COLLABORATIVE HARDWARE-SOFTWARE MANAGEMENT OF HYBRID MAIN MEMORY

by

Santiago Bock

B.S. in Computer Science, University of Los Andes, Bogotá, 2007

B.S. in Computer Engineering, University of Los Andes, Bogotá, 2007

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2017

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Santiago Bock

It was defended on

November 3rd 2017

and approved by

Bruce R. Childers, Department of Computer Science

Rami Melhem, Department of Computer Science

Daniel Mossé, Department of Computer Science

Jun Yang, Department of Electrical and Computer Engineering

Dissertation Director: Bruce R. Childers, Department of Computer Science

COLLABORATIVE HARDWARE-SOFTWARE MANAGEMENT OF HYBRID MAIN MEMORY

Santiago Bock, PhD

University of Pittsburgh, 2017

DRAM has long been the preferred technology choice for main memory. With new challenges of high energy and scalability of DRAM, emerging non-volatile memory technologies, such as phase-change memory (PCM), are being considered. Typically, PCM is used in conjunction with DRAM to form a hybrid main memory. Exposing both the PCM and DRAM to the system software and managing it through the operating system (OS) is a viable architecture. The advantage of this organization is that current systems are more easily adapted to support a partitioned DRAM/PCM address space with only small changes to their design. In addition, this architecture is the easiest path forward to incorporate persistence in the main memory hierarchy by reserving part of PCM for storage.

However, the performance of software-managed hybrid memory is not on par with hardware-only approaches, such as the DRAM cache. This is caused by the large granularity at which data is migrated (OS pages) and the low visibility that the OS has of the access patterns of applications. This thesis proposes an experimental framework for studying software-managed hybrid memory and uses it to understand the causes of its low performance. In addition, this thesis proposes and evaluates several hardware-software co-designed mechanisms to alleviate the performance impacts of managing hybrid memory in software. Lastly, this thesis proposes a new migration policy specifically designed to take advantage of the new hardware support. These contributions show that software-managed hybrid memory with specialized hardware support for migration and monitoring is a viable architecture for PCM-based hybrid main memory.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1.0 INTRODUCTION

A growing number of applications that run in today's data centers have very large memory footprints, often exceeding tens of gigabytes [1, 2]. Due to recent advances in microarchitecture and device manufacturing, modern servers have very high core counts (128 cores or more), enabling the execution of multiple applications in the same machine. To cope with the increasing demand in memory capacity, data centers use servers with huge amounts of memory, often in the terabyte range [2].

Due to its relatively good performance and low cost, DRAM has been the prevailing memory technology used in compute servers during the past 40 years. However, the scalability of DRAM is expected to hit a wall in the near future [3]. This will not only increase the cost of DRAM but also make it more vulnerable to soft errors. In addition, the large memory size of current systems is making the high static power consumption of DRAM impractical, especially since most of DRAM's energy is spent by self-refresh operations while the memory is idle [4].

To deal with these problems, researchers have proposed *hybrid memory*, which combines a small amount of DRAM with a large amount of Phase Change Memory (PCM) [5, 6, 7, 8, 9, 10]. The properties of PCM (high density, slow reads and even slower writes, low static power, low read power, and high write power), make it a good candidate for storing large amounts of data that are not updated frequently. However, since most of the data resides in PCM, the bandwidth and latency of hybrid memory systems is limited.

A viable architecture for combining DRAM and PCM is *software-managed hybrid memory*, where both DRAM and PCM are addressable by the CPU and managed by the operating system (OS) [5, 10, 11, 12, 9]. Apart from exposing a larger physical address space to applications, software-managed hybrid memory allows greater flexibility to manage memory

resources because OS policies can be tailored to the needs of application workloads. Importantly, this architecture is the easiest path forward to incorporate persistence in the main memory hierarchy by reserving part of PCM for storage [13, 14, 15].

Nevertheless, the performance of software-managed hybrid memory suffers from several drawbacks, which must be addressed before it can be adopted [16, 17]. First, current commodity systems do not provide hardware to perform page migration efficiently without OS involvement, resulting in frequent interrupts that degrade performance. Second, data is managed at the granularity of pages, which can lead to excessive data movement that steals bandwidth from applications, hurting performance. Third, the OS has low visibility of application access patterns, particularly at the main memory level (below the caches). As a result, the OS cannot react quickly to changes in application behavior, potentially causing poor data migration decisions between DRAM and PCM.

The hypothesis of my thesis is that software-managed hybrid main memory can effectively provide high performance with low hardware cost by using a carefully hardware-software co-designed system that offloads performance-critical tasks to hardware and leaves complex decision-making tasks in software.

1.1 RELEVANCE

In a hybrid memory system, the small DRAM space and the large PCM space are managed by a *migration policy*, which decides whether and when data should be moved between memories. In addition, a migration policy decides what data should be *promoted* to DRAM, and what data should be *demoted* to PCM. Migration policies can be implemented in hardware or in software. An example of a hardware migration policy is found in hardware DRAM caches [8, 6], in which blocks are moved to DRAM on demand. The accessed block is promoted, and the least recently used block is demoted.

In software-managed hybrid memory, the OS initiates the migration of memory pages between DRAM and PCM while applications continue to execute. Pages are not transferred to DRAM on demand. Instead, accesses to pages residing in PCM are serviced directly from

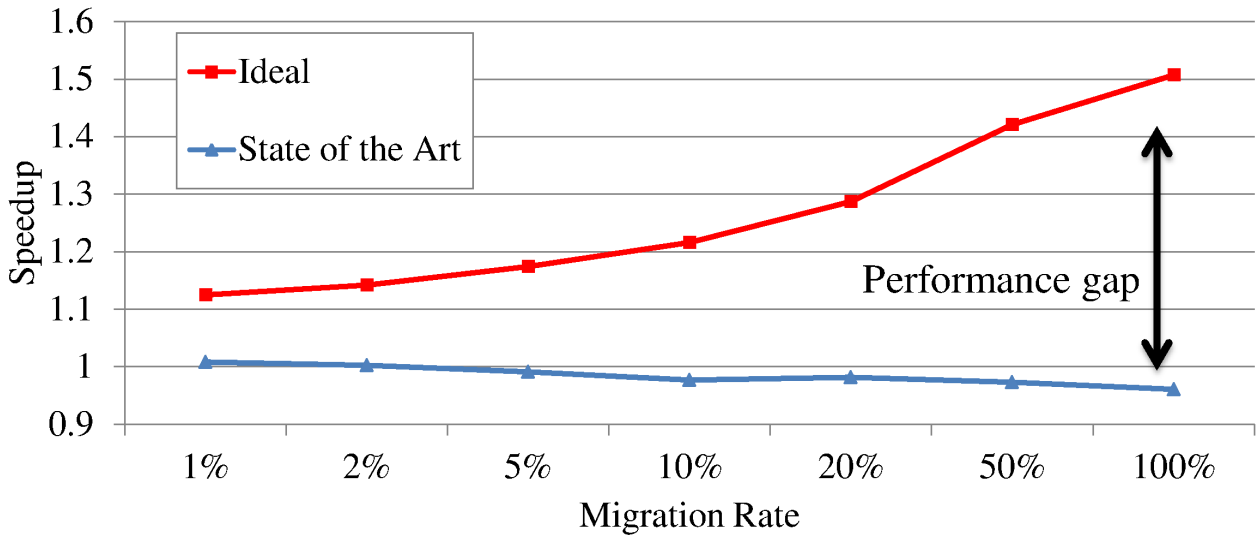

Figure 1: Average speedup over no-migration for different migration rates.

PCM, without moving the page to DRAM. While applications execute, the OS keeps track of memory access patterns and migrates pages in the background. An example of a software migration policy is the Multi-Queue algorithm [9], which ranks pages according to access count, and migrates pages between DRAM and PCM to keep the most frequently accessed pages in DRAM.

Page migration allows the OS to move frequently used data to DRAM and keep less frequently used (colder) data in PCM. This reduces access latency and dynamic power by serving most requests from DRAM. If done correctly, page migration can have a significant effect on performance: Figure 1 compares the state-of-the-art software-managed hybrid memory [9] with an ideal system. The ideal system uses an oracle page migration policy and assumes page migration does not interfere with regular application requests. The graph shows average speedup of both the state-of-the-art and the ideal system relative to a system that does no migration. The no-migration system allocates pages on first touch to DRAM until the DRAM capacity is exhausted. At this point, subsequent pages are allocated to PCM. The x-axis shows *migration rate*, which is the fraction of execution time during which

the OS migrates pages. At low rates, the state-of-the-art system performs close to the baseline. At high rates, however, it does worse due to interference between application and migration memory requests. An ideal memory system with no interference and a "good" migration policy can achieve high performance, especially at high migration rates.

1.2 ELEMENTS OF A GOOD SOLUTION

A good solution for software-managed hybrid memory must satisfy a number of requirements. The first requirement is that the time overhead of managing hybrid memory in software should be low. If the overhead is too high, this approach will simply not be able to compete with a hardware approach (DRAM cache) and will therefore not be adopted. Since managing hybrid memory entirely in software is potentially time consuming, a good solution must include some hardware support to accelerate certain operations. Therefore, the system should use hardware for simple, performance-critical operations, and perform more complex ones in software.

The second requirement is flexibility of choice in the migration policy. Each application has a preferred policy that maximizes its performance, and the preferred policy for each application is different. To maximize overall performance, each application should be able to choose its own migration policy. This flexibility also enables the development of migration policies specifically tailored to an application.

The third requirement is that the additional hardware support be as small as possible while still allowing the system to perform well and be flexible enough to be able to use most migration policies. Making only small, simple hardware changes means that hardware manufacturers will be more likely to include them in future processors, increasing the chances of adoption.

1.3 APPROACH OVERVIEW

In this research, I propose a memory management system for hybrid DRAM/PCM architectures. The proposed system uses hardware-software co-design to separate performance-critical tasks from other tasks that can be implemented in software without high overhead. By co-designing software and hardware, my approach provides a flexible substrate for the OS to implement custom policies.

The proposed system manages memory at the software level with small hardware support for some performance-critical operations. Data migration decisions between DRAM and PCM are performed by the OS based on monitoring information collected by hardware. The actual migration of pages is performed by hardware once the OS has selected the source and destination.

Figure 2 shows an overview of the proposed system. The system has two main new components. The Hybrid Memory Manager (HMM) is a software module that is part of the OS. The Hybrid Agent (HA) is a hardware component that is part of the chip multiprocessor.

The HMM is the central component. It is in charge of coordinating all actions of the system, including communication with other modules (software and hardware). The HMM uses one or more migration policies. In general, each application has its own migration policy tailored to its specific memory access patterns.

The migration policy relies on monitoring information. This information is stored in the Monitoring Information (MI) component, and can be accessed directly from software. However, the information is not collected by the OS. Instead, a hardware module, called Memory Access Monitor (MAM), collects this information and passes it to the MI.

The other main new component of the system is the HA, which is composed of the Migration Manager (MM), the Memory Redirector (MR), and MAM. The MR is in charge of deciding where memory requests should be directed to (PCM or DRAM), based on their physical address. The MR is aware of ongoing page migrations, and can redirect memory requests of pages under migration to the correct location. The MR also notifies the MAM of memory requests. The MAM updates its data structures with the new information, which

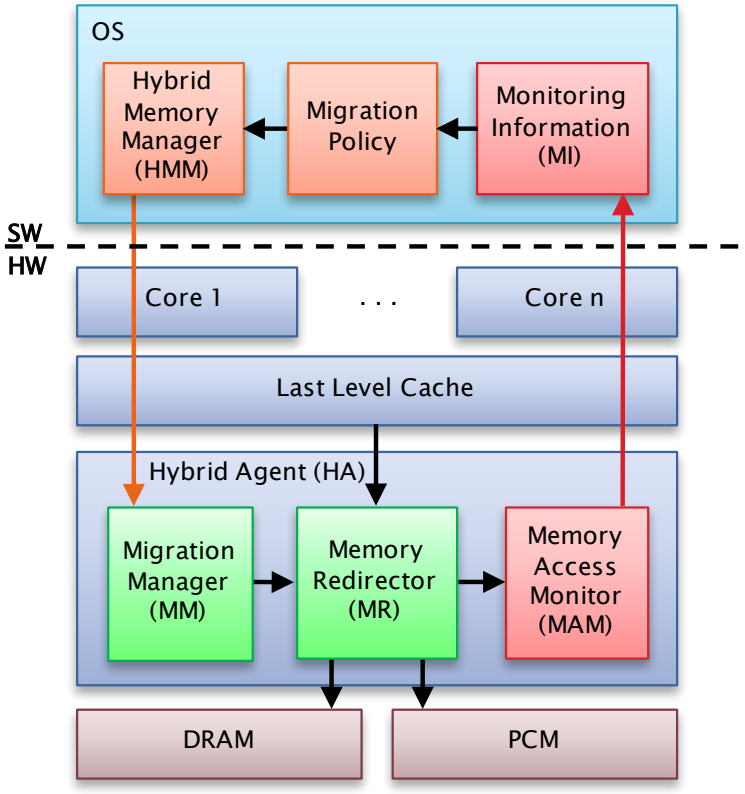

Figure 2: Overview of the proposed system.

is later stored in the MI for use by the migration policies.

When the HMM decides to migrate one or more pages, it communicates with the MM. The MM is in charge of coordinating all hardware actions necessary for performing page migrations, including copying pages between memories, updating page table and TLB entries and flushing the caches. The MM also notifies the MR of page migrations, so that the MR can update its data structures.

To characterize the performance bottlenecks of hybrid memory and evaluate the proposed architecture, I developed a new simulation infrastructure specifically tailored for software-managed hybrid memory. Although the infrastructure is capable of simulating a wide range of systems, the focus of this thesis is on mobiles devices, such as smartphones and tablets. These devices can benefit greatly from the low energy consumption of PCM-based hybrid

memory.

1.4 THESIS CONTRIBUTIONS

This thesis contributes new hardware mechanisms and software algorithms for enabling software management of hybrid main memory. This approach offers an alternative to hardware-only hybrid memory, such as DRAM caches. This thesis opens up new research opportunities for developing migration policies, which can be tailored to the access patterns of applications. In addition, it provides an evaluation infrastructure that can be easily adapted to new policies.

This thesis makes the following contributions:

- A simulation infrastructure to evaluate software-managed hybrid main memory is developed. This includes components to simulate migration policies, which model the software side of the proposed system. Furthermore, the infrastructure includes a detailed model of caches, memory devices (DRAM and PCM) and specialized hardware for hybrid memory [18].
- A characterization study to determine the most important factors that limit performance in software-managed hybrid memory is presented. This study is the building block that guides the development of the proposed mechanisms of this thesis [17].
- Two hardware mechanisms for supporting page migration in hardware to reduce OS overhead are proposed. The first mechanism prevents applications from pausing when they write to a page currently under migration. The second mechanism extends and generalizes the first one to allow for multiple simultaneous migrations [16].
- A hardware-software co-designed mechanism to reduce the bandwidth consumption of software-managed hybrid memory is proposed. This design corrects the main drawback of multiple simultaneous migration (high migration bandwidth) by preventing unnecessary movement of data [19].
- A hardware-software co-designed mechanism for monitoring memory access patterns is proposed. This mechanism requires a small amount of hardware resources but provides

enough monitoring information to make good migration decisions. The design is flexible enough to accommodate a wide range of migration policies [19].

- A new migration policy designed specifically to be used with the new migration and collection mechanisms is proposed. This policy requires a small amount of processing power in the OS but provides better page placement than the current state of the art.

1.5 THESIS ORGANIZATION

The rest of this thesis is organized as follows. Chapter 2 presents background and related work including PCM, architectural support for PCM and hybrid memory. Chapter 3 introduces the simulator infrastructure developed specifically to address the challenges of evaluating hybrid memory. Chapter 4 presents the characterization study that guides the development of the proposed mechanisms. Chapters 5 and 6 propose two mechanisms for performing migration in hardware, including the proposed scheme for bandwidth reduction and the access monitoring mechanism. Next, Chapter 7 presents the new migration policy. Finally, a summary of this work and future research directions are presented in Chapter 8.

2.0 BACKGROUND AND RELATED WORK

2.1 PHASE CHANGE MEMORY

Phase Change memory (PCM) [20, 21] is a non-volatile memory technology that stores information by changing the physical state or phase of chalcogenide material. PCM works by applying electrical currents of different intensity and duration to a small volume of phase-changing material. These varying currents melt the material and let it cool at different rates, which determine its final phase. When the material cools quickly, ions do not have enough time to form a lattice, and the material ends up in an amorphous phase. Conversely, when the material cools slowly, ions form a crystalline structure.

The resistance of the phase-change material depends on its physical state. When the material is in a crystalline phase, electrons can move more freely within the material because they encounter less resistance from aligned atoms. Hence, the resistance of the material is low. Conversely, when the material's phase is amorphous, electrons move more slowly because of the increased friction caused by randomly placed atoms. When amorphous, the resistance of the material is high.

Changing the phase of chalcogenide material is a complex process that involves carefully controlling the profile of the current that is applied to each cell. Due to this complex process and to the slow cool-down period when changing the state to crystalline, writing to PCM is orders of magnitude slower than writing to DRAM.

Because PCM stores information by changing the configuration of atoms inside the material, the energy required to change the stored value is higher than in DRAM, which maintains information by changing the amount of stored charge. This fact has two important consequences. First, the state of PCM cells persist for longer periods of time. Since there is no

Table 1: Comparison of DRAM and PCM Parameters

	DRAM	PCM
Read Latency (ns)	13.7	60
Write Latency (ns)	12.5	150
Read Energy (pJ/bit)	1.17	2.47
Write Energy (pJ/bit)	0.39	16.82
Endurance (writes)	10^{15}	10^{8}

need to periodically refresh the contents of memory, PCM requires very little energy when idle, and can be used as non-volatile storage. Second, actually changing the state of the material requires more energy than simply moving electric charge. Hence, writing to PCM is more energy-consuming than writing to DRAM.

A major disadvantage of PCM is that its cells lose the ability to change their physical state after they have been written a large number of times. This is caused by detachment of the phase-change material from the electrodes that provide the programming current, and it is generally irreversible. Hence, the endurance of PCM is limited to between 10^6 and 10^9 write cycles.

In summary, PCM is slower than DRAM, especially when writing. Dynamic energy consumption is also higher than DRAM, also especially when writing. Static energy consumption, on the other hand, is lower in PCM than DRAM. Table 1 shows a comparison of PCM and DRAM, based on values from a recent paper [22].

2.2 ARCHITECTURAL SUPPORT FOR PCM

Due to its characteristics, PCM cannot be used as a direct DRAM replacement without considerable performance, energy and lifetime penalties. Researchers have proposed several

mechanisms that provide architectural support for enabling PCM in main memory. The following subsections describe some of these mechanisms. Hybrid main memory, the primary mechanism for supporting PCM, is described in Section 2.3.

2.2.1 Write Reduction

One of the main drawbacks of PCM is that writes are costly in terms of performance, energy consumption and lifetime. Therefore, reducing the number of writes that PCM devices sustain has a direct impact on energy consumption and device lifetime. In addition, reducing the amount of time the memory spends on slow write operations increases the average read time due to higher availability of the device.

Lee *et al.* propose tracking dirty data in the processor caches at the granularity of a word [22]. Upon a LLC write-back, only the dirty words within a cache line are actually written to the PCM device. The main advantage of this technique is that no comparison between new and old data is necessary to determine which bits can be safely ignored during write-backs. However, this is done at the expense of additional on-chip area (3.1% for 4-byte words and 64B-byte cache lines). This technique can improve memory lifetime by up to 8 times compared to keeping track of dirty data at cache line granularity.

Bock *et al.* propose a software technique for identifying dead blocks in the LLC or in a DRAM cache using calls to the memory allocator [23]. A dead block contains data that will not be read before it is written again. Hence, dead blocks can be discarded without affecting the correctness of the program. When a dead block is evicted from the cache, its contents are not written back to the next level of the memory hierarchy (the PCM). This technique reduces the number of writes to PCM and reduces energy consumption.

Ferreira *et al.* propose a technique called Read-Write-Read (RWR) to determine exactly which bytes of a write operation have been modified [6]. RWR works by reading the original contents of PCM before issuing a write operation, and comparing the old and new versions of the data. Only modified data is written to PCM. This technique can indeed reduce the number of writes to PCM. However, this is done at the expense of one additional read before every write. Lifetime savings depend on how much data is modified while it is in the cache

hierarchy. If data is heavily modified, the lifetime will be close to that of the baseline system. Energy savings also depend on how much each write benefits from RWR. If the energy spent in reading the data from PCM is not offset by writing fewer data, the net energy savings will be negative.

A similar technique, called Data-Comparison Write (DCW), was proposed by Yang *et al.* [24]. Like RWR, DCW also compares stored data with the new version of the data. The main difference with RWR is that DCW operates at the device level. Hence, the granularity of data comparison can be much smaller (DCW's granularity is the bit). On average, DCW can reduce the number of written bits by half, although the actual value depends on the data being written.

Flip-and-Write builds upon DCW by adding an additional bit to each word of stored data that keeps track of whether the word is stored inverted [25]. When writing new data, the bit-by-bit comparison counts how many bits must be actually changed from their current state. If more than half of the bits need to be changed, the data is stored inverted and the new bit indicating this is set. Flip-and-write guarantees that at most half of the bits in each write are actually changed.

2.2.2 Write Cancellation and Write Pausing

One of the disadvantages of PCM is its slow write latency, which can be between 2 and 4 times slower than the read latency, and 10 times slower than DRAM's write latency. In general, memory write latency is not exposed directly to an application's execution time because the completion of a write is not required by the CPU to continue execution (as is the case for reads). Write buffers can store writes and delay them until the memory becomes idle. Once a write request has been issued, however, subsequent read requests must wait for the write to finish, exposing additional latency to the processor. With slow PCM writes, this additional latency can have a considerable impact on performance.

