss penalty of GT-PDE for 1MB, 2MB, and 4MB are all similar because the page tables fit in the same cache level.

5.6.2 Performance

Figure 47 shows IPC improvement over the traditional 4KB page baseline. The graph shows the improvement for GT-PDE with different superpage sizes and *Ideal* superpages (i.e., superpages with no retired/faulty pages). Similar to Figure 46, strong correlation between PTE access breakdown and IPC performance improvement is observed. For *GUPS*, *Canneal*, *BFS* and *dict*, significant performance improvement (10% to 30%) is observed with GT-PDEs because PTEs are no longer accessed from memory. For *setCover*, *MST*, *SPMV*, *matching* and *MIS*, moderate performance improvement is observed with GT-PDE because these workloads access PTEs that are mostly cached in the L2 and the LLC; the address translation overhead is not significant enough to cause a large difference in the IPC, which is approximately 3% to 8%. *MIS* has the lowest

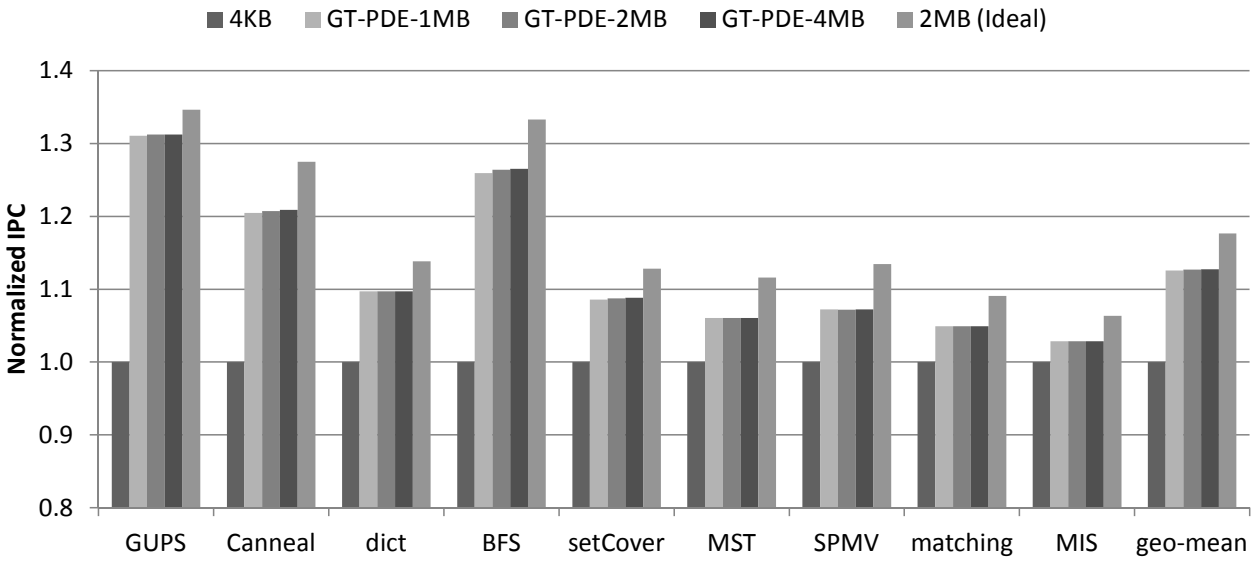


Figure 47: IPC normalized to traditional 4KB page baseline.

performance gain (2.9%). *MIS* has 99% PTE accesses to the L2 cache, and is less sensitive to address translation overhead. On average, GT-PDE-4MB achieves 95.8% performance of Ideal. The 4.2% overhead mainly comes from the extra 3 cycles to translate and access GT-PDE entries from the cache hierarchy. For the workloads that I studied, the IPC performance of GT-PDE for 1MB, 2MB, and 4MB are all similar because they use the same address translation procedure and have similar TLB miss penalty. Due to the same reasons (both fit in the L2 or LLC), even though the paired schemes (P-GT-PDE, not shown) reduce the page table size by 50%, they have similar performance as GT-PDE. The performance advantage of a smaller page table size is demonstrated in Section 5.6.4.

In my default configuration, the L2 cache is the highest level to cache the page table. I also evaluated the configuration that PTEs can be cached in the L1 cache. For the workloads that I studied, the performance change is very small ($< 0.1\%$) compared to my default configuration for the traditional 4KB pages. On the other hand, there is an extra 1% performance gain on average if GT-PDEs can be cached in the L1 cache instead of only in the L2 cache.

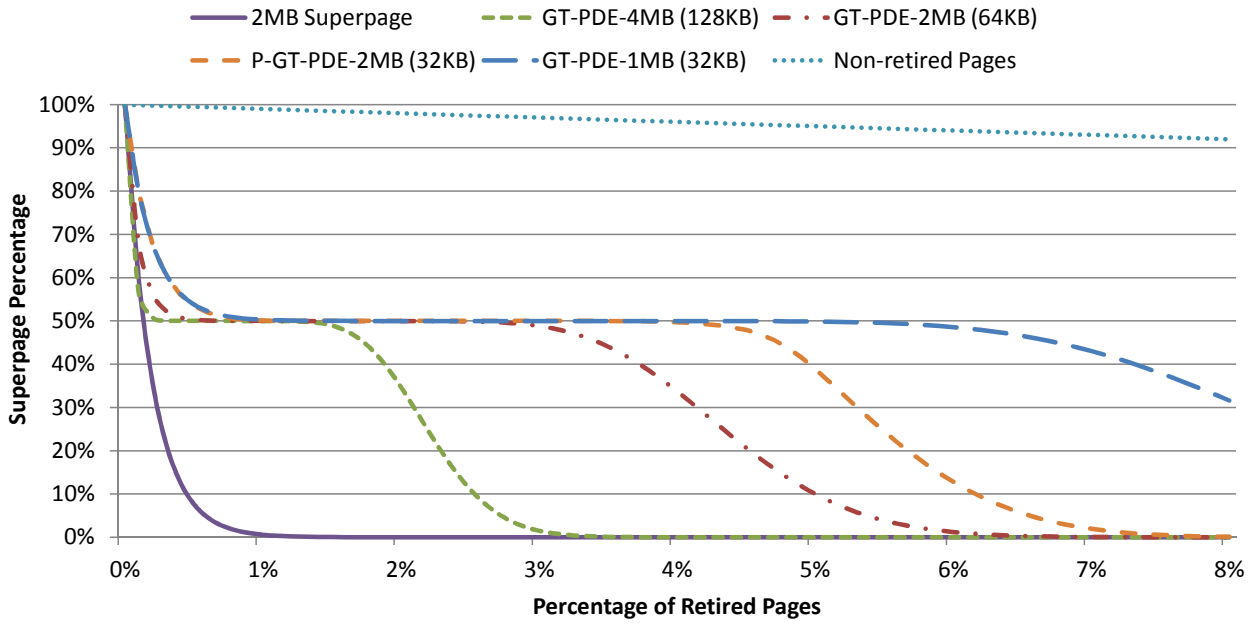


Figure 48: Superpage percentage of different GT-PDEs.

5.6.3 Memory Capacity Used as Superpages

Figure 48 shows *superpage percentage*, that is, the percentage of memory capacity that can be used as superpages using different page table formats. Using more superpages is beneficial due to the speedup achieved (recall that superpages do not need to traverse the last level of page tables). Note that the remaining non-retired memory pages can still be used and mapped using traditional 4KB-page PDEs. For traditional 2MB superpages, the percentage quickly drops to 0%. For GT-PDEs, the superpage percentage to utilize GT-PDEs drops to 50% with increased retired pages, because each memory slice is likely to have at least one retired page. When there is a retired page, only 50% of B-blocks of a memory slice can be used in the GT-PDEs. As shown in the figure, to have 50% superpage percentage, the thresholds of retired pages should be 1.4%, 2.8% and 5.5% for B-block sizes of 128KB, 64KB and 32KB, respectively. The superpage percentage of P-GT-PDE-2MB (paired approach) is bounded from below by GT-PDE-2MB and above by GT-PDE-1MB. Compared to GT-PDE-2MB, the smaller B-block size allows P-GT-PDE-2MB to tolerate more retired pages while maintaining the same page table size.