To address this issue, Qureshi *et al.* propose two techniques that allow reads to preempt ongoing writes, reducing the latency of reads as seen by the CPU [26]. *Write cancellation* simply cancels ongoing writes, allowing reads to start executing earlier. Writes are only

canceled when they are less than 80% completed. The original write is scheduled to execute later, when memory resources become available or when write buffer utilization exceeds a predefined threshold.

The disadvantage of write cancellation is that canceled writes waste memory cycles because the write needs to be performed again, and previous work performed on the canceled write is lost. This wastes memory bandwidth because the average write time increases. To avoid this problem, the authors also propose *write pausing*, which allows the memory to remember the progress of preempted writes so that they can be restarted closer to when they were interrupted. This technique takes advantage of PCM's iterative writing process, in which the current state of the device is compared to the desired state [27]. After several cycles of writing and comparing, the desired state is achieved and the process finishes. Write pausing leverages this iterative process to reduce the time wasted due to restarts of writes.

2.2.3 Wear Leveling

Another important disadvantage of PCM is its low write endurance. Current PCM devices can sustain between 10^5 and 10^9 write cycles [28, 29, 30]. When used in main memory, PCM devices have a lifetime ranging from a few months to several years, depending on the write rate and the memory access patterns of applications. Wear leveling can extend the lifetime of PCM devices by uniformly distributing the writes across all memory locations within a device. This prevents the memory from failing due to repeated writes to a few memory locations, which can happen in a short period of time. Several wear leveling mechanisms have been proposed to address the issue of limited write endurance in PCM.

Zhou *et al.* propose two wear leveling schemes that work at different data granularities [31]. *Row shifting* distributes the writes within a device row by shifting or rotating the contents of the row by a specified amount in increments of one byte. The shifting amount is stored together with the row so that the data can be correctly reconstructed when the row is read. The second mechanism, called *segment swapping*, distributes writes across 1MB segments by introducing an additional level of indirection that maps the address generated by the core to the address actually used in the device. A table is used to keep this maping,

as well as metadata about the number of writes to each segment and the last time it was swapped. The wear leveling algorithm periodically swaps hot and cold segments to ensure that each segment has a similar number of writes.

Although wear leveling at the page or 1MB segment granularity does extend the lifetime of PCM devices, the overhead of updating the mapping table on every write and of searching through the table to find swapping candidates can be very high. To address this issue, Ferreira *et al.* propose a similar mapping scheme that chooses the swapping candidates randomly [32]. This avoids the overhead of updating the access counts and of searching the hottest and coldest segments at the expense of reduced lifetime. However, for current devices and applications, it is enough to guarantee several years of operation.

Start-gap wear leveling uses algebraic mapping between logical and physical address to avoid the mapping table, which consumes area and energy, and adds latency to each memory request [33].

2.3 HYBRID MAIN MEMORY

The problem of using PCM in main memory has been widely studied in recent years. Because of its slow performance, high energy consumption and low endurance, PCM has not been considered as a direct DRAM replacement. Instead, researchers have proposed using it together with DRAM in a hybrid memory architecture, where a small DRAM is combined with a large PCM.

There are traditionally two approaches for architecting a hybrid memory system. In hardware-managed hybrid memory, PCM is invisible to software, and hardware is entirely responsible for managing data and providing the correct data to applications. In software-managed hybrid memory, the OS is aware of both DRAM and PCM memory spaces, and it manages data by allocating memory pages and initiating their migration. A discussion of these two approaches follows in the next sections. A third approach which combines both hardware and software mechanisms, is the focus of my thesis.

2.3.1 Hardware-Managed Hybrid Memory

In hardware-managed hybrid memory, the OS has a homogeneous view of memory and is not aware that the physical memory is composed of DRAM and PCM. Hardware decides where a particular block of data is placed, and keeps track of where each block is using hardware structures. When a memory access misses in the LLC, the hardware consults this hardware structure to retrieve the physical address of the requested block, and forwards the request to the appropriate location.

2.3.2 DRAM Caches

The simplest organization of a hardware hybrid memory is the DRAM cache [8, 6]. In this architecture, hybrid memory is organized as a non-inclusive set-associative hardware cache that is accessed after the LLC. Cached blocks are stored in DRAM, and the rest are in PCM. A tag array keeps track of which blocks are currently in DRAM. Upon an access, the tag array is first queried to determine whether the requested address is cached. If it is, the address of the block is constructed based on the original address and on the number of the way that hit in the cache, and the requested is forwarded to the correct DRAM address. If it is not, the request is forwarded to the original address in PCM.

In hardware DRAM caches, the size of the block used in the cache has several important consequences on performance and other design considerations. First, the tag array area depends on the block size. If the block size is too small and the DRAM size too large, the area required to store all tags can become impractically large. For example, a 1GB DRAM cache requires a 96MB tag array when using 64-byte blocks [36]. Second, the block size has a big impact on PCM bandwidth utilization. When using large blocks (for example, 4KB), the amount of data that is transferred between DRAM and PCM while servicing requests and writing dirty data back can saturate the available bandwidth of PCM. This causes the memory system to slow down considerably, hurting performance. Third, exploiting spatial locality is less effective when blocks are small, as less data is pre-fetched into DRAM on a miss.

To deal with the area overhead of small blocks, researchers have proposed various meth-

ods. Sub-blocking uses large blocks but keeps individual present and dirty bits for each sub-block [6]. Upon a miss, only the requested sub-block is brought into DRAM. Since there are less blocks to keep track of, the required tag array area is smaller. However, sub-blocking is not as efficient as using smaller blocks in terms of capacity management because part of the DRAM space and the tag array capacity is wasted. Sub-blocking also reduces bandwidth utilization because not all parts of a block are brought into DRAM.

Another technique to reduce the area overhead of small blocks is to keep tags in DRAM instead of in the tag array. Upon a memory request, the DRAM is accessed twice: once to read the tag array and once to get the actual data. Since accessing DRAM is considerably slower than on-chip SRAM tag arrays, the latency overhead of this architecture is usually too high to be practical. Compound scheduling partially solves this problem by co-locating the tags and the corresponding data in the same DRAM row [36]. When accessing memory, the tag for that particular set is retrieved from the DRAM row, and the row is left open. Upon a hit, the data is read from the still-open row buffer.

Other techniques have been proposed to deal with the overhead of keeping tags in DRAM. A miss map can be used in conjunction with compound scheduling to reduce the latency of misses and bandwidth utilization [36]. An on-chip SRAM miss map keeps track of which blocks may be in DRAM but does not keep perfect information, which reduces area requirements. The miss map does not know whether a block is definitely in DRAM, but does know whether one is definitely not there. Therefore, miss maps can reduce the miss penalty and bandwidth consumption of DRAM caches that keep tags in DRAM.

Another technique for reducing the latency overhead of DRAM tags is to use a direct-mapped cache [37]. When using caches with large associativities, the number of blocks that need to be read from DRAM to perform the tag checks is high. For example, 3 blocks need to be read to check the tag array of a 29-way set-associative cache with 64-byte blocks [37]. Using a direct-mapped cache means fewer blocks need to be accessed and transferred. This reduces the latency of both hits and misses at the expense of a higher miss rate. With large cache sizes, which have low miss rates, the average access latency is reduced.

Finally, the footprint cache allocates data at the granularity of pages, but fetches only the cache blocks that will actually be used by the CPU while the page is in the DRAM

cache [38]. To determine which blocks should be fetched, this scheme keeps track of blocks while the page is in the cache. Upon eviction, this information is stored in a predetermined location in memory. When the page is brought back to the cache again, only those blocks that were previously touched are transferred to the cache.

2.3.3 Smart Memory Controllers

Ramos *et al.* describe a sophisticated memory controller that ranks pages based on the frequency and recency of memory accesses using the Multi-Queue algorithm [9]. The hardware is able to perform migrations based on this information without involving the OS. Hardware structures keep track of which physical addresses have had their data moved to a new location. When these hardware structures are full, the OS is notified of page migrations, and it updates its own data structures accordingly.

Although the OS is aware of both DRAM and PCM address spaces, this scheme is still categorized as a hardware hybrid memory because the memory controller performs all of the operations required for migrating and ranking pages. The OS does not implement migration policies or performs migrations. It is simply aware that the hardware can change the location of data depending on memory access patterns.

2.3.4 Software-Managed Hybrid Main Memory

In software-managed hybrid memory, DRAM and PCM have non-overlapping physical address ranges which are directly accessible from the CPU (via virtual-to-physical mappings). The OS is aware of this separation and must choose which type of memory to assign to each virtual page that it allocates. Hardware determines the type and location of a memory request solely by its physical address and the address ranges of DRAM and PCM, without consulting hardware structures.

The OS can change the type of a virtual page by migrating it to a new memory location. To do this, the OS must pause the application (or execute until the virtual page is written to, then pause), copy the physical page to the new location, flush the page out of all caches and update the page table and TLB entries of the virtual page. This is generally a costly op-

eration, although its overhead can greatly be reduced by using specialized hardware support for migration.

From the perspective of the OS, managing hybrid memory consists of deciding what data should be in DRAM and what data in PCM. For most workloads, the available DRAM space is not enough to hold all data. Therefore, some data must be kept in DRAM and some in PCM. Ideally, the OS should try to keep the most frequently used data in DRAM. However, this is not easily accomplished, for several reasons. First, the set of most frequently used data changes over time as applications enter other phases of execution. Second, actually determining the set of most frequently used pages is difficult because the OS cannot record and keep track of every memory access. Third, page migration is costly in terms of latency to copy the page and how it affects the latency of other memory requests. Therefore, the OS can not blindly migrate pages constantly because this can affect other requests and can waste precious memory bandwidth.

A *migration policy* is a set of rules or an algorithm that determines what pages are migrated between DRAM and PCM and when they are migrated. A migration policy must constantly determine when to migrate data, what data to move from DRAM to PCM (demotion or eviction), and what data to move from PCM to DRAM (promotion). This is similar to the decisions made by cache replacement policies or paging algorithms, which must decide on what cache block or page to evict from the cache or from memory. A migration policy is different in that, in addition to selecting an eviction candidate, it must also decide whether to migrate a page at all and select a promotion candidate. In a cache replacement policy or paging algorithm the decision of when to migrate is done tacitly (migrate when there is a cache miss), as well as the decision of what to promote (promote the page that missed in the cache).

A major advantage of software-managed hybrid memory is the flexibility in choosing a migration policy. Since it is implemented in software, a new migration policy can be easily deployed and tuned for a particular application. Researchers have proposed a number of policies, each aimed at solving a particular problem of using PCM in main memory [10, 5].

3.0 SIMULATOR INFRASTRUCTURE

While researching new hardware mechanisms and migration policies for software-managed hybrid memory, it is important to have the appropriate tool for the job. One of the main challenges of researching software-managed hybrid memory is the need to model different parts of a computer's hardware and software hierarchy. For instance, a simulator must be capable of modeling details about a computer's processor, cache hierarchy and memory system, and at the same time be able to simulate the migration policies that would run as part of the OS in a real system. Another challenge is the trade-off between accuracy and performance of the simulation. When prototyping new migration policies, it's advantageous to be able to run simulations and obtain results quickly. However, abstracting away too many details in the simulator can lead to inaccurate results. Another challenge of simulating software-manage hybrid memory is the tool's ease of use. When researching new policies, it's important to be able to implement them easily without major changes to the simulator or the simulated OS. In addition, it must be possible to implement different types of policies without being limited to a particular family of policies.

To aid in the design and evaluation of new hardware mechanisms and migration policies for software-managed hybrid memory, I created HMMSim, a trace-driven simulator for hardware-software co-design of hybrid main memory [18]. HMMSim is capable of simulating DRAM and PCM in several hybrid memory organizations, including single-memory systems, DRAM cache, and software-managed hybrid memory. HMMSim simulates the entire memory hierarchy, including the load store units at the CPUs, caches and their queues, main memory controllers, queues, banks and buses, as well as interactions caused by OS page management in the hierarchy (e.g., flushing of pages from the caches after migration). HMMSim can also emulate the behavior of the OS related to page migration as well as different migration

policies.

There are many memory simulators available, each with its own degree of accuracy, performance, flexibility and ease of use. However, none of them is well suited for modeling both hardware and software in a flexible and extensible way. Table 2 shows a comparison of features of three of the most popular memory simulators. Both DRAMSim2 and USIMM provide detailed DRAM models but do not support PCM. NVMain supports PCM and hybrid memory. However, using it to simulate a software-managed system, including the complete hierarchy and OS emulation, requires using and potentially modifying another simulator such as gem5 [43]. One of the contributions of HMMSim is that it provides this capability in a single tool that abstracts away the details of the OS, allowing for easy changes to migration policies.

This chapter describes the architecture of HMMSim and details about some of the new techniques used in the simulator. It describes the Application Programming Interface (API) that HMMSim provides for extending functionality to model new mechanisms. Lastly, this chapter reports figures about the performance of HMMSim and show that it is fast and scalable.

The next section provides an overview of the HMMSim software architecture, and describes the design of the simulator and implementation choices.

3.1 OVERVIEW

Figure 3 shows the architecture of the simulator. Traces are gathered with Pin [47] and stored in a high compression format. The configuration of the simulated system is read from a file that specifies the value of each parameter. There are over 100 configurable parameters in HMMSim, most of which can be easily set from processor and memory specifications. The simulator includes several components that model the CPUs, caches, DRAM and PCM memory, OS memory management and migration policies. Each component can define *statistics*, which are registered with a centralized gathering component that writes the value of each statistic at the end of simulation. The discrete event simulation engine allows any object

Table 2: Comparison of some existing memory simulators

Feature	**DRAMSim2** [44]	**USIMM** [45]	**NVMain** [46]	**HMMSim**
DRAM	Yes	Yes	Yes	Yes
PCM	No	No	Yes	Yes
Hybrid Memory	No	No	Yes	Yes
Software-Managed Hybrid memory	No	No	No	Yes
Complete Hierarchy	No	No	No	Yes
OS Emulation	No	No	No	Yes

to schedule an event in the future and receive a callback when the simulation reaches the specified cycle.

3.2 MEMORY HIERARCHY

The memory hierarchy is composed of three main components: CPUs, caches and memory. The CPU reads entries from the trace reader and recreates the instructions executed during trace collection. Each instruction consists of an instruction memory access and zero or more data memory accesses. The CPU first sends the instruction memory access to the level 1 instruction cache (I-L1) and waits for a response. Once the instruction access comes back, the CPU sends the data requests to the level 1 data cache (D-L1). The CPU tracks in-flight instructions with a data structure similar to a reorder buffer (ROB) that only tracks memory operations. The CPU retires instructions in order after all data reads have completed. Multi-core systems are modeled by creating multiple CPU objects, each connected to its own trace reader.

The simulator models the cache hierarchy by connecting several cache objects to form a

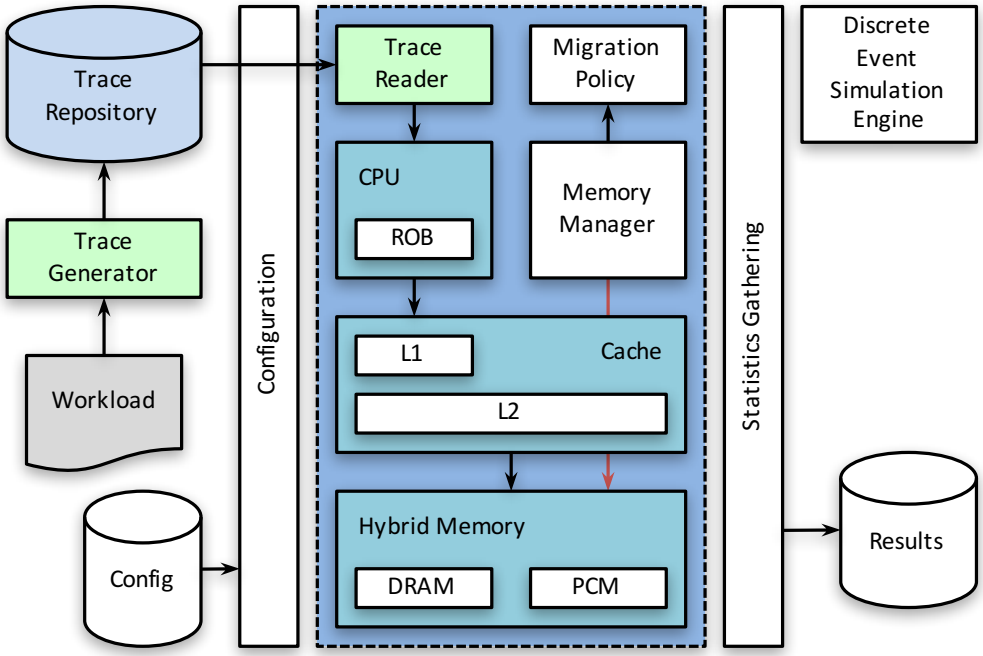

Figure 3: Overview of Hybrid Main Memory Simulator

multi-level cache that can be configured with any number of private or shared levels. Cache objects receive requests from the CPUs or from previous levels in the hierarchy. On a cache hit, the requested data is sent back to the previous level. On a miss, the cache forwards the request to the next level (another cache level or memory). Each cache has a queue that limits the number of requests being serviced at this or lower levels. If a queue is full, requests from previous levels are stalled until a slot becomes available.

HMMSim simulates memory through a set of configurable objects that can model either DRAM or PCM accessible through a DDR interface. The model includes multiple banks, row buffers, per-bank or global queues, a bus and a scheduler, as well as support for different address mappings and row buffer policies. HMMSim also provides a hybrid memory controller that redirects requests to either DRAM or PCM based on physical address, migrates pages and collects monitoring information.

The flow of simulation starts at the CPUs and proceeds down the hierarchy through the caches and to memory. Each component of the hierarchy that receives a request must either

return a response to the previous level or forward the request to the next level. Components model internal delays (such as tag access latency or bank operations) by scheduling events with the simulation engine. When a request is satisfied, the response is sent to the CPU by calling back each component in the hierarchy that forwarded the original request until the response reaches the CPU.

3.3 MEMORY MANAGER

The purpose of the memory manager is to translate a virtual address (collected in the trace) to a physical address that is used by the memory hierarchy. The memory manager follows an *allocation policy* for assigning virtual pages to DRAM or PCM. HMMSim supports several allocation policies, including round robin and random.

In software-managed hybrid memory, the memory manager must also perform page migration. The manager orchestrates all the necessary actions related to page migration, including changing address translation maps, copying the data, flushing the caches and preventing applications from accessing pages under migration. A *migration policy* that is invoked by the memory manager decides whether to migrate pages and what pages to migrate. A policy uses monitoring information retrieved from hardware to make migration decisions, as well as other information available to the emulated OS, such as page type information and offline access counts. HMMSim offers several migration policies, including Multi-Queue and Oracle, and allows for easy creation of new policies.

3.4 HMMSIM API

HMMSim provides a C++ API to extend the functionality of the simulator. Table 3 lists the main components, interfaces and the methods offered. The API has a method for scheduling an event that triggers a callback after the specified amount of cycles. All objects that schedule events have a reference to a singleton `Engine` object and must implement the `IEventHandler`

interface. I provide methods for registering and resetting statistics, and for creating counters that trigger interrupts after they reach a threshold (e.g., for counting executed instructions). Memory hierarchy components must implement the `IMemory` and/or the `IMemoryCallback` interface so that they can be connected with other components. In addition, the caches implement the `IFlushable` interface and objects that issue flush requests (such as caches and the hybrid memory manager) must implement the `IFlushCallback` interface to be notified when the flush completes. Lastly, HMMSim has an interface for different memory managers (single memory, hybrid) and arbitrary migration policies.

3.5 TRACE COMPRESSION

Since HMMSim simulates memory accesses as they traverse the entire memory hierarchy, traces must collect memory accesses before the caches. Storing traces of a few seconds of native execution requires significant storage capacity, even when traces are compressed.

To solve this problem, I created a special trace format that splits the contents of the original traces into various sub-traces and compresses them individually. This results in a higher compression ratio because entries within each sub-trace are similar. Figure 4 shows the trace compression format. Each entry in the original trace is encoded in binary (to achieve higher compression) and contains the type, address size and timestamp (sequence number) of the memory access. The trace is first split into 3 sub-traces according to the entry type. Note that the entry type no longer needs to be stored: it is implicit in the name of each sub-trace. Each sub-trace is further divided into 3 sub-traces, each containing either the address, size or timestamp. The timestamp is delta encoded (the value is the difference between this and the previous entry) to reduce the number of possible values to store, increasing the compression ratio.

HMMSim has several command line tools for manipulating traces. A text converter can be used to output the trace in text format for analysis. The converter can also be used to convert a trace stored in another format into HMMSim's own format. In addition, there are tools for analyzing the contents of the trace directly, without converting to text first. These

Table 3: Main API components and their methods currently provided by HMMSim

API Component	**API Component Description**	**API Methods**	**Notes**
Engine	Discrete event simulation engine	addEvent	Add event
IEventHandler	Callback for Engine event	process	Process event
ITraceReader	Interface for trace readers	readEntry	Read next entry in trace
StatContainer	Statistics container	insert	Register statistic
		reset	Reset value of statistics
		print	Print all statistics
Counter	Counter that triggers interrupts when threshold is reached	setHandler	Sets interrupt handler
		add	Adds a value to the counter
		reset	Resets the counter
IInterruptHandler	Callback for counter interrupt	process	Called when interrupt happens
IMemory	Interface for components that receive memory requests (e.g., caches and memory)	access	Issue memory request
IMemoryCallback	Interface for components that issue memory requests (e.g., CPU and caches)	accessDone	Called when memory request completes
		unstall	Called when component is no longer stalling
IFlushable	Interface for caches that are flushable	flush	Issue cache flush request
IFlushCallback	Callback for flush requests	flushDone	Called when cache flush completes
IMemoryManager	Interface for memory managers (single, hybrid)	access	Translate virtual to physical address
		allocate	Allocate physical page to virtual address
IMigrationPolicy	Interface for migration policies	allocate	Decide where to allocate page
		migrate	Decide what page to migrate
		monitor	Called when memory is accessed

include counting number of accesses of each type, and counting the number of unique caches blocks or pages accessed.