5.6.4 Sensitivity to Problem Size

Figure 49 shows IPC of *GUPS* with different problem sizes and page table formats. I choose to evaluate *GUPS* because it is a common benchmark in scalability studies, given that its memory footprint varies with problem size from 64MB to 64GB. As shown in the figure, the performance of traditional 4KB pages is sensitive to the problem size. When the problem size is increased from 64MB to 8GB or more, IPC reduces by more than 50%. This is due to two reasons. First, the limited capacity of the dram cache (256MB) is less effective for very large problem sizes. Second, the address translation overhead is significantly increased with a larger problem size. On the other hand, the performance advantage of superpages is significantly increased with lager problem size. When the problem size is 8GB, the IPC of GT-PDE-4MB superpage is 40.9% better than 4KB page. The performance of GT-PDEs with different B-block sizes shows difference when the problem size is very large. When the problem size is 64GB, GT-PDE-4MB is 8.7% better than GT-PDE-1MB because of the smaller size of the page table. Since the major advantage of P-GT-PDE is to reduce the size of the page table, this also implies that P-GT-PDE should only be used for processes with very large memory footprints.

5.6.5 Sensitivity to GT-PDE Address Translation Latency

Figure 50 shows the IPC of GT-PDE-4MB assuming different translation latencies of GT-PDEs. All results are normalized to the default 3-cycle extra latency. As shown in the figure, the performance overhead is mostly consistent among the workloads and is less than 1% if GT-PDE translation latency is 6 cycles instead of 3 cycles.

5.6.6 Sensitivity to GT-PDE Cache Size

Figure 51 shows the IPC improvement of GT-PDE-4MB with a larger PDE cache. The results are normalized to the 32-entry PDE cache baseline. As shown in the figure, the performance is not very sensitive to the size of the PDE cache (the range of the Y-axis is quite small); 32 or 64 entries are enough for the PDE cache. In fact, the maximum improvement of making cache sizes much larger is less by approximately 2% on average (see last column of the figure).

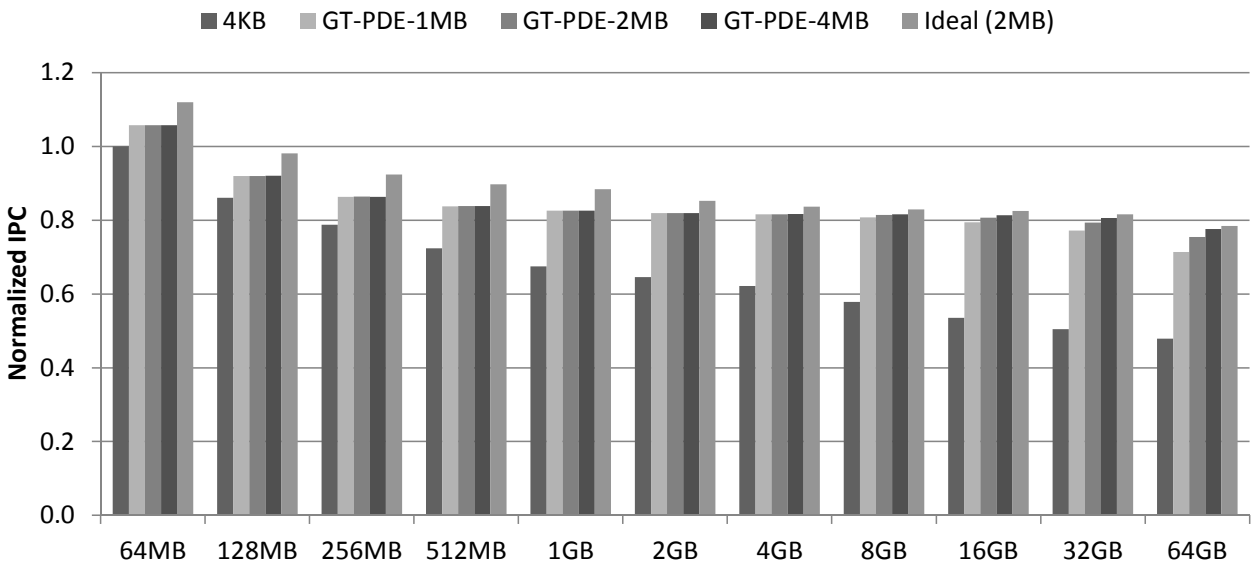


Figure 49: IPC with different problem sizes normalized to a problem size of 64MB using traditional 4KB page.