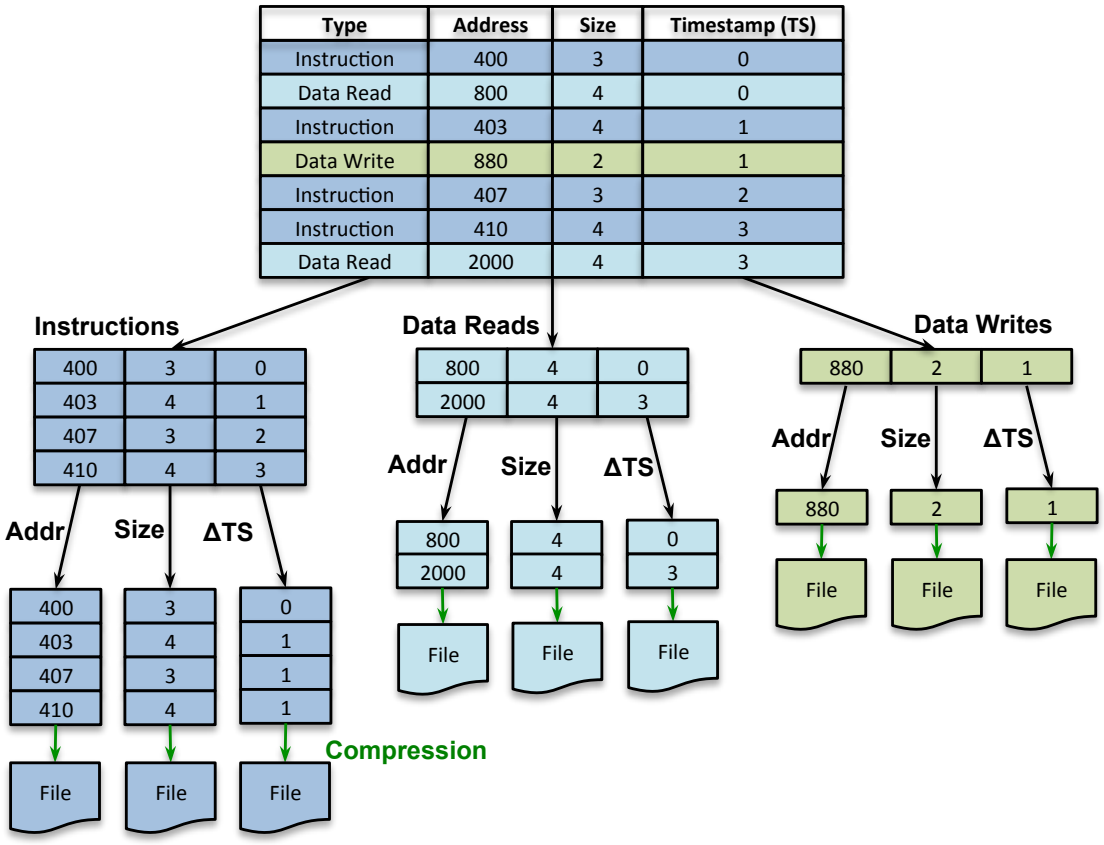

Figure 4: Splitting of traces for compression.

3.6 PERFORMANCE

This sections evaluates the performance and resource requirements of HMMSim. For all experiments, I ran HMMSim on a lightly loaded machine with a 2.8 GHz Intel Xeon processor, 25MB of LLC and 128GB of main memory. I measure execution time as reported by the *time* utility (wall clock time) and memory resident size as reported by *top*. I use million of instructions per second (MIPS) as our figure of merit for simulation performance.

I simulate three memory configurations: DRAM-only, PCM-only and software-managed hybrid memory. Each configuration runs one SPEC CPU2006 benchmark at a time. Each benchmark is run for one billion instructions. I report the average over all benchmarks

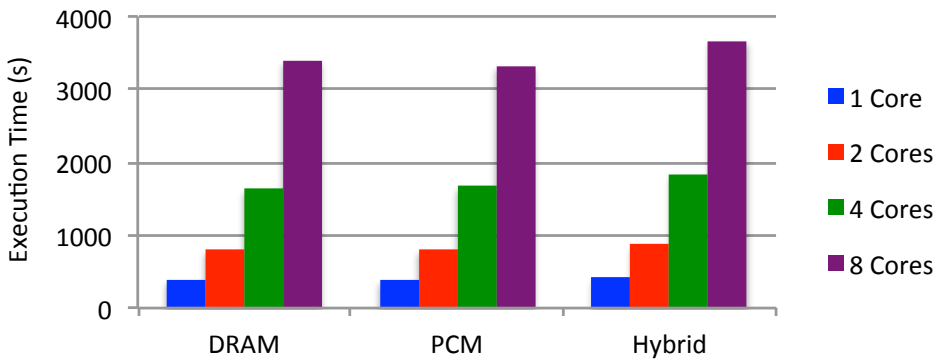

Figure 5: Execution time of HMMSim for different configurations and simulated cores counts.

because the variation among different workloads is small. I simulate systems with four core counts to show that execution time scales linearly with the number of simulated events.

3.6.1 Execution Time

Figure 5 shows the average execution time of HMMSim for 1, 2, 4 and 8 simulated cores. As expected, each doubling of the cores results in twice as many instructions being simulated. The execution time, however, increases by a little over 2 times. This is due to an increase in the LLC miss rate of the simulated system, which results in more events being simulated. However, the effect is small: on average there is less than 6% slowdown in MIPS for each doubling of the cores.

In general, the performance of HMMSim lies between 2.1 and 2.6 MIPS. This is approximately the same performance as a full system simulator, such as Simics, running a fast functional model without caches and memory. I believe this is an adequate speed for testing and experimenting with new ideas without having to spend considerable resources modifying complex tools.

3.6.2 Memory Usage

Figure 6 shows the resident memory size of HMMSim for different simulated core counts. For a DRAM-only system, memory usage varies from 38MB to 57MB as core count increases; for PCM-only, memory usage is between 262MB and 282MB, and for hybrid it varies from 262MB to 443MB.

Memory usage does not change after initialization and remains stable during the entire execution. The amount of memory used is linearly correlated with core count. This increase is due to some objects within the simulator being replicated when more cores are simulated, such as private caches, CPUs and traces readers. The difference in resident memory size between the three memory configurations is due to the particular organization of DRAM and PCM that is being simulated. PCM is configured to have more banks of smaller size than DRAM, requiring more memory. The hybrid configuration has the highest resident size because it contains both types of memory and needs other data structures to manage page migration.

The memory usage of HMMSim is relatively low and scales well with core count. The memory requirements are well within the capacity of current servers. HMMSim can simulate memory organizations with large number of components (PCM has 128 banks in this example).

3.6.3 Trace Compression

To show that the storage requirements of HMMSim can be handled by current infrastructure using typical storage capacity, I measured the size of SPEC CPU2006 memory traces stored in our high compression trace format. Each trace contains 1 billion instructions and a variable number of data accesses which depend on the characteristics of the workload. Typically, for each instruction there are between 0.2 and 0.7 data accesses. Traces are collected after a 5 billion instructions of the benchmark's execution have completed to avoid tracing the warm-up phase.

Trace sizes vary from 385MB to 1.45GB, with an average of 838MB. The difference in sizes is due to variation in data accesses per instruction and compression ratio of individual

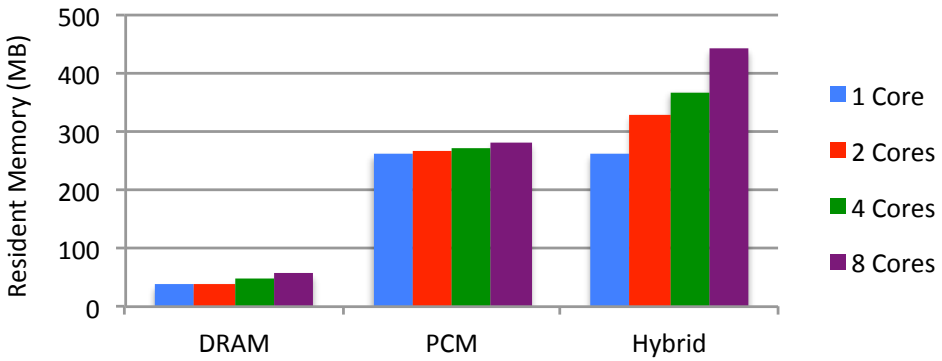

Figure 6: Resident memory of HMMSim for different configurations and simulated cores counts.

sub-traces. In the worst case, storing traces containing 200 billion instructions, which is more that enough to evaluate the performance of current benchmark suites, would take at most 300GB. The overall compression ratios of the traces are between 21 and 74, with an average of 41. Storing 200 billion instructions without compression would take at least 5TB of storage.

3.7 SUMMARY

This chapter presented HMMSim, a trace-driven simulator for software-hardware co-design of hybrid main memory. HMMSim models the entire hybrid memory system, including the processor, caches, DRAM and PCM, and migration policies that run in the OS. HMMSim provides an easy-to-use API for extending its functionality. HMMSim uses a novel trace compression scheme that significantly reduces the amount of storage required for traces.

4.0 CHARACTERIZATION OF OVERHEAD

As shown in Figure 1 in Chapter 1, there is great potential for achieving high performance in software-managed hybrid main memory by reducing interference and using good migration policies. However, before setting out to design hardware for reducing interference or new migration policies, it's imperative to first analyze and understand why current systems experience high overhead due to page migration [17].

4.1 OVERVIEW

Current commodity memory systems do not provide hardware support for page migration. In these systems, page migration is performed in software by the OS, which can have a detrimental impact on performance due the long duration of migrations. This long duration is caused by the slow write performance of PCM, which results in application pauses during page migration. Section 4.2 analyzes the potential impact of reducing the duration of migrations.

In addition to the impact of pauses due to long migrations, interference in the memory system due to data migration between DRAM and PCM affects the performance of software-managed hybrid memory. In Sections 4.3 to 4.5, I present analysis and simulation techniques to investigate the nature of this performance gap and understand how to make software-manage hybrid memory perform better. I characterize the overhead of page migration and study the delays that applications experience in the memory hierarchy. I identify the factors that cause the highest overhead.

4.2 PAGE MIGRATION LATENCY

To quantify the cost of pausing during page migration, I conducted experiments to measure the potential performance improvement that eliminating migration latency can bring. The cost of migration clearly depends on many parameters, including size of the DRAM and page migration policy; these results are illustrative of the problem. I measure the execution time of several SPEC CPU2006 benchmarks, as a proxy for general-purpose applications in smartphones and tablets. I compare total execution time when the latency of copying a page is *zero* and compare it to the total execution time when copy latency realistically accounts for memory subsystem parameters, which I briefly describe next.

To mask as much write latency as practical, I provision the memory system with significant parallelism. I use an 8-bank memory with 4 KB pages and 64-byte cache blocks. PCM reads take 125 cycles and PCM writes take 1K cycles, which is equivalent to Qureshi et al. [26] for a 64-byte cache block. With this organization, a DRAM to PCM migration takes 8K cycles with a page striped across 8 banks (there are 8 writes per memory bank for each migration). I use the Multi-Queue page migration selection policy [9].

Figure 7 shows normalized execution time (application execution time of the zero cost case divided by execution time of the realistic case). The figure shows that normalized execution time varies from 0.68 (*bzip2-4*) to 0.93 (*gobmk-2*) with a weighted average of 0.75 (i.e., 25% reduction in execution time). The benchmarks with smaller normalized execution times are the ones that are most harmed by migration cost. This latency is largely due to pausing during page copying (DMA transfer) from DRAM to PCM, which is expensive due to long PCM write time. Further, when migrations happen frequently, the writes to PCM may overwhelm the available buffers in the memory system, causing a bottleneck that stalls an application. Many benchmarks suffer from this latency, such as *bzip2-6* (0.68 normalized execution time), *leslie3d* (0.72) and *lbm* (0.74). In general, these results demonstrate that migration latency, as seen by the application, in a hybrid memory may be significant, and there is a large opportunity to recoup the performance loss, if in practice, migration can be done inexpensively.

In my view, any software-managed hybrid memory system that aims to perform suffi-

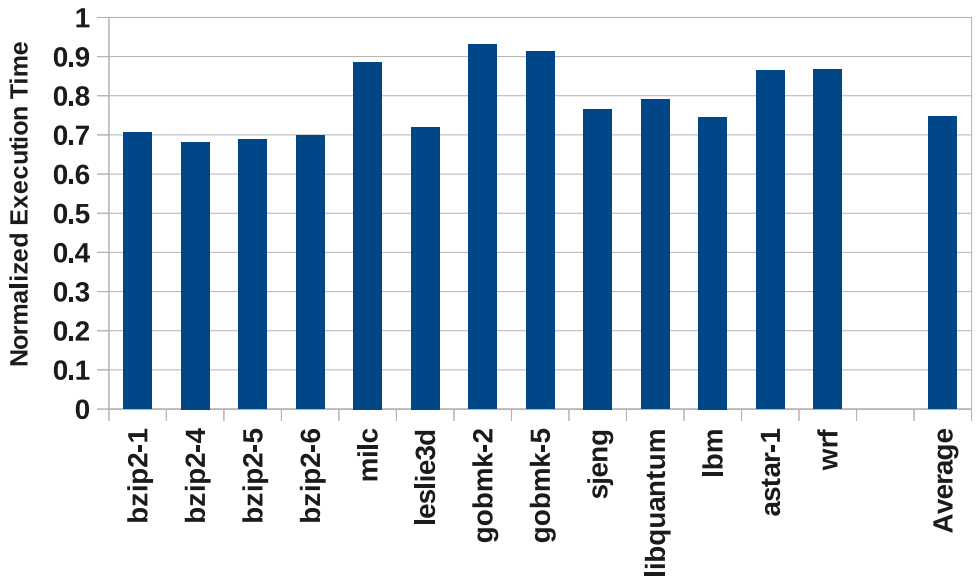

Figure 7: Normalized execution time of selected benchmarks without and with (baseline) migration cost.

ciently well must include a mechanisms for hiding pauses caused by long migrations. Therefore, for the remainder of the analysis presented in this chapter, I assume a baseline system that can perform migration without pausing application execution. A mechanism for doing this is presented in Chapter 5.

4.3 OTHER LIMITING FACTORS

As shown in Figure 1, migration can have a negative impact on performance. This happens because page migration is performed *concurrently* with regular application requests. The latency experienced by regular accesses changes due to interactions between application and migration memory traffic. For example, memory resource contention caused by migrating pages can harm (increase) the access latency of regular requests, increasing application

execution time. Through analysis, I identify performance-limiting factors in hybrid memory.

Migration Policy. In software-managed hybrid memory, the OS decides which pages to put in DRAM or PCM. However, the OS has coarse-grain information about application memory accesses. Further, due to computational limits and time constraints, the OS may be unable to fully analyze available data. Thus, the OS does not always make the best decisions for page migration. The difference between a realistic policy and an "ideal" one is migration policy overhead.

Cache Flush. A migrated page needs to be flushed from the L1 and L2 caches due to the changed physical address. The flush from L1, and especially L2, can result in an increased cache miss rate for application requests. In general, flushing pages from L2 has a much higher impact on average access time because cache blocks from a page are much more likely to stay longer in L2 than in L1.

Bank Contention. When application requests arrive at either the DRAM or PCM memory controller, they are forwarded to the appropriate bank based on their physical address. If the bank is busy, the requests are queued. The time a request spends in the queue depends on many factors, including queue occupancy, whether the request has higher priority than other requests, and whether the request is for an open row. When migrating a page that is mapped to a bank, a large number of requests will be issued to that bank. Application requests to the same bank wait longer in the queue, increasing overhead. Even if application requests have higher priority than migration requests, application requests must wait for ongoing migration requests to finish.

Bus Contention. Even when a bank is ready to serve requests, an incoming request may still be queued. This happens because the shared bus may be busy with requests from other banks. Migration requests, therefore, delay regular applications requests.

Row Buffer Interference. Most applications access memory in regular patterns, which hardware designers exploit to increase performance. For example, a physical page may be mapped to the same row of a bank so that the memory system can exploit the locality of reference of the application by keeping the bank row open. When migration is enabled, migration requests to or from banks with high application access locality interfere with open row buffers, increasing the row buffer miss rate seen by application requests. This in turn

causes extra delays to serve application requests because the row must be unnecessarily reopened multiple times.

4.4 ANALYSIS

To design hardware mechanisms and migration policies for hybrid memory that perform close to an ideal system, I must first determine how much each of the factors presented in Section 4.3 contributes to the overhead of page migration. In general, attributing performance slowdown to a particular factor is difficult because of the way they interact. For example, flushing the cache affects the cache miss rate (which directly impacts access latency), but it also affects bus and bank contention through increased memory traffic (which indirectly impacts access latency). To perform this detailed study of page migration overhead, I propose techniques to analyze memory access latency and determine the importance of the limiting factors.

The techniques use HMMSim, the simulator presented in Chapter 3, to collect detailed statistics about memory requests as they travel through the memory hierarchy. I present two simulation techniques to approximate the behavior of an ideal system: a *zero-interference migration* hardware configuration and an *offline migration* policy. I also present two new techniques for analyzing the memory behavior of applications. The first technique does *memory latency attribution analysis* that determines the average number of cycles memory requests spend in each component of the memory hierarchy. By comparing the memory latency attribution of different hardware configurations, memory performance bottlenecks can be identified. The second technique does *factor isolation analysis* of applications executed with different hardware configurations. It uses an analytic model and the statistics collected during simulation.

4.4.1 Zero-Interference Migration

To analyze the limiting factors of page migration, I consider a hypothetical system, *zero-interference migration*, that is not affected by the memory traffic and other costs associated with migration. This technique enables comparing the performance of current systems to an ideal system that does page migration without interfering with regular memory traffic. The ideal system still abides by the migration policy. In this hypothetical system, migration still has the same latency as the realistic full-interference case: migration is not instantaneous. However, accesses to DRAM and PCM from migration do not interfere with regular application requests. In addition, other side-effects, such as flushing the caches, are modified for zero-interference.

Zero-interference migration requires changes to two parts of the system. The first part is the hybrid-memory controller. In a full-interference system, migrations are done by reading the cache blocks of a page from the source memory and writing them to the destination memory. If regular application requests arrive at the memory controller while a page is migrating, the requests may be delayed. In zero-interference migration, memory accesses due to page migration are not modeled in the memories. Instead, each migration takes a fixed amount of time. To estimate page migration latency for the zero-interference model, I assume the memory is idle and calculate the time it takes to read all blocks from the source memory (including bank access and bus transfer) and write them into the destination memory queues. If the queues have enough capacity to store all requests, then the write time to the destination memory is not exposed in the migration time.

The second modification is to the cache. Because the state of the cache is changed by migration (flushing), future accesses to migrated pages might not hit in the cache, hurting performance. To avoid exposing this cost in zero-interference migration, I remap addresses in the cache. Instead of flushing a page, the mechanism inserts an entry into a remap table that tells the cache to look into the old cache location whenever an access for the new address arrives. The remapping entry is removed when all cache blocks from the page are eventually evicted. This scheme allows keeping the state of the cache in terms of which cache accesses hit or miss as if there had been no page flushes, while allowing migration to change physical

Algorithm 1 Offline migration policy

```
OFFLINEMIGRATE(current, intervals, threshold, size, counts[ ][ ], pageType[ ])
 1  // Get sums of access for each for the following intervals
 2  sums[1..size] ← 0
 3  for p ← 1 to size
 4        for i ← current to current + interval
 5              sums[p] ← sums[p] + counts[i][p]
 6  // Find least accessed DRAM page
 7  minSum ← ∞
 8  for p ← 1 to size
 9        if pageType[p] == DRAM and sums[p] < minSum
10              minSum ← sums[p]
11              min ← p
12  // Find most accessed NVM page
13  maxSum ← 0
14  for  p ← 1 to size
15        if pageType[p] == PCM and sums[p] > maxSum
16              maxSum ← sums[p]
17              max ← p
18  if maxSum − minSum > threshold × intervals
19        swapPages(min, max)
```

addresses.

4.4.2 Offline Migration Policy

To understand the effects of migration and isolate its impact on performance, I use an offline migration policy (an oracle) that uses future memory access counts to estimate the relative importance of pages. It migrates only pages that will benefit performance in the future. Although this policy is not optimal, it is a good approximation of an optimal one.

The offline migration policy works as follows. Time is divided into intervals, and access counts for each page during each interval are kept. These access counts are generated by analyzing the trace. The access counts can be gathered prior to the caches or after the L2 cache. The data available to the policy is the following: number of reads, number of writes, number of unique blocks that were read, number of unique blocks that were written, and number of unique blocks that were accessed (read or written).

Table 4: Memory components where application reads accumulate time

Name	Name
Translation Pause	DRAM/PCM Close
CPU Stall	DRAM/PCM Open
L1/L2 Access	DRAM/PCM Access
L1/L2 Stall	DRAM/PCM Bus Queue
DRAM/PCM Bank Queue	DRAM/PCM Bus

Algorithm 1 shows how the offline migration policy makes migration decisions. The policy first examines the next few intervals (the number of intervals is an algorithm parameter), and counts the number of accesses to each page (lines 2 to 5). The policy then compares the future access counts of the least accessed page in DRAM (lines 7 to 11) with the most accessed page in PCM (lines 13 to 17). If the number of accesses to the *slow page* is greater than the *fast page* by a threshold (an algorithm parameter), the pages are swapped (lines 18 to 19). Otherwise, no pages are migrated for the remainder of the interval. Once a page has been migrated, the next most accessed slow page and the next least accessed fast page are considered.

4.4.3 Memory Latency Attribution Analysis

Memory Latency Attribution Analysis (MLAA) determines the average number of cycles that read requests from the CPU spend in different memory structures. Requests that miss in both the L1 and L2 caches spend a number of cycles accessing the tag arrays (equal to the access latency of the caches), plus other cycles in the memory queues, possibly opening the row buffer, reading the data from the row buffer and transferring it on the bus. Requests that hit in a cache do not spend any cycles in lower levels of the hierarchy. MLAA captures the relative importance of each component of the memory hierarchy in the total latency experienced by requests from the CPU. This enables comparing different configurations to determine how the access latency changes with design parameters.

Table 4 lists all the memory components where memory requests accumulate time. *Trans-*

lation pause is the time that the CPU must be paused because the physical address of the requested virtual address is being changed at the memory manager as a result of migration. L1/L2 access is the time spent accessing the tag array in the cache level. *CPU, L1 and L2 stall* are due to a full queue in the next lower level. *Bank and bus queue time* are due to the corresponding bank or the bus being busy, respectively. *Close time* is the latency a request waits for a closed bank to become available, and *open time* is the time it takes to open a requested row. *Access time* is how long it takes to transfer the data at the row buffer to the bus, and *bus time* is the bus transfer time.

4.4.4 Factor Isolation Analysis

To estimate the effect that independent components of the memory hierarchy have on performance, I introduce Factor Isolation Analysis (FIA) that estimates the potential performance improvement of removing the overhead of a specific limiting factor. FIA uses an analytic model that calculates the average L2 memory access latency based on parameters and other data collected during simulation.

Table 5 describes the variables used in the model, and whether they are calculated by the model, parameters of the simulated architecture or values measured from simulation. The following formulas show how the average L2 access latency is calculated (X stands for either DRAM or PCM):

$$AR_{PCM} = 1 - AR_{DRAM}$$

$$AL_{L2} = L_{L2} + MR_{L2} \times AL_{Mem}$$

$$AL_{Mem} = AR_{DRAM} \times AL_{DRAM} + AR_{PCM} \times AL_{PCM}$$

$$AL_{X} = AL_{XBank} + AL_{XBus}$$

$$\begin{aligned} AL_{XBank} &= L_{XAccess} + AL_{XBankQ} \\ &+ MR_{XOpen} \times (L_{XClose} + L_{XOpen}) \\ &+ MR_{XClose} \times L_{XOpen} \end{aligned}$$

Table 5: Variables used in the analytic model. X stands for either DRAM or PCM

Variable	Description	Type
AR_{PCM}	PCM access rate	Calculated
AL_{L2}	Average L2 access latency	Calculated
AL_{Mem}	Average memory access latency	Calculated
AL_{X}	Avg. X access latency	Calculated
AL_{XBank}	Avg. X bank access latency	Calculated
AL_{XBus}	Avg. X bus access latency	Calculated
L_{L2Tag}	L2 tag access latency	Parameter
L_{XClose}	X bank close latency	Parameter
L_{XOpen}	X bank open latency	Parameter
$L_{XAccess}$	X bank access latency	Parameter
L_{XBus}	X bus transfers latency	Parameter
MR_{L2}	L2 miss rate	Measured
AR_{DRAM}	DRAM access rate	Measured
MR_{XOpen}	X row buffer open miss rate	Measured
MR_{XClose}	X row buffer close miss rate	Measured
AL_{XBankQ}	Avg. X bank queue latency	Measured
AL_{XBusQ}	Avg. X bus queue latency	Measured

$$AL_{XBus} = L_{XBus} + AL_{XBusQ}$$

The formula contains two types of row buffer miss rate. The close miss rate is due to accesses to the bank that miss in the row buffer because the row buffer is closed. These only pay the row open penalty. The open miss rate is due to accesses that miss in the row because the row buffer is open in a different row. These pay both the row close and row open penalties.

FIA determines the factors that cause the highest overhead due to page migration. Using this analysis, I can determine how much speedup can be obtained if the effects of a selected factor could be ignored. For example, if there is high contention at the PCM bus due to page migration, I can estimate the speedup of a system that does not have contention at the bus. This is a powerful technique because it allows focusing on the system components that cause the largest performance drop (bottlenecks).

FIA isolates the effects of individual characteristics of a configuration (such as bank queue time) and estimates the performance of another configuration as if it had the single

Table 6: List of variables and associated overhead

Variable	Overhead
MR_{L2}	L2 Cache flushing
MR_{XOpen}, MR_{XClose}	Row buffer interference
AL_{XBankQ}	Bank contention
AL_{XBusQ}	Bus contention

characteristic of the original configuration. In particular, I are comparing full-interference (including the impact of migration) with zero-interference migration.

FIA uses the analytic model to calculate two values for average L2 access latency with different parameter sets. In the first one, all measured variables (see Table 5) come from full-interference simulation. In the second set, all measured variables come from full-interference simulation, except one, which comes from zero-interference simulation. With this technique, the speedup from eliminating an individual factor's overhead can be determined. Table 6 lists the variables I consider for analysis with a description of the overhead that they capture. For row buffer interference, both the open and close miss rates are changed in the formula at the same time, yielding only one affected factor.

4.5 PAGE MIGRATION OVERHEAD

In this section, I describe results from my analysis to understand the nature of overhead in software-managed hybrid memory systems using the techniques described above, and present my main findings. The focus is single-programmed workloads running on low-end systems, such as those found in mobile devices.

4.5.1 Methodology

I use a subset of the SPEC CPU2006 benchmark suite[1] for evaluation. I treat each input of a benchmark with multiple data sets as a separate workload, which yields a total of 52 different benchmark/input combinations. In the figures, a number after a benchmark name is the input for the benchmark (e.g., bzip2-2 is the 2^{nd} reference input for bzip2). I present results for 35 of the 52 combinations. The working sets of the remaining workloads fit in on-chip caches, causing negligible migration overhead. I simulate each benchmark for 1 billion instructions.

Table 7 shows the main architectural parameters. DRAM and PCM access times are from Qureshi *et al.* [26] and energy values from Lee *et al.* [22]. PCM latencies are adjusted to account for a small 64-byte cache block. For PCM, t_{RP} is 0 for clean row buffers (due to non-volatility) and 150ns for each dirty block in the buffer (to account for power constraints in the PCM chip). Since the working set varies considerably across benchmarks, I limit the available DRAM space to 25% of the total footprint of each workload, similar to past work on migration [16]. This choice allows evaluating performance of the benchmarks independent of the DRAM size. Hence, benchmarks that differ noticeably in working set size are comparable. By limiting DRAM space in the memory manager instead of through DRAM parameters such as bank size and count, I avoid configurations that differ too much from realistic systems (for example, too few or small banks).

I use two migration policies: Multi-Queue (MQ) and Offline. **MQ** [9] is the state-of-the-art approach for hybrid memory, and it has been shown to work well in practice. I use the same parameters as the original MQ study. I experimented with other parameters for MQ but found that the values in the original study work best for all benchmarks. In particular, I found that other values either do not improve performance (too few migrations) or hurt performance due to high interference. **Offline** is the oracle policy from Section 4.4.2. I use this to determine how much potential exists beyond MQ: a gap between MQ and Offline reflects lost performance opportunity that a better future policy might achieve. For Offline, I chose the parameters of the algorithm empirically. The interval length is 100,000

[1] I excluded *dealII*, *wrf* and *xalancbmk* due to limitations in my simulation environment

Table 7: Architectural parameters

Parameter	Value
4GHz chip multiprocessor	1 4-issue wide, out-of-order core, 128-entry reorder buffer
L1 I/D private cache	64KB per core, 4-way, LRU, 3 cycle hit, 16-entry queue
L2 unified shared cache	2MB, 16-way, LRU 32 cycle hit, 32-entry queue
4GB DRAM @ 1000MHz	64 banks, 32-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-12-12 (ns)
4GB PCM @ 400MHz	64 banks, 8-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-55-150 (ns)
PCM/DRAM bus	64-bit single-channel

instructions, and the number of intervals to look into the future is 50. I rank pages by the number of unique blocks accessed, and use a threshold of 2 accesses per interval.

I consider two interference regimes: full-interference and zero-interference. **Full-Interference** is a realistic memory configured as described above. It accounts for latencies, contention and bottlenecks. **Zero-Interference** is the scheme from Section 4.4.1. By comparing both regimes, I can find bottlenecks to guide future memory subsystem development.

In addition to MLAA, I report the result of two studies for identifying limiting factors in software-managed hybrid memory. The first study uses FIA to understand the hardware overhead of individual components of the memory hierarchy. For this study, I fix the policy to Offline. The second study compares MQ and Offline using Zero-Interference migration to isolate the effectiveness of the migration policy.

4.5.2 Memory Latency Attribution

The MLAA graphs for the 35 workloads (plus average) are shown in Figure 8. Each workload has two bars. The first bar corresponds to Full-Interference, while the second one to Zero-Interference. Each bar shows normalized number of cycles per read request spent at different components of the memory. The bars are normalized to the total number of cycles per request

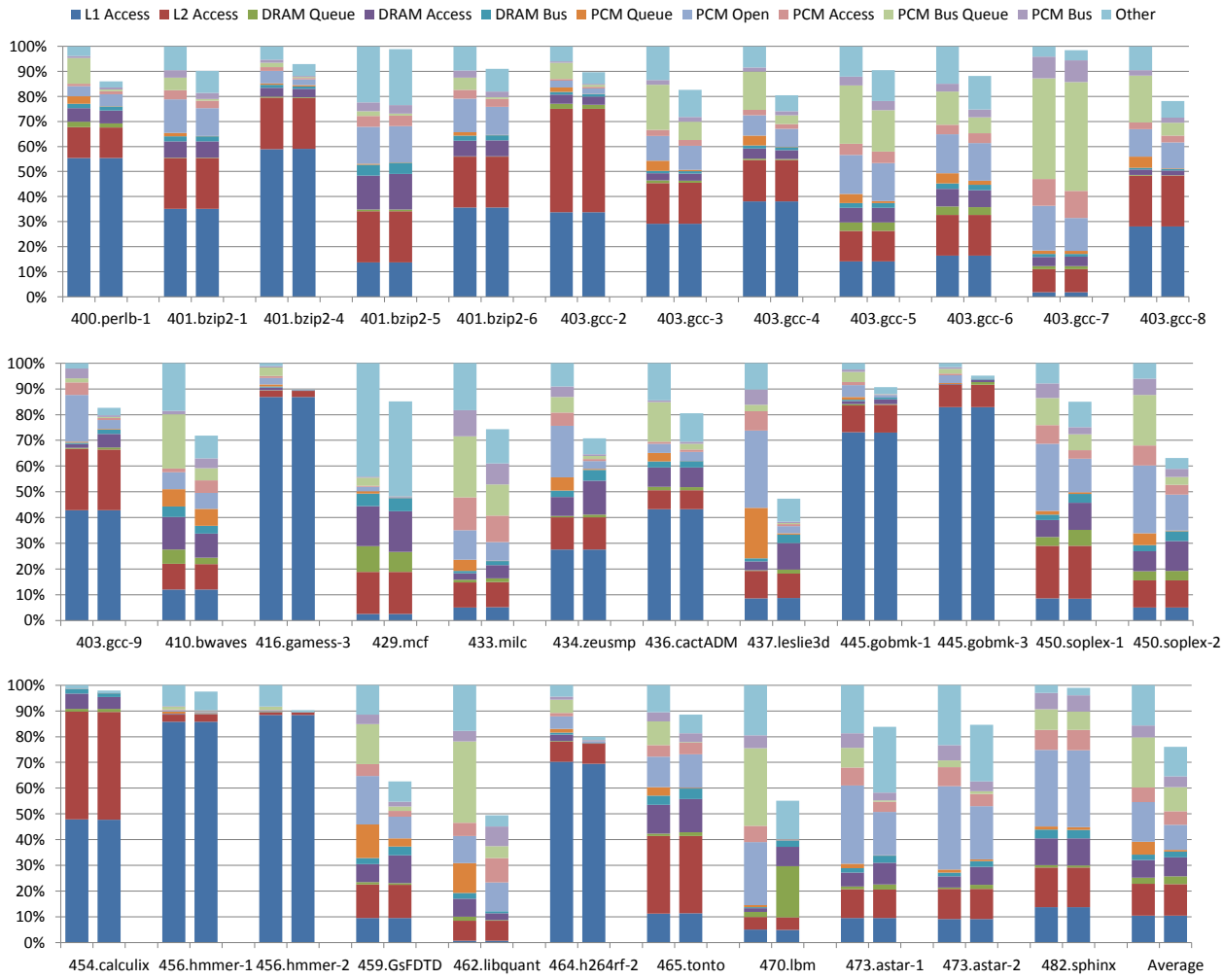

Figure 8: MLAA for Offline. For each workload, first bar is Full-Interference and second bar is Zero-Interference.

of Full-Interference. To avoid a cluttered graph, I aggregated into the "Other' category the cycle counts of 8 components that have an average normalized number of cycles of less than 2%. They include: translation pause, CPU, L1 and L2 stall, DRAM and PCM close, DRAM open and DRAM bus queue.

On average (bottom right bars), the number of cycles can be reduced by 24% when going from Full-Interference to Zero-Interference. This reduction comes mostly from three

sources: PCM bus queue (10%), PCM open (6%) and PCM bank queue (4%). In Full-Interference, the PCM bus queue becomes saturated with migration requests, which delays regular requests. The same holds (to a lesser degree) for PCM bank queue because there are several banks to distribute request but only one bus. The reduction in PCM open time is due to a lower row buffer miss rate. When migration requests are serviced by banks, open row buffers that may service future application requests must be closed. This increases the time spent opening row buffers. These results suggest that, on average, most of the overhead of page migration is caused by interference of migration requests with application request rather than migration latency itself.

Other components of the memory do not have, individually, a significant reduction on cycle count even though they contribute considerably to the total cycle count. In particular, the L1 access time is the same for both interference regimes because migration does not change how often the L1 tag is accessed. Flushing the L1 after migration does change the L2 access count, but to a very small degree due to the small size of the L1. The DRAM components of memory also exhibit little change in cycle count. This is because DRAM is faster and therefore has more idle time, allowing migration requests to proceed with less interference.

The behavior of individual benchmarks varies widely. Some benchmarks benefit little (cycle count reduction of less than 2%) from Zero-Interference migration. *454.calculix* and *456.hmmer-1* are dominated by L1 and L2 access cycles, so changes in other components of the system have no effect. Cycle counts for *401.bzip2-5* are more evenly distributed across the hierarchy, but do not change. In *403.gcc-7*, cycles counts for PCM open decrease by the same amount that PCM bus queue cycles increase, resulting in similar performance. In this case, a decrease in row buffer miss rates (less cycles opening rows) causes a reduction in service time that is shifted to the queue.

Other benchmarks benefit considerably, often more than 40%. Again, there are different reasons: *437.leslie3d* (53% reduction) benefits from PCM bank queue and PCM open; *450.soplex-2* (37%) and *470.lbm* (45%) benefit from PCM open and PCM bus queue; *459.GemsFDTD* benefits from PCM bank queue, PCM open and PCM bus queue; and *462.libquantum* (51%) benefits from PCM bus.

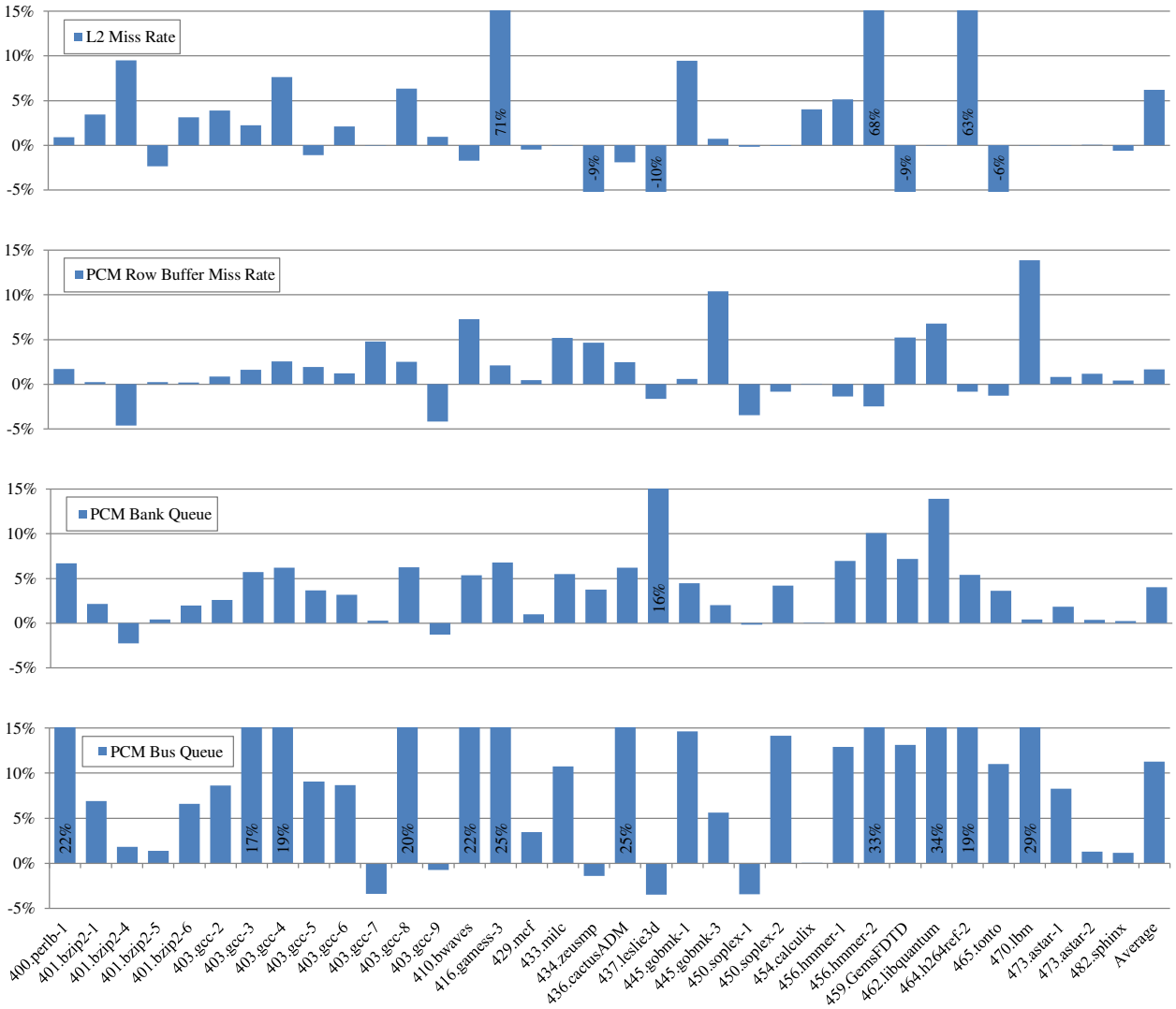

Figure 9: Potential L2 access latency reduction that can be obtained by eliminating 4 different factors that cause overhead.

Although MLAA identifies where requests spend most of their time and how much each component contributes to the total latency, they do not determine the cause of the latency. For example, a low cycle count for PCM bank queue time in Zero-Interference means the banks are less busy because migration requests do not use PCM resources. However, a low

count for PCM open time can be caused by increased L2 miss rate, DRAM access rate or PCM row buffer miss rate. Using only latency attribution, I cannot determine the underlying cause of the delay. To solve this problem, I use FIA to isolate the effects of individual factors that increase application memory access latency.

4.5.3 Factor Isolation Analysis

The results of FIA are shown in Figure 9. Each graph shows the reduction in L2 access latency obtained by eliminating the overhead caused by one particular factor (see Table 6). I omit graphs for the DRAM factors, as their reductions are less than 1% on average.

On average, eliminating the overhead of the PCM bus queue has a potential reduction of 11%; the highest among all factors. Next are L2 miss rate (6% average reduction), PCM bank queue (4%) and PCM row buffer miss rate (2%). Again, PCM bus queue and PCM bank queue have a high reduction potential because they are busy under Full-Interference. The reduction from the L2 miss rate factor is due to flushing the cache under Full-Interference. Likewise, the PCM row buffer miss rate is reduced because migration request are not competing with regular requests for open row buffers.

The relative importance of factors varies across workloads. For example, in *416.gamess-3*, L2 miss rate (71%) is more important than PCM bus queue (25%) and PCM bank queue (7%). This benchmark has a very high L2 hit rate, and flushing pages from the L2 causes too many additional misses.

A negative reduction means that the measured variable (see Table 5) has a larger value with Zero-Interference than with Full-Interference. Although this is counterintuitive, it happens because the flow of requests to components of the hierarchy varies under Zero-Interference, and some structures might get more requests than with Full-Interference. For example, the L2 miss rate is generally reduced because no flushing takes place under Zero-Interference. However, sometimes flushing evicts dead data that remains a long time in the cache, which frees space for more useful data.