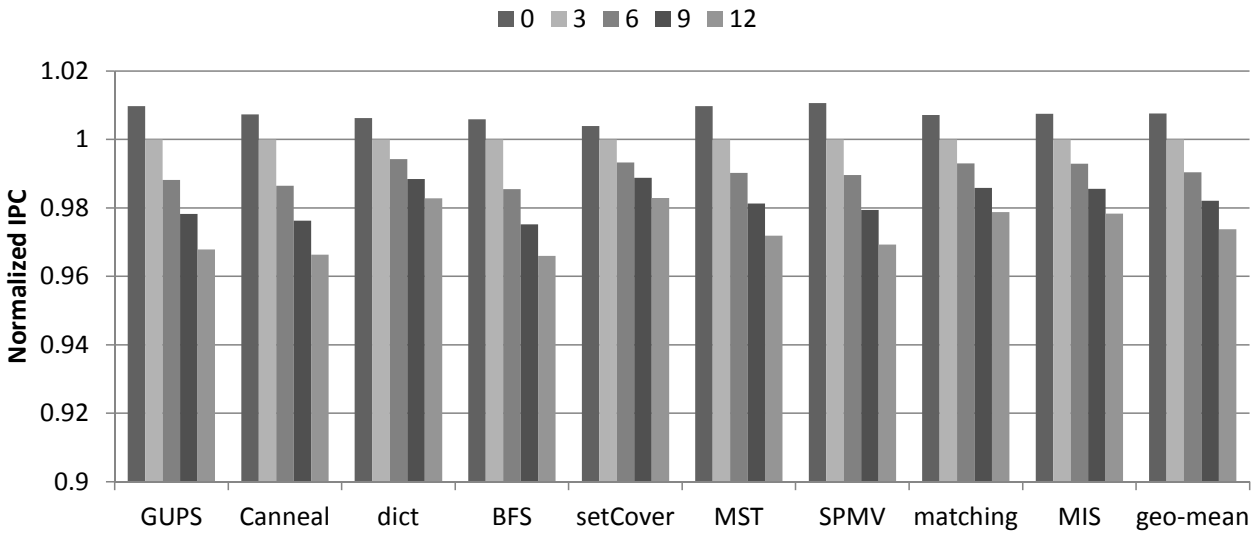


Figure 50: IPC of GT-PDE-4MB with different translation latencies of GT-PDEs normalized to a default latency of 3 cycles.

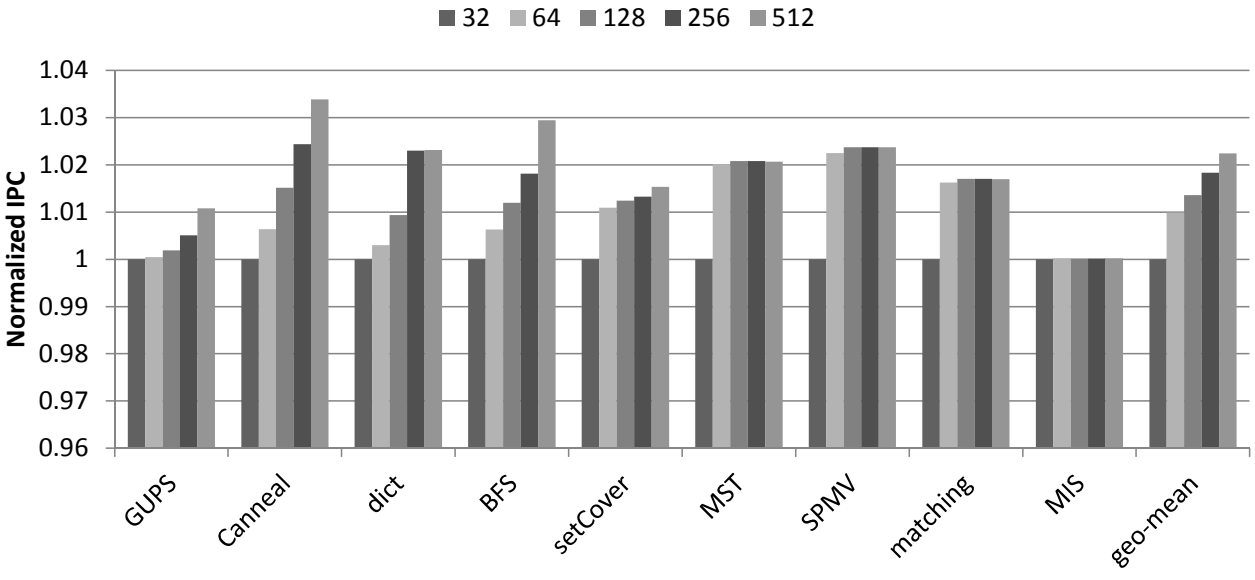


Figure 51: IPC of GT-PDE-4MB with different PDE cache sizes normalized to 32-entry PDE cache.