The results of FIA match those of the MLAA. DRAM factors are found to be unimportant from both analyses. On average, the factor with the highest reduction (PCM bus queue)

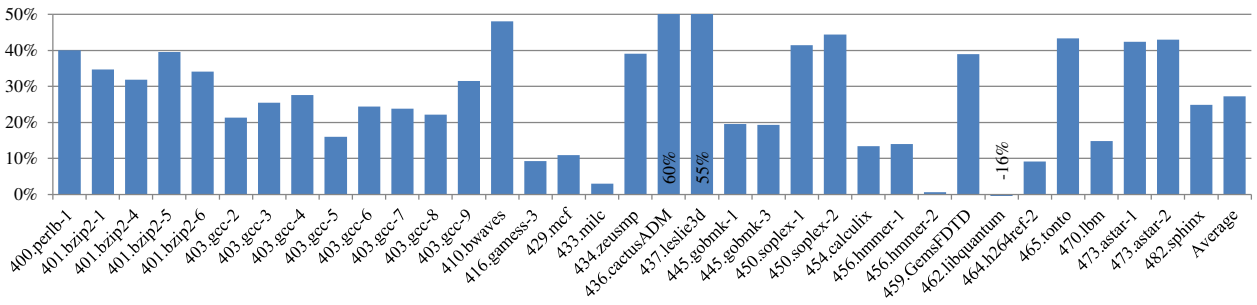

Figure 10: L2 access latency reduction from using the Offline migration policy relative to Multi-Queue.

matches the component with the highest decrease on cycle count. The next highest factor (L2 miss rate) does not have a corresponding component because L2 misses cause additional cycles in several components of the hierarchy.

The results of MLAA and FIA also match for all individual workloads. For example, in *416.gamess-3* the high reduction of L2 miss rate causes a cycle count decrease in all components except the L1 and L2 tags. Another example is *403.gcc-7*, with a reduction of 5% due to PCM row buffer (lower cycle count due to PCM open in the MLAA graphs), and an *increase* of 3% due to PCM bus queue (*higher* cycle count due to PCM bus queue).

Although MLAA and FIA identify bottlenecks in the memory hierarchy, they do not provide information about the effectiveness of the migration policy. To study this, I study migration policy overhead by comparing Offline and MQ.

4.5.4 Migration Policy Overhead

The results of the study of migration policy overhead are shown in Figure 10. The graph shows the reduction in L2 access latency that Offline has relative to MQ under Zero-Interference. The reduction is quite dramatic (27% on average), suggesting a large potential. The large reduction is due to the ability of Offline to move data to DRAM before it is used.

In addition, Offline is able to compare the access counts of pages to determine the actual benefit (number of accesses) of swapping two pages. In workloads that have low reductions, such as the streaming benchmark *462-libquantum*, Offline is unable to benefit from low memory access latency because the migrated pages are only accessed a few times while in DRAM and the system cannot migrate data quick enough to continuously service requests from DRAM. This causes most accesses to go to PCM, which does not have enough bandwidth to serve requests quickly, hurting performance. This effect does not happen with MQ because fewer pages reach the access count threshold that triggers their migration.

4.6 DESIGN IMPLICATIONS

In this section, I discuss the main implications of my findings in the design of software-managed hybrid memory.

The results of Section 4.5.4 about migration policy overhead show that an ideal policy has a large potential for improving performance relative to state-of-the-art policies. Thus, researchers should focus on finding better policies, as there is ample room for improvement.

There are several aspects of migration policies that need to be included in future systems. First, the migration policy needs to be aware of the cost of migration in the underlying hardware, as well as the benefits of moving a page to faster memory compared to leaving it in PCM. My own offline policy does a limited cost-benefit analysis with oracle information. However, online policies, which do not have future access counts, must have a more complete cost and benefit model if they want to be as effective as offline policies. Second, a good migration policy must be effective even if the hardware does not provide complete information about memory access patterns. To achieve this, I must determine how to collect and filter access pattern information at the OS level efficiently, while still providing accurate information for migration policies.

Although my techniques provide a detailed analysis of the hardware overhead of page migration, they do not provide a similar level of detail for the migration policy overhead. For instance, I do not know if the large gap between ideal and state-of-the-art policies is

due to the ranking algorithm of the policy, the inability of the policy to estimate the cost of migration, or the reduced access pattern information or something else. An analysis of the factors that cause overhead, similar to my FIA but for migration policies, is essential for understanding migration and designing future policies.

In Section 4.5.3 I also showed that hardware overhead is considerable and varies widely in importance across workloads. Some of the causes of overhead can be easily fixed. For example, the factor with the highest L2 access latency reduction is the PCM bus queue. Its overhead can be eliminated with faster, wider or simply more buses. Eliminating the overhead of the L2 miss rate requires a more complex solution than simply adding hardware. The overhead of the PCM bank queue is also more difficult to tackle. Simply adding more banks or splitting banks might not solve the problem if interference still occurs. In addition, many banks might increase the cost of the system significantly. Lastly, the PCM row buffer miss rate overhead also requires more complex solutions.

4.7 SUMMARY

This chapter presented new analysis and simulation techniques to aid in the development of new policies and hardware mechanisms for low-cost migration. The chapter identifies factors that limit performance in hybrid main memory, and rank their relative importance. Using the new techniques, it is shown that the highest L2 access latency reduction potential comes from developing better migration policies (27% average), followed by eliminating the overhead of the PCM bus queue (11%), L2 miss rate (6%), PCM bank queue (4%) and PCM row buffer miss rate (2%).

5.0 CONCURRENT PAGE MIGRATION

As shown in Chapter 4, there is considerable potential for software-managed hybrid memory. Any hybrid memory system that uses PCM must be able to hide the long latency of PCM writes. In software-managed hybrid memory, this is especially important, because page migration can amplify the negative impact of long writes by forcing pauses during page migration, as shown in Section 4.2. This chapter proposes basic hardware support for performing page migration without pausing a program's execution.

5.1 OVERVIEW

5.1.1 Memory Management

Like any main memory, in a hybrid main memory system, the OS assigns physical memory pages to programs based on their needs, available memory capacity and system load. It must also decide the type of memory (PCM or DRAM) and how to allocate each type among applications. These decisions are made using memory access behavior, virtual page type, application type (e.g,. real time vs. general purpose) and/or quality of service (QoS) requirements. For instance, if a virtual page is frequently written, the OS assigns a DRAM page to it, as this assignment reduces the number of writes to PCM and the associated performance, energy and endurance impact [5, 9]. Similarly, the OS can decide to leave certain pages in PCM, such as code and read-only pages, because the pages are rarely or never written and allocating them to PCM will save DRAM space for more important pages (e.g., most frequently written). The OS can also decide to allocate all or some pages based

on time criticality (i.e., for a real-time task), or other QoS metrics, to increase predictability of memory access latency and/or application responsiveness.

Most applications exhibit varying memory behavior over distinct phases of execution, causing write frequency of individual virtual pages to change over time. Pages that were once frequently written (e.g., during initialization) can become rarely written in later execution phases, or vice versa. To prevent rarely written pages from taking up too much DRAM space, the OS may change an initial assignment of virtual to physical pages during execution and *demote* a page from DRAM to PCM. Conversely, *promotion* exploits the high speed and low energy of DRAM writes by moving pages from PCM to DRAM.

5.1.2 Page Migration

The promotion and demotion of pages between DRAM and PCM requires copying pages from one type of memory to the other. This procedure is termed *page migration*; it is controlled by the OS. The actual data transfer is done by a DMA engine (similar to other partitioned memory sub-systems) to offload copying in the background of application execution. As long as the application does not write to a page under migration, its execution continues normally, as reads to a migrating page can be serviced from the old copy of the page. However, if the application writes to the migrating page, it must be paused until the migration finishes before proceeding with the write. This pause is necessary for correctness, because the write has to be done *after* the old version of the data has been copied to its new memory location by the DMA transfer.

Due to long PCM write latency, pausing the application causes performance overhead, possibly rendering the use of migration impractical. The question is then, for a hybrid memory, how much does the migration actually hurt application performance relative to the gain of coalescing writes in DRAM? The answer to this question is related to how often pages are migrated, whether an application writes to a migrating page, and how long the application is paused after a write. The answer motivates this work and shows why hardware support for migration is needed. I first explain how migration normally works with DMA, and I then examine the performance impact.

Algorithm 2 Sequence of steps performed during conventional page migration

```
pauseMigrate(app, virtPage, dstPage)
 1  app.pause() // Pause the application that currently maps the migrating page
 2  pageTable.setReadOnly(virtPage) // Mark page as read-only (update PTE and TLB)
 3  tlb.setReadOnly(virtPage)
 4  cache.flush(virtPage) // Flush the page from the caches
 5  srcPage ← pageTable.getPhysPage(virtPage)
 6  dma.copy(srcPage, dstPage) // Program DMA to copy the frame to the new location
 7  app.resume()  // Resume application, but trap to OS to pause on writes
                  // to the migrating page
 8  while (dma.isCopying())
 9        NULL //  Wait until DMA completes
10  app.pause()
11  pageTable.setPhysPage(virtPage, dstPage)    // Update PTE and TLB entry, restoring
                                                // write permissions
12  tlb.setPage(virtPage, dstPage)
13  app.resume()
```

Algorithm 2 shows the process traditionally used in commodity memory systems to migrate pages between DRAM and PCM. To start a migration, the application is temporarily paused (line 1) to mark the virtual page, selected for migration, as read-only in the page table and TLB (lines 2 and 3). This action ensures that a write to the page will be intercepted. The page is flushed from the caches because the page physical address, used to access the cache, will be different after migration (line 4). The cache flush must be done prior to the start of migration. Otherwise, dirty blocks belonging to the migrating page could be written back to the old location during migration, potentially losing updated data. After flushing the cache, the DMA transfer is programmed (lines 5 and 6) and the application is resumed. The program may continue reading from the original physical page, as long as it does not write to it. If the page is written, an interrupt is triggered (through a page write access violation) and the application is paused until the migration is done. At the end of migration, the page table entry (PTE) of the migrated virtual page and the corresponding TLB entry are updated to reflect the new physical location of the page (lines 10 to 13). The application is also briefly paused during this period for atomicity to update the PTE and TLB.

In the algorithm, two pauses in the program's execution are used by the OS to atomically

update the PTE and TLB (lines 1 and 10). These pauses are similar to the ones that are normally done by the OS to update paging structures. The TLB update may also generate shoot-downs to other cores for shared pages. These pauses are *not* the overhead that I address. In fact, atomic updates to paging structures also occur with my approach. I address the longer pause induced by the application writing to a page that is being copied by the DMA engine.

5.2 CONCURRENT PAGE MIGRATION

Concurrent page migration (CPM) is a novel scheme designed to overcome the cost of migration latency in hybrid memories. The technique permits an application to *continue execution during a page migration when it writes to an actively migrated page*. The main challenge solved with CPM is to guarantee that neither reading nor writing to a migrating page will cause the application to pause, while at the same time ensuring that updates are not corrupted or lost.

5.2.1 Buffering Writes

The idea behind CPM is to continue servicing read requests to the migrating page that miss in the LLC from the old physical location. This ensures correctness because the page has not been modified since the migration started, and thus, both pages contain the latest version of the data. To maintain consistency, as usual, if the application writes to a page under migration the typical race condition applies: (a) if the writes are sent to the old page, the part of the page that is being written could already have been copied to its new location and updates are ignored; (b) if writes are sent to the new page, it is possible that the part of the page that is being written has not been copied to its new location and updates are overwritten.

In CMP, the writes are buffered (a copy of the data) and only flushed to a page's new location once the DMA transfer for the migration has finished. During migration, all reads

that follow a write to the same part of the page are serviced from the buffer instead of the original location. The key insight for efficiency in my scheme is that the buffer to hold the dirty data already exists: the last-level cache (LLC).

The scheme works as follows. When a page migration is started, all its blocks are marked as ineligible for eviction, a process I call *pinning*. On cache block replacement, a block that is not part of a pinned page is selected for eviction. Because application writes to memory only come from LLC write-backs, pinning a page guarantees that application writes to the pinned page are not done during the migration. Once migration finishes, the page is *unpinned* and the contents of its dirty blocks are flushed from the cache, redirecting the write-backs to the new page location.

As an alternative to CPM, adding deep buffers to the memory system might hide the long latency of DRAM to PCM migrations. However, the additional energy cost of write buffers make this solution unsuitable for mobile systems, especially since these are associative structures that have to be accessed on every memory request to ensure correctness. The proposed scheme also hides long migrations, but uses existing structures and incurs little overhead.

5.2.2 Page Migration

Algorithm 3 gives the steps for CPM taking into account the asymmetry of memory technology (mainly slow writes in PCM). If a page is migrated to PCM, it is flushed from the cache prior to the start of migration (lines 2 to 3). This step causes only one write to PCM because the dirty data in the LLC is flushed to DRAM before migrating it to PCM. If the cache is not flushed, the block will be written twice to PCM: once when it is copied from DRAM and again when flushed from the cache.

Although an application is not paused during migration, program execution must still be suspended briefly at the end of migration to guarantee that the cache flush and the update of the PTE and TLB occur atomically as seen by the executing program (lines 8 to 12). Allowing the program to execute instructions that access the migrated page between these two operations could lead to data corruption. For example, if the TLB is updated with the

Algorithm 3 Sequence of steps performed during concurrent page migration

```
concurrentMigrate(app, virtPage, dstPage)
 1  cache.pin(virtPage) // Pin the page to the cache
 2  if (dstPage.isPCM())
 3        cache.flush(virtPage) // Flush the page from the caches
 4  srcPage ← pageTable.getPhysPage(virtPage)
 5  dma.copy(srcPage, dstPage) // Program DMA to copy frame to new location
 6  while (dma.isCopying())
 7        NULL // Wait until DMA unit finishes copying the frame
 8  app.pause()
 9  pageTable.setPhysPage(virtPage, dstPage) // Update PTE
10  tlb.setPage(virtPage, dstPage) // Update TLB
11  cache.flushRedirect(virtPage, dstPage)        // Flush page from cache, redirect write-backs
                                                  // to the new location
12  app.resume()
13  cache.unpin(virtPage)
```

new location of the page but the cache has not been flushed yet, reading from the page could return a stale (old) data value (which is in the new memory location) instead of the most recent one (which is in the cache but with the old physical address). Reversing the order of these operations does not solve the problem, because flushing redirects write-backs to the new memory location, but the TLB still points to the old location.

Pinning and unpinning a page do not have to occur atomically with respect to the cache flush or TLB update. If the page is pinned prior to the start of the flush or copying of the page, the application can continue to execute (line 1). Similarly, the page must be unpinned after the flush and TLB update (line 13). Since the old physical page is free after migration and it is not present in cache (it was flushed), keeping it pinned a little longer does not affect correctness or performance, as long as the page is unpinned before it is returned to the OS for reuse. The (optional) cache flush of the page before migration does not need to occur atomically to program execution because this flush is not required for correctness but only to reduce the number of writes to PCM. If a cache block that has already been flushed is written (regardless of whether the rest of the cache blocks in that page were flushed), it would simply remain in the cache (the page is pinned) until it is flushed after migration.

5.3 HARDWARE SUPPORT

Implementing CPM requires small hardware modifications in three parts of the system. First, cores need a mechanism to temporarily pause program execution when a write is done to a given page, so that a cache flush and update of the page table and TLB entry occur atomically after a migration. The TLB in each core is also modified to support updates of individual entries initiated by hardware. Second, the LLC must be modified to support pinning pages while a migration is under way. Finally, the memory interface must be changed to accommodate a hardware entity that controls the process of migration (by instructing the cores and caches to perform certain tasks) and handles the exchange of information between the hardware and the OS (by accepting requests for migration). The DMA unit that does the actual copying is not modified. I start this section with an overview of the new architecture and then explain the required changes. In addition, I show how migration works and how the hardware structures are used.

5.3.1 Overview of Architecture and Changes

Figure 11 shows an overview of the hardware changes. The structure in the figure is based on current chip multiprocessors that feature separate cores and a banked LLC [48]. The LLC is composed of four physical slices; each core sees a single, large shared LLC, corresponding to all slices. Cache requests are forwarded to the appropriate cache slice, according to their physical address, transparent to the executing program. LLC misses are handled by the system agent (using Intel's terminology), which is a separate structure that accesses main memory and performs other tasks.

Each core is modified to include a *Pause on Access* (POA) register, which is used to briefly pause execution at the end of migration to guarantee that updates to the TLB and PTE and cache flushes occur atomically. The TLB logic is modified to support the update of individual entries upon reception of a message. Each LLC slice is modified to contain a *Pinning* register (PIN), which holds a valid bit and the address of the physical page that is currently pinned to the cache. The use of the PIN and POA registers is explained in

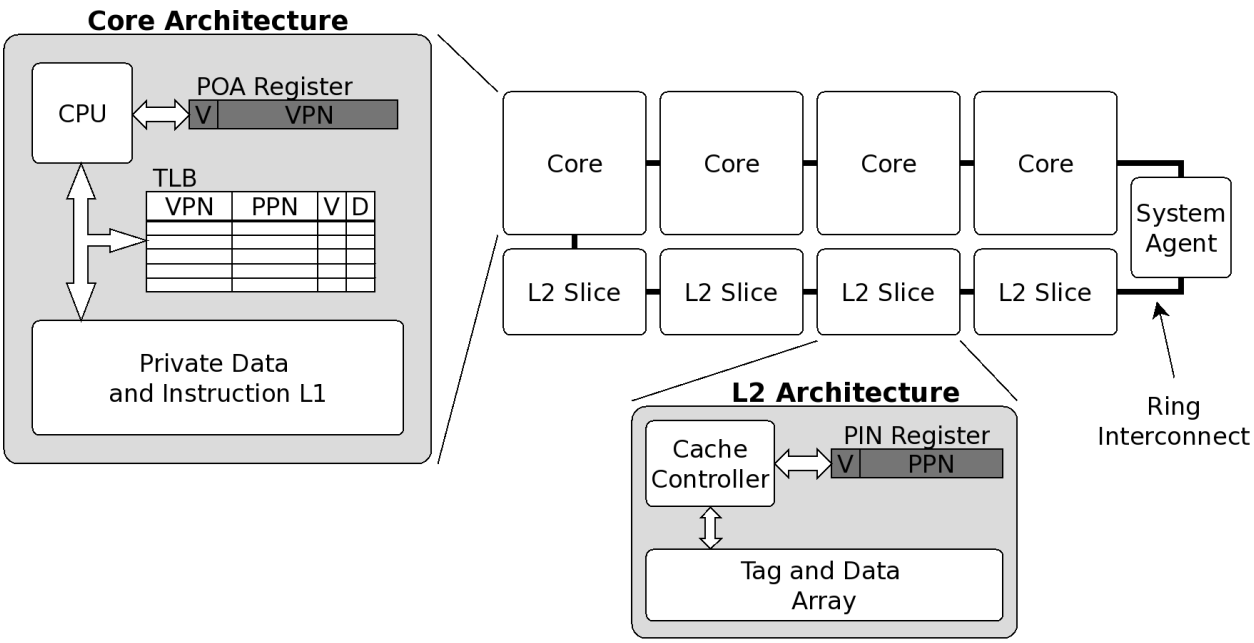

Figure 11: Overview of architecture changes for CPM. New components are shown in dark gray. Changes to the system agent are shown in Figure 12.

Sections 5.3.2 and 5.3.3, respectively.

Most of the hardware changes are made to the system agent. The modified system agent is shown in Figure 12, with old components shown in white (controllers, DMA unit and migration buffer) and the new ones in gray (migration manager with migration queue). The main new component is the *migration manager* (MM), which coordinates the sequence of actions done by different components during page migration. The MM contains a data structure called the *migration queue* (MQ) to hold pages to be migrated. This queue is updated by the OS with several migration requests, once the OS has decided what pages to promote to DRAM and demote to PCM. The queue is programmed in batch to avoid involving the OS in every page migration. Once updated, the hardware does the migrations in the queue until there are no more entries.

Figure 12 also shows some of the messages sent and received by the system agent as

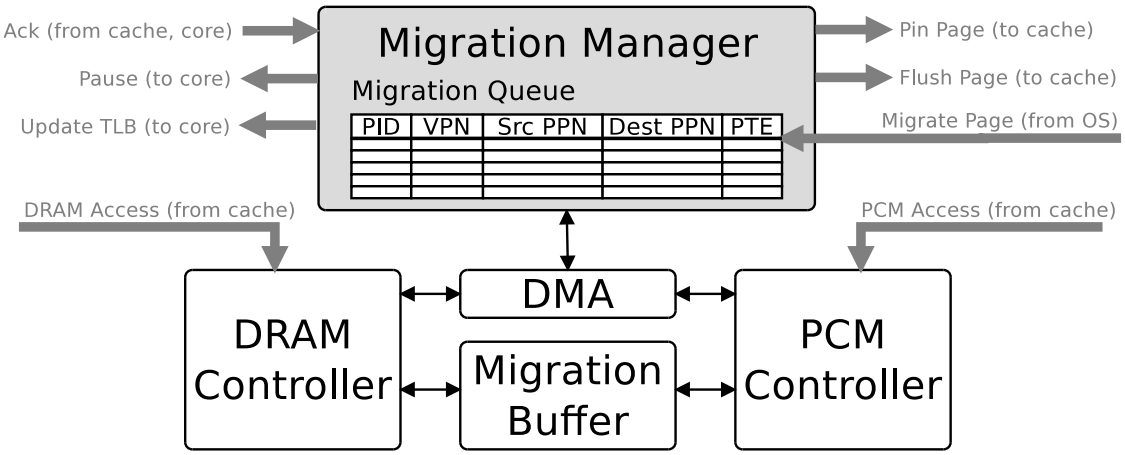

Figure 12: The modified system agent, showing new components in gray. Gray arrows represent messages from/to the cores, LLC cache slices and OS.

part of migration and conventional operation (read and write memory requests). During page migration, the MM sends messages to pause cores, update TLBs, pin pages in the LLC and flush pages from the LLC. The MM receives acknowledgments from the cores and cache slices to signal the completion of requests. The migration process is explained in more detail in Section 5.3.3.

5.3.2 Cache

The new capability that the cache must provide is the ability to pin cache blocks that belong to a page being migrated by the DMA unit. My mechanism for pinning is based on a new PIN register, which holds the physical page address of the pinned page (see PPN in Figure 11). When the cache receives a message to pin a page, it stores the page number in this register and sets the valid bit (v) to indicate the register holds a pinned page. All blocks belonging to the pinned page are *not* candidates for eviction between page pinning (set v) and unpinning (clear v). Only one page can be pinned simultaneously.

This mechanism is designed to avoid latency impact on normal cache operation; only

the eviction process is modified. Read and write hits to the cache do not involve additional delay because PIN is not consulted during hits, and the tag and data array are unmodified. Pinning a page does affect the eviction procedure, but eviction can be performed in parallel with the memory access that brings new data into the cache. It involves only a few additional comparisons. Therefore, cache eviction with pinning does not add extra latency to a cache fill into the LLC.

Figure 13 shows the architecture of a 4-way set-associative cache and the way it is modified for pinning. Traditionally, addresses are divided in three parts for a cache access. The offset is used to identify what part of the cache block is being requested, while the index is used to select the correct set in the tag array. The tag is used to identify which way holds the correct block, if any. An address can also be divided into its page number and its page offset. Since my mechanism pins the whole page (as opposed to a single block), I keep the page number in PIN. In a typical cache configuration, the number of bits required for the page offset is greater than the number of bits for the cache offset (a page is usually larger than a cache block) and smaller than the number of bits for the cache offset and the cache index (cache size divided by associativity is usually much bigger than page size). Thus, only some of the bits are part of the page number. I term these bits *high index bits* (HIB), and the remaining index bits *low index bits* (LIB). As shown at the bottom of Figure 13, the page number is formed from the cache tag bits and the HIB, while the page offset is composed of the LIB and the cache offset.

When a cache block needs to be evicted, the HIB of the requested address and of the pinned page are compared. If their values are equal and the PIN register's valid bit is set, then this means one block in the cache set is potentially pinned. To determine if a block is actually pinned, the tag of the pinned page is compared with the tags from each way. If one of the tags match, the matching block belongs to the pinned page and should not be evicted. Note that it is unnecessary to access the tag array again because the tags for the set are already available from the original access that missed in the cache.

The result of the HIB and tag comparisons and the LRU bits for the set are fed into the LRU logic, which selects the way to evict. Assuming an LRU stack algorithm, the logic checks whether the LRU block is pinned, as determined by the tag comparisons. If it is, the

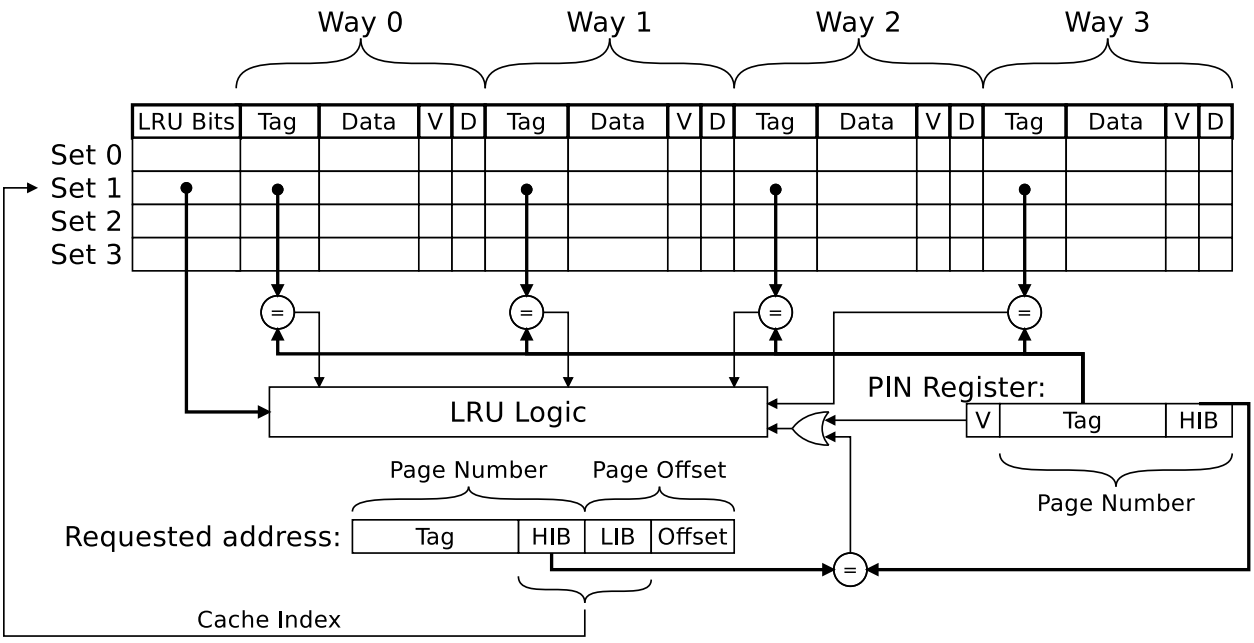

Figure 13: Cache organization with support for page pinning.

eviction algorithm simply chooses the second LRU block instead of the first one.

There are two observations to note about pinning. First, pinning affects only one cache block in a set because only one page is pinned at a time and each block of a page is mapped to a different set[1]. Therefore, there are always other blocks in a set that may be evicted to hold newly requested data. Second, PIN serves to avoid eviction of pinned blocks from the LLC. The caches "above" the LLC (e.g., L1 and L2) behave normally since holding blocks in the LLC does not influence their operation. The coherence protocol for these caches is similarly unaffected and needs no change for CPM.

[1]With the usual cache indexing function.

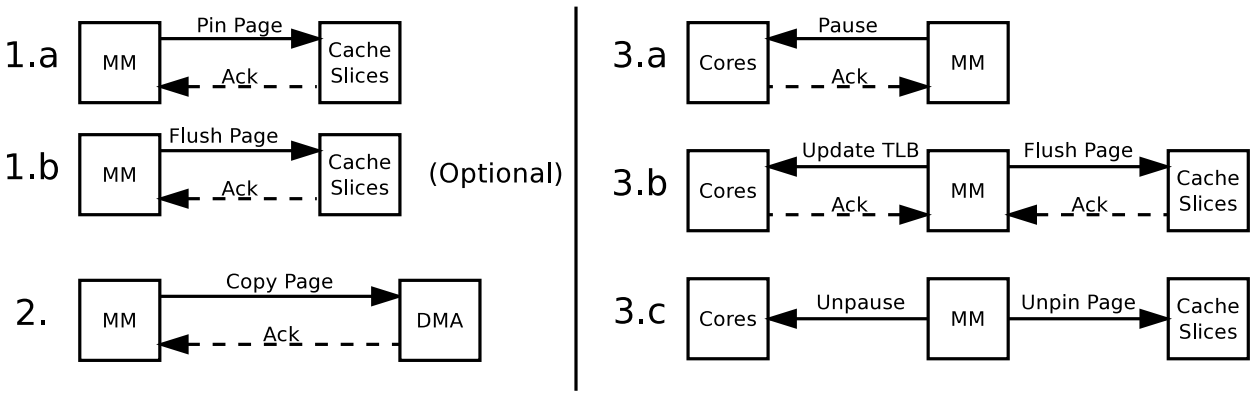

Figure 14: Steps for hardware for CPM.

5.3.3 Concurrent Page Migration

CPM is initiated when the OS writes to the migration queue after determining what pages to migrate. To start a migration, the MM reads and removes the first entry in the migration queue. A migration is characterized by a 5-tuple of process ID, virtual page number, source physical page number, destination physical page number and the address of the PTE in memory. Figure 14 gives an overview of CPM.

The first step pins the source physical page to the cache. To do this, the MM sends one message over the control portion of the interconnect to all cache slices with the physical page number of the page to be pinned (step 1.a). The caches write the page number into the PIN registers and set the PIN valid bits. If the destination page is a PCM address, the MM tells the caches to flush the source physical page (step 1.b).

In the second step, the MM programs the DMA unit to transfer the physical page to its new location. The DMA unit uses the migration buffer to temporarily hold data read from one memory before it is written to the other memory. When the copying finishes, the DMA unit signals the MM that the migration finished.

The third step atomically updates the PTE and TLB and flushes the page from the cache. To do this, the MM sends a message to all cores with the process ID and virtual

address of the page that is being migrated (step 3.a). The contents of this message are written to a core's *Pause on Access* (POA) register and its valid bit is set. On every access, POA is compared to the requested address. Because the register holds a virtual address, the comparison is done simultaneously with address translation to hide the comparison latency.

Once POA is set, if an executing program with the same process ID accesses the virtual address in the POA register, the core executing the program is stalled until the POA register is cleared. This action guarantees the PTE and TLB update and cache flush occur atomically. The application is allowed to execute as long as it only accesses other pages, since pinning prevents evictions of the migrating page caused by cache misses from other pages. After receiving acknowledgments from all cores that they updated their POA, the MM sends a message to all cores with the process ID, virtual page, PTE address and destination physical page of the migrating page (step 3.b). All cores that are executing the process under migration must update the entry, if present, in their TLB and invalidate the PTE from their private L1 caches. The PTE invalidation prevents the page table walker from reading a stale L1 value during TLB misses. The MM also writes the destination physical address into the page table by accessing the PTE directly in the corresponding L2 cache slice. In parallel with the update of the page table and TLB, the MM instructs the cache slices to flush the source physical address of the migrating page (step 3.b). Finally, the MM requests all cores to clear their POA valid bits and, concurrently, instructs the cache slices to unpin the page by clearing their PIN valid bits (step 3.c).

5.4 EVALUATION

This section examines how CPM improves performance and energy of single and multi-programmed workloads in a chip multiprocessor.

Table 8: Architectural parameters

Parameter	Value
2GHz Chip multiprocessor	Four single-issue, in-order cores
L1 I/D private cache	32KB per core, 4-way, 1 cycle hit
L2 unified shared cache	1MB, 16-way, 16 cycle hit, LRU
32MB DRAM memory	50 cycle access
4GB PCM memory	125 cycle read, 1000 cycle write
PCM/DRAM energy ratios	2.1 (reads) and 43.1 (writes)
PCM/DRAM request queues	Separate 16-entry request queues
PCM/DRAM bus	32-bit single-channel at 800MHz

5.4.1 Methodology

To evaluate CPM, I first used Pin [47] to generate a trace of memory references. The trace includes the address of all application loads and stores, as well as the address of every instruction executed. The trace is next input to an in-house cycle-accurate simulator that models the cache hierarchy and memory system. The simulator faithfully accounts for latency and power of all low-level protocols and actions performed by the PCM and DRAM devices, buses, read/write queues, and device parallelism (banks/ranks).

Table 8 shows the main architectural parameters, which are derived from current mobile devices (i.e., smartphones and tablets) [48]. I assume four single-issue in-order cores to allow a larger portion of each benchmark to be evaluated. DRAM and PCM access times are from Qureshi *et al.* [26] and energy values from Lee *et al.* [22]. PCM latencies are adjusted to account for a smaller 64-byte cache block, buffering in the PCM device, and an 8-bank PCM organization. For single-programmed workloads, I assume that each process has one fourth of the shared resources (i.e., each process has a 256KB slice of the 1MB cache, 8MB of the 32MB DRAM, and 4 of the 16 request queue entries). For multi-programmed workloads, I fully model contention for all shared resources, including all request queues.

For the experimental study, I use SPEC CPU2006[2]. While these benchmarks were originally intended for desktop and server systems, they exhibit a wide range of architecture

[2]I could not compile *dealII* and *tonto* due to limitations in the simulation environment.

and memory behavior expected for mobile computers. In particular, prior research identified similarities in memory behavior between SPEC CPU2006 and interactive smartphone applications [49]. In addition, SPEC CPU2006 has a mix of behavior reflecting different kinds of application characteristics [50]. In comparison, older benchmarks, like MiBench and MediaBench, do not put much pressure on the memory sub-system and are not representative of current systems. I treat each input of any benchmark that has multiple data sets as a separate workload, which yields a total of 53 different input/program combinations. In the figures, a number after a benchmark name is the input used for the benchmark (e.g., *bzip2-2* is the 2nd reference input for *bzip2*). I simulate each benchmark for 1 billion instructions. I present results for 13 of the 53 combinations, since the working set of the remaining workloads is small enough to fit in DRAM, thereby avoiding migrations. I also run a multi-core configuration of the selected benchmarks, where four instances of the same benchmark program are run on a separate core in the simulated 4-core chip multiprocessor.

In the experiments, I use the Multi-Queue page migration selection policy, a well-known algorithm that has been shown to perform well in hybrid memory architectures [9]. The parameters of the Multi-Queue policy are the same as in the original paper, except for *LifeTime* (250μs) and *FilterThreshold* (0.25μs), which were changed to reflect the architectural parameters of a mobile system. The baseline (PAUSE) consists of an OS-only system without CPM that migrates pages by pausing the application on writes to migrating pages. I model the behavior of CPM and all its latencies, including all costs associated with pausing the application during atomic operations, flushing the caches, buffering and contention for buffer space, and updating the TLBs on multiple cores.

5.4.2 Single-Programmed Benchmarks

Figure 15 shows execution time normalized to OS-only without CPM for 13 of the single-programmed workloads. The figure also shows the maximum potential speedup (NO COST), which is calculated by assuming a migration cost of 0 cycles. As the figure shows, CPM always has a performance gain, with speedup ranging from 1.05 (*gobmk-2*) to 1.22 (*sjeng*) and an average of 1.16. This gain happens because applications can continue to execute (i.e.,

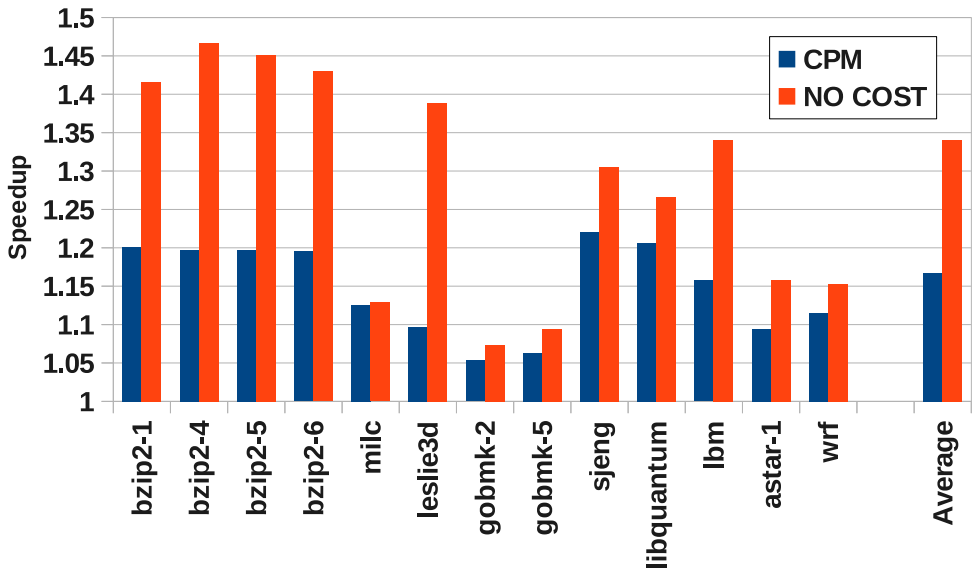

Figure 15: Single-programmed: Speedup.

write) during long latency DRAM to PCM migrations. Some benchmarks have an especially large improvement, such as *bzip2-1*, *sjeng* and *libquantum*. These results are explained by two factors. First, these applications have frequent migrations. For example, in *libquantum*, stalls due to migration in the baseline account for approximately 28% of total execution time. These stalls are avoided with CPM. Second, the baseline requires that the migrated page is flushed from the cache prior to a migration to DRAM. Assuming that the page is frequently written, it is likely that most of its cache blocks are dirty, causing more blocks to be flushed. CPM does not require this flush before migration, which reduces pressure on the PCM write buffer and application stall time.

5.4.3 Stall Behavior

To gain insight into the gains of CPM, I further examined the source of program stalls for single-programmed workloads. Figure 16 shows the number of cycles that each application

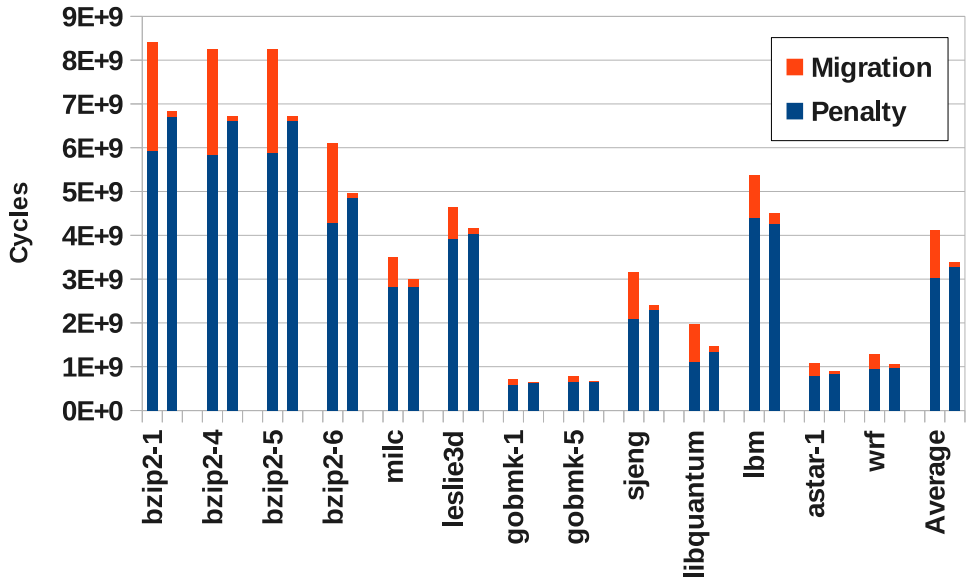

Figure 16: Single-programmed: Number of cycles waiting. First bar is baseline and second bar is CPM.

stalls waiting for (a) migrations to finish (labeled *Migration* in the graph), and (b) read operations to the cache hierarchy and memory system (labeled *Penalty*). The first bar in each set corresponds to the baseline, and the second to CPM.

A taller bar means an application stalls longer, and the large proportion of migration stalls suggests that applications pause frequently due to ongoing migrations. CPM improves performance because it reduces the cycles waiting for migrations. For example, in *libquantum*, the number of cycles spent waiting for migrations is reduced from 1.1 billion to 98.8 million. In some cases, such as *libquantum* and all *bzip2* inputs, the memory penalty increases slightly under CPM. This is because page migrations occupy a fraction of the memory bandwidth to PCM, which delays application memory accesses that happen concurrently under CPM. However, the increased penalty is not large enough to offset the reduction in stall cycles due to migrations. For example, the cache hierarchy penalty of *bzip2-6* increases from 4.2 billion cycles in the baseline to 4.8 billion cycles with CPM (14% increase), while the time waiting

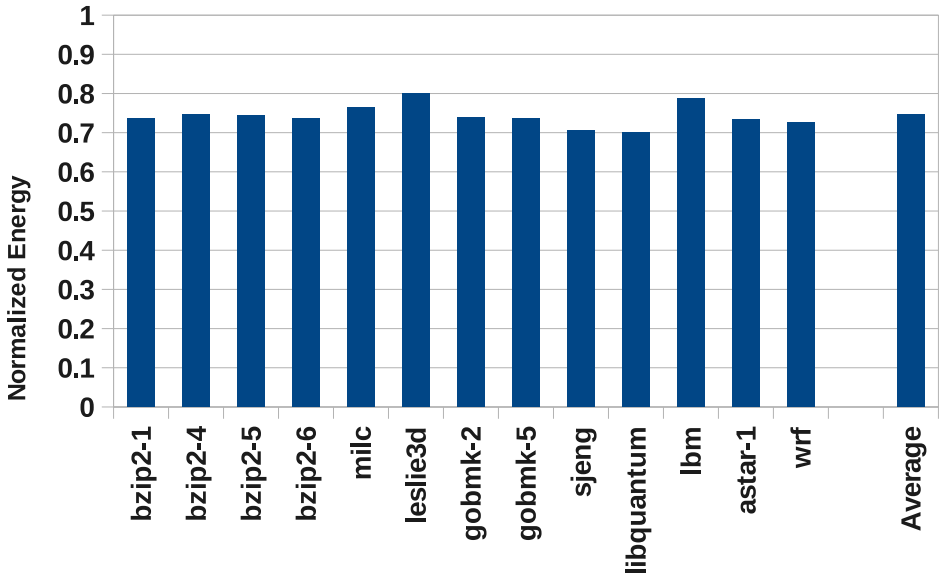

Figure 17: Single-programmed: Energy consumption.

for migrations is reduced from 1.8 billion cycles to less than 100 million cycles.

5.4.4 Energy

Figure 17 shows memory energy for the single-programmed workloads. The values in the graph are normalized to the baseline's memory energy: a smaller number (< 1.0) is a larger improvement over the baseline. The memory energy includes the dynamic energy for DRAM and PCM accesses, but it is conservative in that it does not consider savings in program execution time. Energy for each type of access (read or write) and memory (DRAM or PCM) is calculated by multiplying the number of accesses of that type to that type of memory by the energy per access for that access and memory type (see Table 8). Total energy is the sum of all four access type/memory type combinations.