5.6.7 Comparing to TLB Coalescing

TLB coalescing is a technique which can substantially increase TLB reach by coalescing multiple adjacent PTEs into a single TLB [42, 41]. Note that for my target workloads, that is, those with large memory footprints, the gain of TLB coalescing is not as significant as for smaller applications.

Figure 52 shows the performance comparison between TLB coalescing and GT-PDE. I evaluated the configuration that the 512-entry L2 TLB supports 8x and 32x TLB coalescing, which merges adjacent 8 PTEs and 32 PTEs, respectively. In order to avoid favoring my own scheme, I assume no extra CPU cycles to load L2 TLB entry with TLB coalescing. As shown in the figure, the performance gain with TLB coalescing is virtually non-existent because TLB reach is still limited even with 32x TLB coalescing considering workloads with large memory footprints. For address translations that are missed in the TLB, the dominant performance overhead is from accessing PTEs. To avoid address translation becoming a performance bottleneck, it is critical to eliminate PTE accesses by supporting superpages.

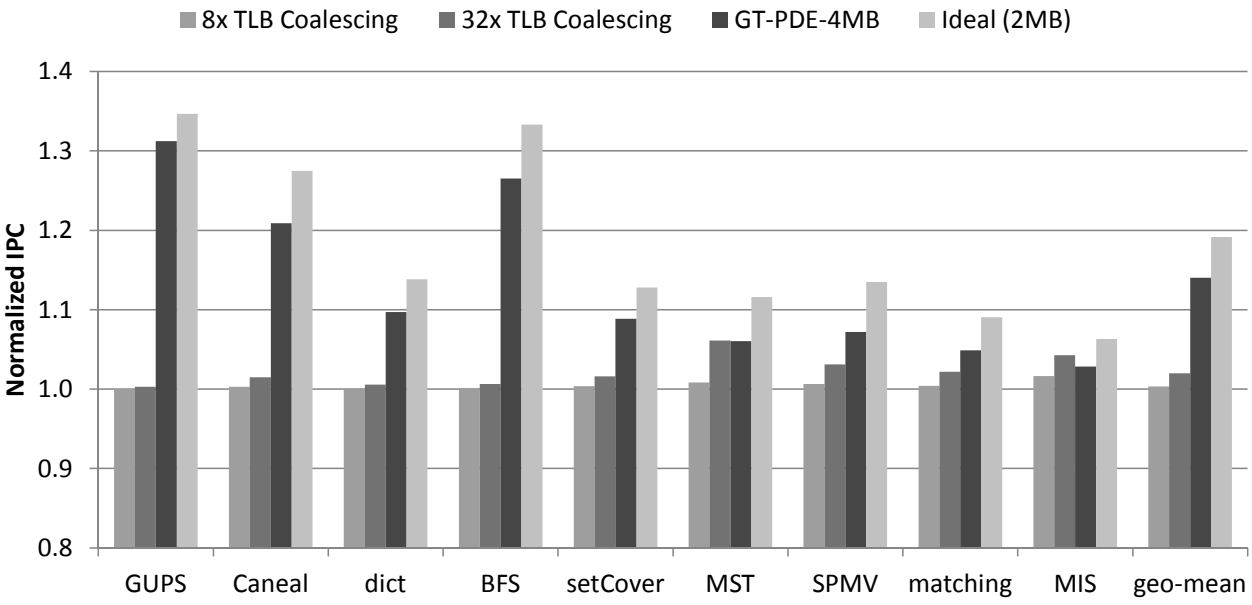


Figure 52: IPC of TLB coalescing and GT-PDE normalized to traditional 4KB page baseline.