As the graph shows, the normalized energy for CPM varies from 0.70 (in *libquantum*) to 0.8 (in *leslie3d*), with an average of 0.75. This large reduction happens because CPM

allows the migration policy to have a more up-to-date view of application memory access patterns, which leads to better migration decisions. For example, in *libquantum*, the fraction of memory accesses that are serviced by DRAM increases from 50.7% in the baseline to 58.4% for CPM. In general, migration policies that rely on recency information to make migration decisions can greatly benefit from CPM because it allows applications to continue gathering access information during migrations, which prevents pauses from interfering with the migration algorithm. Multi-Queue is particularly susceptible to long pauses because pages become eviction candidates when they are not accessed during a pre-defined interval of time.

5.4.5 Sensitivity to Migration Cost

I artificially varied the cost of migration to analyze the behavior of my scheme with different hardware configurations. Figure 18 shows the average speedup of all 13 single-programmed workloads for different migrations latencies. The range of latencies effectively accounts for different amounts of memory ranks and banks (parallelism) and write buffering: the lowest cost corresponds to significant parallelism and buffering, while the highest cost corresponds to minimal parallelism and buffering. The X-axis shows the number of cycles required to migrate a page from DRAM to PCM. The migration cost from PCM to DRAM is one half the migration cost to PCM.

The average speedup is lowest at both ends of the range, and highest toward the center. For the lowest and highest migration costs (1,000 cycles and 64,000 cycles), the average speedup is 18% and 2%, respectively. The highest average speedup is 35% for a migration cost of 4K cycles. In general, a low migration cost has a small gain for CPM because applications are paused for short periods of time. In addition, when migrations are fast, the probability of the application writing to a page under migration is low. These two factors reduce the amount of time that CPM can save by allowing writes to proceed while migrations are under way.

For values larger than 4,000 cycles, speedup decreases because long migrations prevent the system from reacting quickly to changes in access behavior, which extends the period

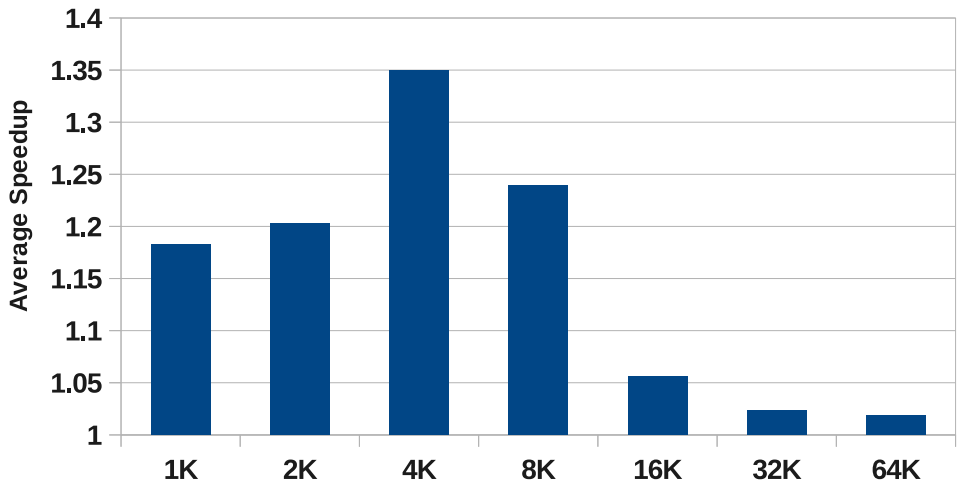

Figure 18: Average speedup of CPM with single-programmed workloads for different migrations costs in cycles.

of time in which pages are assigned to memory resources in a sub-optimal way. Since CPM allows writes to continue, more memory accesses will be performed under the sub-optimal assignment, reducing performance.

The difference between the average speedup in Figures 15 and 18 is due to the actual cost of migration not being exactly 8K cycles. In my simulations, migration cost is 8,750 cycles on average due to contention in the memories and buses from regular accesses.

From these results, I conclude that CPM will help even when the system has significant memory parallelism, but the approach is best suited for typical mobile systems that fall in the middle range of effective migration latency.

5.4.6 Multi-Programmed Workloads

Figure 19 shows the speedup of multi-programmed workloads normalized to OS-only without CPM. The speedup ranges from 1.00 (*milc*) to 1.13 (*bzip2-1*), with an average of 1.08. CPM is still effective for multi-programmed workloads, although the gains are generally smaller

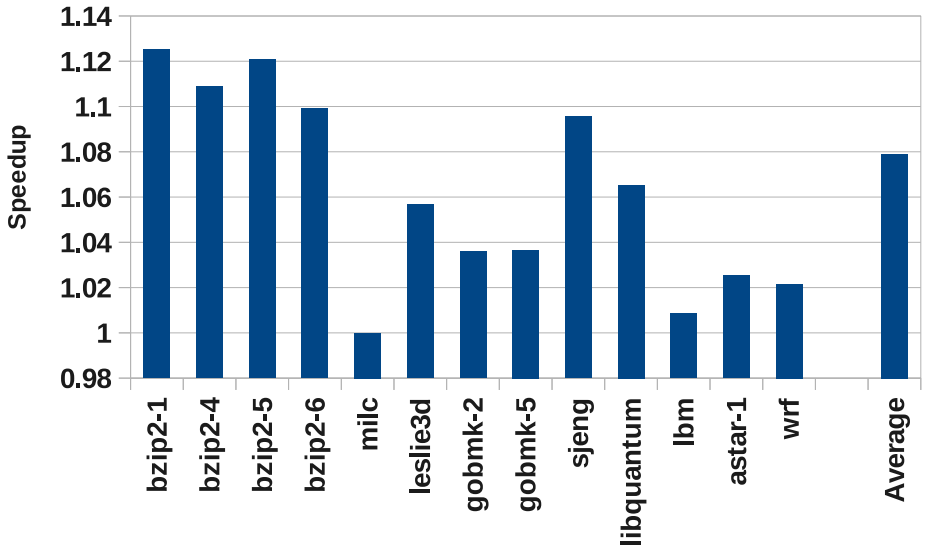

Figure 19: Multi-programmed: Speedup.

than for their single-programmed counterparts. This is because CPM only migrates one page at a time. During a given migration, only one of the four applications running may pause due to a write to a page under migration.

Figure 20 shows the stall time behavior of multi-programmed workloads. The difference in stall cycles between single and multi-programmed applications shows the impact of doing only one migration at a time. On average, multi-programmed workloads spend only 9% of stall cycles waiting for migrations in the baseline, while single-programmed workloads spend 27%. For *milc*, CPM does not improve performance for multi-programmed workloads at all because the number of stall cycles due to migrations in the baseline is only 5% of all stall cycles. In addition, the Multi-Queue migration policy is less effective under CPM, resulting in a higher average memory access time that is not offset by the reduction in migration stall time.

As these results show, CPM is also effective for multi-programmed workloads, achieving

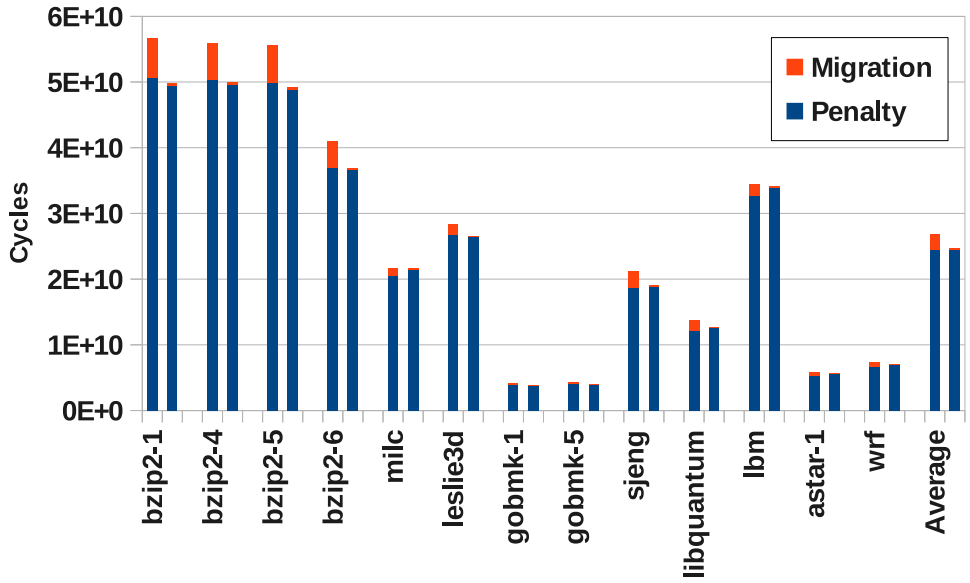

Figure 20: Multi-programmed: Number of cycles waiting.

8% average speedup with only one migration at a time. With more DMAs, memory channels and additional support for multiple concurrent migrations and smarter migration policies, an even better improvement in performance is possible, as shown in Chapters 6 and 7.

5.5 SUMMARY

This chapter introduced simple hardware support to mitigate the cost of page migration. Concurrent page migration (CPM) is proposed to pin the contents of pages under migration in the last-level cache, which avoids the need to stall an application. It is demonstrated that pinning reduces page migration overhead: The performance of single-programmed applications is increased by 17% (average) with CPM and the performance of multi-programmed workloads is increased by 8% (average).

6.0 CONCURRENT MIGRATION OF MULTIPLE PAGES

CPM, presented in Chapter 5, allows applications to continue executing ever after writing to migrating pages. However, in CPM, only one page can be under migration at a given time. A single migration can become a bottleneck in multi-core systems, where several independent applications may require migration at the same time. Although CPM can be augmented to support multiple migrations, the number of concurrently pinned pages is ultimately constrained by the structure of the cache: if more pages are pinned than there are ways in a cache set (typically 16), the eviction algorithm will not be able to find an eviction candidate, and the application will have to be paused. In addition, cache performance might be impacted before this limit is reached because the cache eviction algorithm will be forced to make non-optimal eviction decisions. This chapter proposes hardware support to overcome this limitation. My technique enables *multiple simultaneous migrations*, while allowing applications to continue execution during migration, even during writes.

6.1 CONCURRENT MIGRATION OF MULTIPLE PAGES

Concurrent Migration of Multiple Pages (CMMP) is a hardware-software co-designed scheme that provides software-managed hybrid memory the ability to migrate multiple pages at the same time. CMMP allows writes to migrating pages by keeping track of which blocks of a page have already been copied to the destination. On an access to a block, the state of the block is consulted and the access is redirected to the appropriate location. CMMP inherently supports multiple concurrent migrations by tracking the block states of multiple pages.

CMMP has four parts: Concurrent Migration (CM), Access Redirection (AR), On-

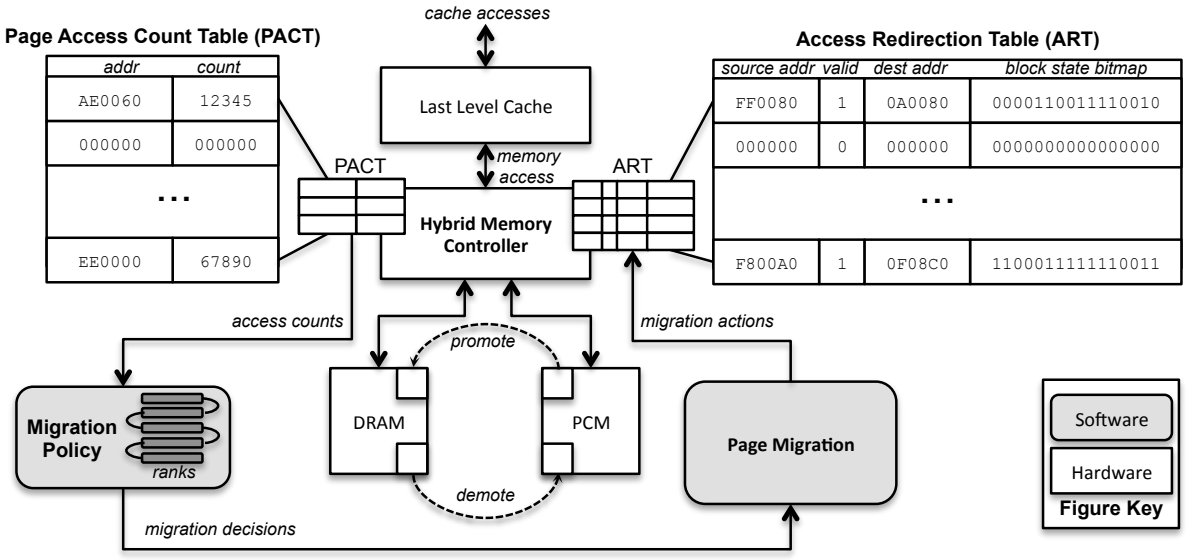

Figure 21: Overview of software and hardware components for CMMP.

demand Block Migration (OBM) and Partial Demotion (PD). CM and AR provide the ability to concurrently migrate multiple pages while allowing applications to continue executing during migration. OBM allows migrations to transfer blocks as they are accessed by the application, minimizing memory system interference. PD allows partially migrated pages to be selected for migration back to PCM, also reducing interference.

Figure 21 shows the design of CMMP. Access counts are collected by hardware, using the Page Access Count Table (PACT). The contents of PACT are periodically read by the OS and passed to the *Migration Policy*. The migration policy, which is implemented in software as part of the OS, ranks pages according to access counts. The policy decides what pages to promote from PCM to DRAM and what pages to demote from DRAM to PCM. Page migration is performed by the Hybrid Memory Controller, which copies pages between memories and redirects memory accesses using the Access Redirection Table (ART) to keep track of the state of migrating pages.

6.1.1 Migration Policy

The migration policy is run periodically to generate a list of candidate pages for migration. To make promotion and demotion decisions, the migration policy categorizes memory pages based on access count and recency.

For CMMP, we use the threshold queue of Multi-Queue as the criterion to categorize hot and cold pages. Hot pages that are in PCM are candidates for promotion, and cold pages that are in DRAM are candidates for demotion. Once a page becomes a promotion candidate, it is migrated concurrently with other pages. Demotion copies cold data back to PCM to keep enough free DRAM space for performing promotion of hot PCM pages. Demotion candidates are migrated one at a time starting from the lowest ranked page until there are no more candidates, or more than *FreeThreshold* DRAM capacity is available (*FreeThreshold* is a parameter). We note that CMMP is not limited to Multi-Queue. In fact, any algorithm that can rank pages and categorize them based on coldness/hotness can be used with CMMP.

To support Multi-Queue and similar migration policies, CMMP maintains post-LLC access count information. To gather page counts, PACT tracks accesses to a *subset of pages*. The table is indexed by physical page number; each entry in the table stores how many times the page has been accessed. The table is accessed on LLC misses and writebacks outside of the critical path of delivering cache hits to the processor. When accessed, if a page is not found in PACT, a new entry is allocated and the count set to 1. A count of zero is used to identify an invalid entry. When the table becomes full, the entry with the lowest count is evicted. Periodically, the OS reads the contents of PACT and resets the table. The gathered information is used by the migration policy.

6.1.2 Concurrent Migration

CMMP uses ART, a special data structure to enable concurrent migration. ART has one entry for each ongoing migration. ART entries are inserted by the OS in response to decisions from the migration policy. After inserting a new entry, the hybrid memory controller copies the page from source to destination, relying on ART to track the state of each block.

Algorithm 4 PCM to DRAM migration (promotion)

```
PROMOTE(pcmPage)
1  if (¬ART.LOOKUP(pcm_page))
2      entry ← ART.INSERT(pcmPage)
3      entry.valid ← 1
4      entry.dest ← dramFreeList.GETPAGE()
5      entry.CLEARSTATEALL()
```

For PCM to DRAM migrations (promotion), blocks are not transferred immediately to the destination but copied *on-demand* as they are accessed (OBM, see Section 6.1.3). As a results it is possible that some pages are only partially copied to DRAM at a given point of time. For DRAM to PCM migrations (demotion), blocks are copied immediately. OBM is not used for page demotion because pages are cold, and thus, unlikely to have blocks copied as they are accessed.

ART is a hardware table indexed by physical address of the source page. Each entry contains a valid bit, destination page number and block state bitmap, which has the location of the most current version of each block in a page (0 means the block is in source location and 1 means the block has been copied to the destination). The page addresses are 36 bits (256TB physical address space) and the bitmap is 64 bits (4KB page and 64B blocks) for a total of 136 bits per entry.

Next, we describe how promotion and demotion operate, using ART.

6.1.2.1 Promotion Algorithm 4 shows the steps for PCM to DRAM migration. This algorithm is run for every PCM page that becomes hot (i.e., crosses the threshold queue) during periodic execution of the migration policy, as long as there are free DRAM pages and free ART entries. If the page is not currently being migrated (no allocated entry in ART), then a new entry in the table is allocated (lines 2 to 3). A free DRAM page is obtained from the free list (line 4) and the block state of all blocks is set to not copied (line 5). Note that the page is not copied immediately; instead, CMMP relies on OBM to copy the blocks as they are accessed by the application (see Section 6.1.3).

Algorithm 5 DRAM to PCM migration (demotion)

```
DEMOTE(dramPage, pcmPage)
 1  // Insert new entry
 2  entry ← ART.INSERT(dramPage)
 3  entry.valid ← 1
 4  entry.dest ← pcmPage
 5  // Lookup old PCM to DRAM migration
 6  oldEntry ← ART.LOOKUP(pcmPage)
 7  if (oldEntry.valid)
 8      // Old entry exists: copy state and remove old entry
 9      entry.state ← COMPLEMENT(oldEntry.state)
10      oldEntry.valid ← 0
11  else
12      // No old entry: entire page was copied to DRAM
13      entry.CLEARSTATEALL()
14  // Issue copy of blocks in DRAM to PCM
15  for (b ← 0; b ≤ MAXBLOCKS; b ← b + 1)
16      if (entry.GETSTATE(b) == 1)
17          COPY(BLOCKADDR(dramPage, b), BLOCKADDR(pcmPage, b))
```

6.1.2.2 Demotion Figure 5 shows the algorithm for DRAM to PCM migration. This algorithm is invoked after the migration policy has determined the next page to demote. DRAM pages that were previously allocated to PCM can reuse their original PCM page. This allows writes of clean blocks to proceed at read speed using write minimization [25]. The algorithm receives as the second parameter either the original PCM page or a free PCM page.

Due to OBM, it is possible that pages are not fully migrated to DRAM when scheduled for demotion back to PCM. CMMP provides support for Partial Demotion (PD), which copies back to PCM only those blocks that have been migrated to DRAM. PD uses the block state information in ART to identify the blocks to copy. After allocating and initializing an entry in ART, the algorithm checks whether the original page (PCM) is still being promoted (Algorithm 5, lines 6 to 7). If so, the block state is bitwise complemented and copied to the new ART entry and the old one is removed (lines 9 to 10). The bitwise complement of the state guarantees that blocks that were not copied to DRAM as part of promotion are not copied back to PCM (lines 15 to 17), reducing memory bandwidth.

6.1.3 On-Demand Block Migration

Page migration is usually done at the granularity of pages: a whole page is allocated and copied from source to destination. This effectively prefetches a page into DRAM, which can improve performance if most of the page is accessed (spatial locality). However, copying a whole page can have a detrimental impact on performance due to significant interference with regular application requests [18].

To reduce migration bandwidth, we introduce On-demand Block Migration (OBM) for CMMP to copy blocks *as they are accessed*. OBM reduces pressure on the memory system in two ways: 1) only blocks that are requested are copied, and 2) bursts of traffic injected by copying the whole page are avoided and spread over more time, effectively giving higher priority to application requests.

OBM uses the block state bitmap in ART to determine the location of the most current copy of a block. On a read access, if the block has not been copied, it is read from the source memory, delivered to the LLC and written to the destination memory. If the block has already been copied, it is read from the destination memory and delivered to the LLC. On a write access, the block is written directly to the destination memory regardless of the block state (the latest version is in the LLC).

6.1.4 Access Redirection

To allow LLC misses and writebacks to pages under migration to be redirected to their correct location, we incorporate Access Redirection (AR) in CMMP. With AR, migration is transparent and applications can continue executing without pausing, even during writes to migrating pages. AR works after the LLC, inherently supporting multiple cores. Blocks are cached as usual and there are no changes to the caches.

Algorithm 6 shows the steps for AR, which uses ART to determine the current location of blocks. For every LLC miss or writeback, ART is checked to see if the page is currently being migrated. If not, the access is redirected to the corresponding memory based on the physical address of the request (lines 4 to 7). Otherwise, another action is performed depending on the source page of the request, the type (read vs. write) and the state of the block (copied

Algorithm 6 Access Redirection

```
AccessRedirect(addr, read, data)
 1  page ← PageNum(addr)
 2  entry ← ART.LookUp(page)
 3  if (¬entry.valid)
 4      if PCM_lowpage ≤ page ≤ PCM_highpage
 5          return pcm.access(addr, read, data)
 6      else
 7          return dram.access(addr, read, data)
 8  else
 9  block ← BlockNum(addr)
10  if PCM_lowpage ≤ page ≤ PCM_highpage
11      dramAddr ← Address(entry.dest, block)
12      if (read)
13          if entry.getState(block)
14              return dram.access(dramAddr, true)
15          else
16              entry.setState(block)
17              d ← pcm.access(addr, true)
18              return dram.access(dramAddr, false, d)
19      else
20          entry.setState(block)
21          return dram.access(dramAddr, false, data)
22  else
23      pcmAddr ← Address(entry.dest, block)
24      if (read)
25          if entry.getState(block)
26              return pcm.access(pcmAddr, true)
27          else
28              return dram.access(addr, true)
29      else
30          entry.setState(block)
31          return pcm.access(pcmAddr, false, data)
```

or not copied). Writes are always redirected to the destination (lines 21 and 31). Reads are redirected only when the block has already been copied to the destination (lines 14 and 26). Otherwise, the source location is accessed (lines 17 and 28). OBM happens when reading a block from PCM that has not yet been copied. It reads the block from PCM, sets the state of the block to copied, writes it to DRAM and delivers it to the LLC (lines 16 to 18).