5.7 CONCLUSION

Superpages are critical for workloads with large memory footprints. Traditional 2MB superpages are not suitable for memory with retired pages, because a superpage must be mapped to large contiguous physical memory. I proposed gap-tolerant sequential mapping (GTSM) to allow mapping a superpage to memory with retired pages. I proposed GT-PDE which has a block selection bitmap to support GTSM. I also proposed P-GT-PDE, a variant of GT-PDE, which can reduce the size of the page table by 50%.

The proposed scheme is also applies to other memory and not only hybrid memory. When applied to a DRAM-only system, the performance results are better and are presented in [20]. Because the evaluated DRAM/PCM hybrid memory system in this thesis assumes a faster 256MB DRAM cache (30ns) than the commodity DRAM (50ns). Another difference is that in this thesis, I assumes that the L2 cache is 512KB per core which is also larger than the 256KB L2 in [20]. Since the page table is cached in the DRAM cache, the evaluated DRAM/PCM hybrid memory system has lower TLB miss penalty than a DRAM-only system. For example, the average TLB

miss penalty of the GUPS workload is reduced to 113 cycles from 187 cycles. For all workloads, the average IPC improvement of GT-PDE-4MB over the traditional 4KB page baseline is reduced to 12.7% from 19.0% . Though the absolute performance gain is less significant for a DRAM/PCM hybrid memory system than a DRAM-only system, the proposed GT-PDE page table format can close the performance gap between the 4KB page and the ideal 2MB superpaging (i.e., with no retired pages) for both main memory configurations. In the evaluated DRAM/PCM hybrid memory system, the 4MB-page GT-PDE achieves 95.8% of traditional 2MB superpaging, while tolerating memory faults. In the evaluated DRAM-only system, the 4MB-page GT-PDE achieves 96.8% of traditional 2MB superpaging.

6.0 SUMMARY AND CONCLUSION OF THE THESIS

With cloud computing and the rise of big data, there is an urgent need to build large-capacity energy-efficient main memory systems. A DRAM/PCM hybrid memory system is a promising solution to achieve this goal. In my research work, I addressed two challenges that are unique to hybrid memory systems with non-volatile memory. The first challenge is the *limited PCM write bandwidth*, which is a potential performance bottleneck for hybrid memory systems. The second challenge is the *non-contiguous physical memory* due to retired memory pages. Due to limited PCM write endurance, some memory pages will inevitably contain uncorrectable errors and will be retired by the OS. These retired memory pages create unusable "holes" in the physical memory, which makes it difficult to find enough contiguous memory to form superpages. Without the support of superpages, workloads will incur significant performance overhead, specifically for memory-intensive workloads with large memory footprints and random access patterns. This thesis proposes three computer architecture techniques to address the above two challenges.

First, this thesis studied the mapping between program data bits and PCM cells to improve the effective write bandwidth of PCM. It characterizes the distribution patterns of modified data bits inside memory requests: cyclical and cluster patterns. Based on the characterization, I observed an unbalanced distribution of modified data bits among PCM chips which significantly increases PCM write time and hurts effective write bandwidth. This thesis proposes new XOR-based mapping functions to evenly distribute modified data bits to PCM cell groups which significantly improves PCM write throughput. The proposed *double XOR mapping (D-XOR)* reduces PCM write service time by 45% on average, which increases PCM write throughput by $1.8\times$. As error correction (redundant bits) is critical for PCM, I also consider the impact of redundancy information in mapping data and error correction bits to cell groups. To avoid an imbalance distribution of modified bits between data bits and redundant bits, a bit swap function is proposed to extend D-XOR for PCM

with redundant bits. An overall 51% reduction in write service time with D-XOR and swapping leads to a 12% average IPC improvement over Flip-N-Write for a PCM main memory with ECC.

Second, this thesis studies DRAM compression to hold more modified data in DRAM and reduce write traffic to PCM. This thesis proposes a *delta-compression* scheme that is specifically designed to compress only modified data for hybrid memory. The proposed delta-compressed scheme has a much higher compression ratio than traditional memory compression schemes. To improve delta-compression's efficiency and reduce its overhead, this thesis also proposes *selective and predictive compression* that avoids unnecessary delta-compression operations. It also describes a complete example of how to incorporate delta-compression in IBM's MXT memory compression architecture when used for DRAM cache in a hybrid main memory. For the fourteen different memory-intensive workloads that I evaluated, the proposed DRAM delta-compression reduces, on average, the number of PCM writes by 54.3% and improves IPC performance and system energy consumption by 24.4% and 11.0%, respectively.

Third, this thesis studies a new memory page mapping scheme to **construct superpages from non-contiguous physical memory**. The thesis shows that superpages are critical for workloads with large memory footprints and simply increasing the size of 4KB-page TLB is not enough. The thesis describes a new *gap-tolerant sequential mapping* that includes (a) a new page table format, and (b) an access mechanism to support superpages for non-contiguous physical memory. The thesis also describes a new page table compression scheme that uses (a) a variant of the new page table format to further reduce the page table size by half, and (b) a matching algorithm to construct the compressed page table. In comparison to an ideal memory without any retired physical pages, the proposed gap-tolerant sequential mapping, with retired pages, achieves nearly 95.8% of the performance of traditional 2MB superpaging.

There are several possible future research opportunities beyond the work of this thesis. First, this thesis only evaluates DRAM/PCM hybrid memory systems. There are other promising non-volatile memory technologies to be used in hybrid memory systems. *Resistive Random-access Memory (ReRAM)* utilizes the change of the resistance across a dielectric solid-state material to represent data. ReRAM shares the same drawbacks as PCM: limited write bandwidth and write endurance. Similar to PCM, the write service time of ReRAM is affected by the distribution of modified bits[64]. Additional research is required to identify whether we can apply the proposed

techniques in this thesis to DRAM/ReRAM hybrid memory systems. Another promising research direction is new virtual memory systems for hybrid memory systems. Current virtual memory systems are only designed for DRAM-only systems. Future hybrid memory systems are expected to have much larger capacity, and have mechanisms to tolerate memory errors and support efficient data movement among different types of memories. New efficient virtual memory designs are needed to address these requirements and support future hybrid memory systems.

BIBLIOGRAPHY

[1] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, Jun 2004.

[2] Alaa R. Alameldeen and David A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. Technical report, University of Wisconsin-Madison, 2004.

[3] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[4] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Spectlb: A mechanism for speculative address translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, 2011.

[5] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[6] Abhishek Bhattacharjee. Large-reach memory management unit caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, 2013.

[7] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared last-level tlbs for chip multiprocessors. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[8] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core cooperative tlb for chip multiprocessors. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, 2008.

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining (ICDM)*, 2004.

[11] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[12] Sangyeun Cho and Hyunjin 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.

[13] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min Gu Kang, Jaeyun Lee, Yongjin Kwon, Soehee Kim, Jaehwan Kim, Yong-Jun Lee, Qi Wang, Sooho Cha, Sujin Ahn, H. Horii, Jaewook Lee, Kisung Kim, Hansung Joo, Kwangjin Lee, Yeong-Taek Lee, Jeihwan Yoo, and G. Jeong. A 20nm 1.8v 8gb pram with 40mb/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, Feb 2012.

[14] CPU2006. Spec cpu2006: http://www.spec.org/cpu2006/docs/readme1st.html, 2011.

[15] R. Das, A.K. Mishra, C. Nicopoulos, Dongkook Park, V. Narayanan, R. Iyer, M.S. Yousif, and C.R. Das. Performance and power optimization through data compression in network-on-chip architectures. In *Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*, HPCA '08, Feb 2008.

[16] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, 2011.

[17] T. J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory., 1997. IBM Microelectronics Division.

[18] J. Dongarra and P. Luszczek. Introduction to the hpc challenge benchmark suite., 2005.

[19] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the 1993 Winter USENIX Conference*, 1993.

[20] Yu Du, Miao Zhou, B.R. Childers, D. Mosse, and R. Melhem. Supporting superpages in non-contiguous physical memory. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15, Feb 2015.

[21] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, 2005.

[22] Lee et al. Nonvolatile memory device and related methods of operation, 2011. U. S. Patent 7,876,609 B2.

[23] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. Increasing pcm main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[24] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *Proceedings of the Conference on Data Compression*, DCC '96, Mar/Apr 1996.

[25] Narayanan Ganapathy and Curt Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, 1998.

[26] Tiejun Gao, Karin Strauss, Stephen M. Blackburn, Kathryn S. McKinley, Doug Burger, and James Larus. Using managed runtime systems to tolerate holes in wearable memories. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[27] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development*, Jul 1970.

[28] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[29] Intel. Intel 64 and ia-32 architectures developer's manual., 1997.

[30] Intel. The intel xeon processor e7 v2 family, 2014. www.intel.com.

[31] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010.

[32] Lei Jiang, Yu Du, Youtao Zhang, B.R. Childers, and Jun Yang. Lls: Cooperative integration of wear-leveling and salvaging for pcm main memory. In *Dependable Systems Networks (DSN)*, Jun 2011.

[33] I.S. Kim, S.L. Cho, D.H. Im, E.H. Cho, D.H. Kim, G.H. Oh, D.H. Ahn, S.O. Park, S.W. Nam, J.T. Moon, and C.H. Chung. High performance pram cell scalable to sub-20nm technology with below 4f2 cell size, extendable to dram applications. In *VLSI Technology (VLSIT), 2010 Symposium on*, Jun 2010.

[34] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[35] Kwang-Jin Lee, Beak-Hyung Cho, Woo-Yeong Cho, Sangbeom Kang, Byung-Gil Choi, Hyung-Rok Oh, Chang-Soo Lee, Hye-Jin Kim, Joon-Min Park, Qi Wang, Mu-Hui Park, Yu-Hwan Ro, Joon-Yong Choi, Ki-Sung Kim, Young-Ran Kim, In-Cheol Shin, Ki-Won Lim, Ho-Keun Cho, Chang-Han Choi, Won-Ryul Chung, Du-Eung Kim, Kwang-Suk Yu, Gi-Tae Jeong, Hong-Sik Jeong, Choong-Keun Kwak, Chang-Hyun Kim, and Kinam Kim. A 90nm 1.8v 512mb diode-switch pram with 266mb/s read throughput. In *International Solid-State Circuits Conference, 2007. ISSCC 2007*, Feb 2007.

[36] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, Feb 2002.

[37] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for nvm+dram hybrid main memory. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, 2009.

[38] Prashant J. Nair, Dae Hyun Kim, and Moinuddin K. Qureshi. Archshield: architectural framework for assisting dram scaling by tolerating high error rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[39] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Operating System Review*, 36(SI), 2002.

[40] Olga Pearce, Todd Gamblin, Bronis R. de Supinski, Martin Schulz, and Nancy M. Amato. Quantifying the effectiveness of load balance algorithms. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, Jun 2012.

[41] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. Increasing tlb reach by exploiting clustering in page translations. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[42] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach tlbs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[43] Moinuddin K. Qureshi. Pay-as-you-go: low-overhead hard-error correction for phase change memories. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, 2011.

[44] Moinuddin K. Qureshi, Michele M. Franceschini, Luis A. Lastras-Montaño, and John P. Karidis. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[45] Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-Monta no. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture*, HPCA '10, Jan 2010.

[46] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, 2007.

[47] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, 2009.

[48] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[49] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[50] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '11, 2011.

[51] Samsung. Greening ofthe data center: How green memory and ssds impact roi, 2011. www.samsung.com.

[52] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[53] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, 2009.

[54] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin S. Lee. Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[55] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, 2012.

[56] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, May 2011.

[57] Michael Sporer. The power demands of data centers require memory innovations, 2011. www.crucial.com.

[58] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic tlbs for high performance address translation in chip multiprocessors. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, 2010.

[59] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization. In *Lossless Compression Handbook*, 2002.

[60] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, 1992.

[61] Dong Tang, Peter Carruthers, Zuheir Totari, and Michael W. Shapiro. Assessment of the effect of memory page retirement on system ras against hardware faults. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, 2006.

[62] 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.*, 2001.

[63] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[64] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA '15, Feb 2015.

[65] Doe Hyun Yoon, Naveen Muralimanohar, Jichuan Chang, Parthasarathy Ranganathan, Norman P. Jouppi, and Mattan Erez. Free-p: Protecting non-volatile memory against both hard and soft errors. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, 2011.

[66] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, 2007.

[67] Jianhui Yue and Yifeng Zhu. Accelerating write by exploiting pcm asymmetries. In *Proceedings of the IEEE 19th International Symposium on High Performance Computer Architecture*, HPCA '13, Feb 2013.

[68] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. Enigma: Architectural and operating system support for reducing the impact of address translation. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, 2010.

[69] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, 2009.

[70] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.