Table 9: Architectural parameters

Parameter	Value
4GHz chip multiprocessor	4 4-issue wide, out-of-order cores, 128-entry reorder buffer
L1 I/D private cache	64KB per core, 4-way, LRU, 3 cycle hit, 16-entry queue
L2 unified shared cache	8MB, 16-way, LRU 32 cycle hit, 32-entry queue
128MB DRAM memory @ 1000MHz	64 banks, 32-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-12-12 (ns) Array energy (pJ/bit): 1.17 (reads), 0.39 (writes) Buffer energy (pJ/bit): 0.93 (reads), 1.02 (writes)
4GB PCM memory @ 400MHz	128 banks, 8-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-55-150 (ns) Array energy (pJ/bit): 2.47 (reads), 16.82 (writes) Buffer energy (pJ/bit): 0.93 (reads), 1.02 (writes)
PCM/DRAM bus	64-bit single-channel

6.2 EXPERIMENTAL RESULTS

6.2.1 Methodology

For our experimental evaluation, we use HMMSim, a hybrid main memory simulator [18]. The main architectural parameters are shown in Table 9. We model a multi-core system with private L1 caches and a shared L2 cache. DRAM and PCM latencies and energy values are from Lee *et al.* [22]. For PCM, t_{RP} is 0 for clean row buffers (due to non-volatility) and 150ns for each dirty block in the buffer (to account for power constraints in the PCM chip).

CMMP has several parameters, which we chose empirically through experimentation. *FreeThreshold* is the maximum fraction of free DRAM space available for page promotion. We saw little variation in performance and energy for this parameter, and set it to 1% of DRAM capacity for all experiments. We set the period of reading the PACT to 1ms, a typical OS interrupt period. We chose a size of 8k entries for the PACT, which is enough to hold the counts of most pages during a 1ms interval. Performance is highly sensitive to the size of ART. Thus, we chose a size of 32k entries, enough to fit all migrations to DRAM.

Both tables have an associativity of 32. We used CACTI 5.3 to estimate the energy of PACT and ART to be 0.01nJ and 0.06nJ per access, respectively. Access latency is 0.6ns for PACT and 1.3ns for ART. Since PACT and ART are only accessed on LLC misses and writebacks, the additional energy of accessing them is very low compared to DRAM and PCM energy (less than 1% on average). Similarly, the latency of accessing PACT and ART is negligible compared to the average memory access latency (77ns on average). We note that the energy and performance effects of accessing PACT and ART are included in all our results.

For workloads, we use a subset of SPEC CPU2006 benchmarks run in rate mode as a multi-programmed workload. We treat each input of the same benchmark as a separate workload (in the figures, a number after the benchmark name is the input for the benchmark). Each workload is run for 1 billion instructions per core after a warm-up phase of 5 billion instructions. Each experiment consists of 4 copies of the same workload run in parallel on our simulated 4-core system. To fully stress migration, we consider only multi-programmed workloads with working set sizes greater than the available DRAM memory (128MB). Programs with a low LLC miss rate (lower than 2 MPKI) are not reported because they put very little pressure on the memory system. Their results are the same as the baseline because they do not benefit from migration.

For our experiments, we assume a baseline system running CPM [16]. We compare it against three systems that incrementally implement each of the techniques of CMMP: 1) **CM** migrates multiple pages concurrently by copying them as fast as possible. CM uses access redirection to avoid pauses. 2) **CM+OBM** incorporates CM but copies pages on-demand as they are used by the application. 3) **CM+OBM+PD** incorporates CM+OBM and also allows for partial demotion of pages that have not been fully migrated to DRAM. All systems, including the baseline, use the Multi-Queue migration policy with the original parameters [9].

6.2.2 Performance

Figure 22 shows the speedup of each of the techniques of CMMP against the baseline system for all benchmarks. On average, CM alone performs around 19% worse than the baseline,

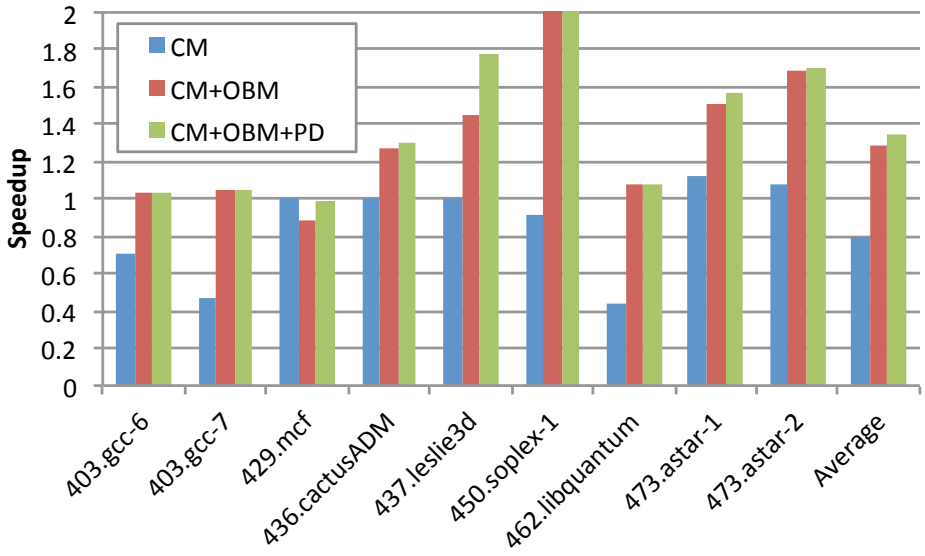

Figure 22: Speedup of CMMP, normalized to the baseline.

with four workloads having a slowdown of more than 10% (*403.gcc-6*, *403.gcc-7*, *450.soplex-1* and *462.libquantum*). In these workloads, CM migrates too many pages at the same time without controlling the interference caused by migration, which slows down regular application requests. Two workloads have speedups with CM: *473-astar-1* (8% speedup) and *473.astar-2* (12%) can increase their share of DRAM requests (from 14% to 26%) with fewer page migrations, reducing interference. For other workloads, additional migrations do not appreciably change the fraction of regular requests serviced from DRAM (52% vs. 49%), resulting in no speedups.

When CM and OBM are enabled, performance increases to 28% on average. The bulk of this performance improvement comes from 5 workloads that have more than 25% speedup (*436.cactusADM*, *437.leslie3d*, *450.soplex-1*, *473-astar-1* and *473.astar-2*). In these benchmarks, the fraction of DRAM requests stays relatively constant. However, the average service time of regular requests to PCM is reduced considerably (from 190ns to 70ns) due to migration of blocks on demand over a large period of time. For the remaining workloads, the reduction in average PCM access time is also significant (140ns to 61ns). However, the

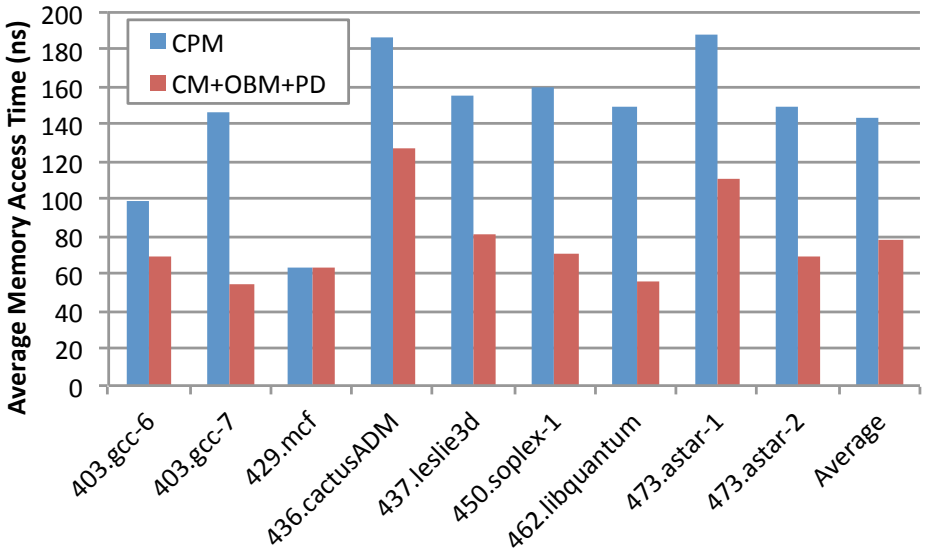

Figure 23: Average memory access time of CMMP.

average DRAM access time increases considerably (from 39ns to 96ns) because CM+OBM does not reduce DRAM bandwidth used by migration. Thus, the speedups are low for these workloads.

Performance is further improved to 35% on average when using PD. Benchmarks with low speedups under OBM benefit little from PD because they suffer mostly from DRAM contention, which PD does not help alleviate. For workloads that do benefit, performance improves because demotions take less time and some writes to PCM can be elided, further reducing contention (average PCM access time is reduced from 70ns to 66ns). In addition, PD releases cold DRAM pages earlier that can be used as destination for hot PCM pages, allowing it to react more quickly to changes in application behavior (fraction of DRAM reads is increased from 18% to 26%).

6.2.3 Average Memory Access Time

To gain insight into the performance of CMMP, we examined the average access time of requests that reach the hybrid memory. Figure 23 shows the average memory access time of requests as seen by the LLC for both the baseline and CMMP. The average access time includes both DRAM and PCM components. The graph includes requests made by the application and writebacks caused by evictions during regular cache operation as well as by cache flushes due to migration. Migration traffic is not included to highlight how the workload is affected by migration.

CMMP reduces the average memory access time of almost all workloads (on average, from 144 to 77ns). As expected, the reduction is small for benchmarks that have little or no speedup (see Figure 22). The exception is *403.gcc-7*, which has a reduction in average access time of 64% but only a 5% speedup. This discrepancy is due to a very high LLC miss rate caused by flushes due to migration, which reduces the average cache access time as seen by the CPU. The only workload that keeps its access time constant is *429.mcf*, which has the lowest speedup. For this benchmark, additional page migrations do not increase the fraction of DRAM accesses or increase contention at the memories, resulting in similar performance to the baseline. For all other workloads, the reduction is considerable (in some cases exceeding 50%) and is accompanied by large performance gains.

6.2.4 Energy

Figure 24 shows dynamic energy consumption of DRAM and PCM memories for CMMP normalized to the baseline. On average, energy consumption is reduced by 26%. Migration of hot pages to DRAM steers requests away from PCM, which receives 60% less accesses on average. Although DRAM requests and energy increase considerably (both by 3.3x on average), the largest proportion of access come from PCM, resulting in an overall reduction in energy.

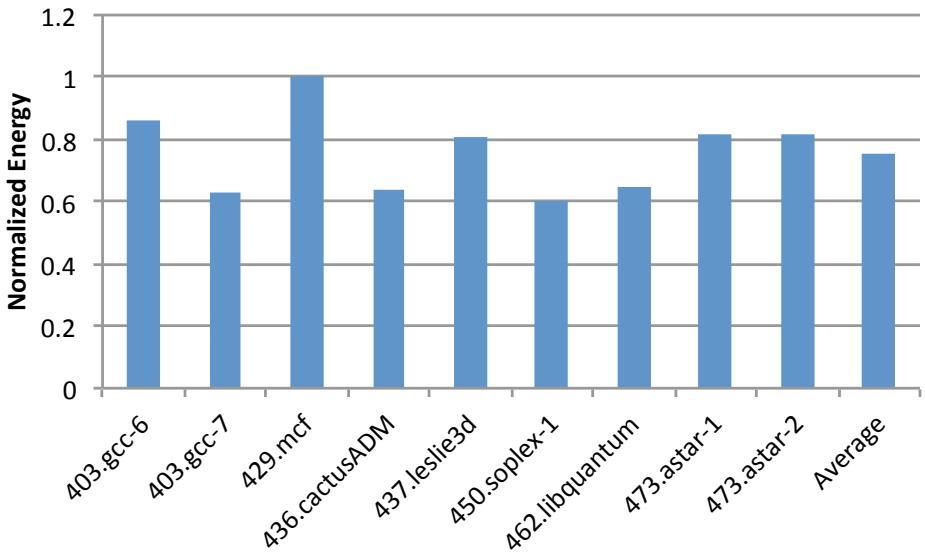

Figure 24: Energy of CMMP, normalized to the baseline.

6.3 SUMMARY

This chapter proposes a mechanism for efficient concurrent migration of multiple pages in software-managed hybrid memory. The mechanism reduces contention at the memory system caused by migration and provides a simple and elegant interface for communicating access counts to software. CMMP reduces migration bandwidth by copying blocks on demand as they are accessed by applications and by eliding the transfer of untouched blocks during migrations to PCM. An evaluation of the proposed mechanisms using multi-programmed benchmarks of different memory access characteristics show that it can improve performance by 14% on average, while reducing energy consumption by 29%.

7.0 THRESHOLD MIGRATION POLICY

The characterization of overhead presented in Chapter 4 showed that software-managed hybrid main memory has great potential for achieving high performance. The study showed two possible ways of achieving this. The first one consists of improving the efficiency of page migration, and was studied in Chapters 5 and 6. The second one consists of improving the choice of pages to migrate, and is studied in this chapter.

This chapter proposes a new migration policy, called Threshold Migration Policy (TMP), that builds upon hardware mechanisms proposed in Chapter 6: OBM and PD. TMP was designed specifically to make better use of these mechanisms than existing policies. In particular, TMP includes an algorithm for making decisions about pages that are currently under on-demand migration. In contrast, existing policies simply treat these pages as regular DRAM or PCM pages and use their promotion or demotion algorithms to manage them.

7.1 OVERVIEW

To reason about policies that use OBM and PD, I consider pages to be in one of three possible states, shown in Figure 25. Since OBM transfers pages as they are accessed by the application, pages can be partly in DRAM and partly in PCM for long periods of time (*Migrating* state). The figure also shows the transitions between the states. A page in state PCM is changed to *Migrating* when the application accesses one of its blocks, which results in the *start* of migration. Pages in state *Migrating* can be selected to either *complete* their migration to DRAM or *rollback* their migration to go back to PCM. Pages in DRAM can be *demoted* to PCM. The goal of a policy is to select pages to go through the complete,

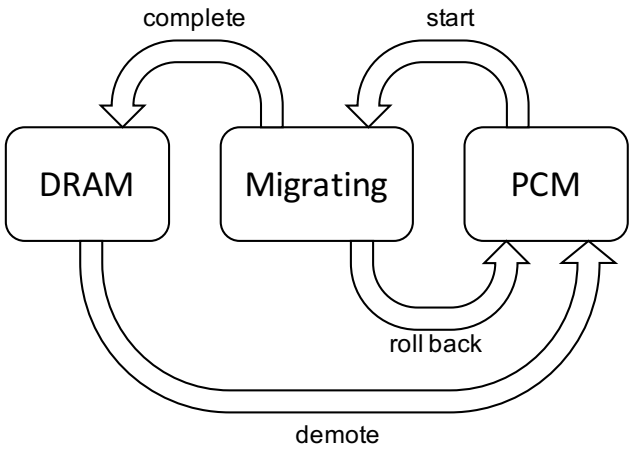

Figure 25: Possible state of pages that use OBM and PD.

rollback and demote transitions. Only one page can be selected for each transition at a time. The policy does not select pages for the start transition. Instead, pages go through the start transition when the application accesses them while in PCM, triggering a migration.

The main idea behind TMP is to count the number of accesses to each page and partition the pages into two sets based on their access counts. The first set has N pages, where N is the number of pages that fit in DRAM. This set contains the pages with the highest access counts. The second set contains all remaining pages. Note that pages in the first set do not necessarily reside in DRAM, and pages in the second set do not necessarily reside in PCM. The sets are used by TMP to identify pages that are good candidates for promotion (pages in the first set that are not in DRAM) and demotion (pages in the second set that are in DRAM).

More specifically, TMP selects pages in the first set that are not in DRAM as *candidates* for completion. Among those candidates, the page with the highest block transfer rate (measured as the number of blocks migrated divided by the time since the start of migration) is selected for completion. Similarly, TMP selects pages in the second set that are not in PCM as candidates for demotion, and the least recently used page among the candidates is demoted. Lastly, the rollback page is selected as the least recently used page that is not in

the first set.

7.2 THRESHOLD MIGRATION POLICY

This section describes the TMP in detail. First, I describe the monitoring information that the algorithm requires and how it is collected by the hardware. Next, I describe the data structures used by the OS to keep this information for easy access by the algorithms. Last, I describe the completion, demotion and rollback algorithms.

7.2.1 Monitoring

TMP uses data about how applications access memory and about the progress of migration to inform its decisions. This data is collected by hardware using the mechanisms presented in Chapter 6. The data used by the migration policy can be classified into two categories. The first category is the number of accesses to each page, which is kept by hardware in the PACT table (see Section 6.1). The second type of data is the progress information about the ongoing migrations. This data is kept by hardware in the ART table, which holds information about the status of each block in the page. These tables are read periodically by the OS to update its own data structures.

7.2.2 Data Structures

TMP uses five different data structures to keep track of the monitoring data read from the ART and PACT tables: an LRU list of all pages in DRAM that are candidates for demotion, called the Demotion Candidate List (DCL); an LRU list of all pages under migration that are candidates for rollback, called the Rollback Candidate List (RCL); a list, ordered by rate of progress (decreasing), of all pages under migration that are candidates for completion, called the Completion Candidate List (CCL); an binary tree of all page ordered by their access counts (decreasing), called the Access Count Tree (ACT); and a hash table of all pages with pointers to the nodes of the preceding lists and tree, called the Page Hash Table (PHT).

The ACT is used to separate pages into two sets. The High Access Count (HAC) set contains the pages with the highest access counts. It has exactly N pages, where N is the number of DRAM pages. The Low Access Count (LAC) set contains all remaining pages. The ACT keeps track of these sets by using a pointer (High Access Count Pointer, or HACP) to the Nth node in the tree. Pages that come before this pointer are in HAC and pages after it are in LAC. When the access count of a page is updated, the pointer is updated accordingly to maintain the invariant that it points to the Nth node.

The PHT is used to gain quick access to the nodes of a page in the DCL, RCL, CCL and ACT without having to traverse those data structures. For instance, when processing an entry from PACT, the page is looked up int the PHT to get the pointer to the page's node in the DCL. The node can then be removed from the list and inserted in the front, keeping the LRU invariant. For the case of the ACT, the PHT also keeps a flag indicating whether the page in the HAC or LAC sets.

Algorithm 7 shows the steps for updating the lists and tree structures when page monitoring information is read from the PACT. First, the page is looked up in the PHT. Second, the page's entries in the DCL and RCL lists, if present, are moved to the front. Third, the page's node in the ACT tree is removed and inserted in the new location (based on the new access count of the page). If the page's new count crosses into the HAC, the HACP pointer is updated to point to the previous node in the tree to keep the number of pages in the HAC constant. The algorithm takes care of the special case when the page's node to be updated is the one pointed to by the HACP.

To update the CCL (Algorithm 8), TMP reads the contents of the ART and creates a list of all pages currently under migration to DRAM and the number of blocks that have been migrated for each page. The number of migrated blocks is computed by counting the number of ones in the block state bitmap, where a one indicates that a block has been migrated. The CCL is obtained by ordering this list by decreasing number of blocks migrated.

Algorithm 7 Algorithm for updating data structures in TMP when page monitoring information is read from the PACT

```
UPDATE(page, count)
 1  entry ← PHT.LOOKUP(page)
 2  if entry.DCL_pointer ≠ NIL
 3        DCL.ERASE(entry.DCL_pointer)
 4        entry.DCL_pointer ← DCL.INSERT(page)
 5  if entry.RCL_pointer ≠ NIL
 6        RCL.ERASE(entry.RCL_pointer)
 7        entry.RCL_pointer ← RCL.INSERT(page)
 8  entry.count ← entry.count + count
 9  if entry.isHAC
10        if entry.ACT_pointer == HACP
11              HACP ← ACT.PREV(HACP)
12        ACT.ERASE(entry.ACT_pointer)
13        entry.ACT_pointer ← ACT.INSERT(entry.count, page)
14  else
15        ACT.ERASE(entry.ACT_pointer)
16        entry.ACT_pointer ← ACT.INSERT(entry.count, page)
17        if newCount > HACP.count
18              HACP ← ACT.PREV(HACP)
19              entry.isHAC ← true
```

Algorithm 8 Process for updating the CCL list with information read from the ART

```
UPDATECCL()
1  CCL.CLEAR()
2  for each entry ∈ ART
3        if entry.valid and ISDRAM(entry.dest)
4              CCL.ADD(entry.page, POPCOUNT(entry.blockState))
5  CCL.SORTBYCOUNTDESC()
```

7.2.3 Completion, Demotion and Rollback

Algorithms 9, 10 and 11 show the steps for selecting pages for completion, demotion and rollback, respectively. In all three cases, the algorithm simply selects the page at the front of the list and removes it. For demotion and rollback, the algorithm must also find the corresponding entry in the PHT so that it can update the pointers to 0, indicating that the entry in no longer in the list. For completion, this last step is not necessary because there is

Algorithm 9 Steps for selecting a page for completion

```
COMPLETE()
1  entry ← CCL.POPFRONT()
2  return entry.page
```

Algorithm 10 Steps for selecting a page for demotion

```
DEMOTE()
1  entry ← DCL.POPFRONT()
2  old ← PHT.LOOKUP(entry.page)
3  old.DCL_pointer ← NIL
4  return entry.page
```

Algorithm 11 Steps for selecting a page for rollback

```
ROLLBACK()
1  entry ← RCL.POPFRONT()
2  old ← PHT.LOOKUP(entry.page)
3  old.RCL_pointer ← NIL
4  return entry.page
```

no pointer from the PHT.

For the case of completion, the page at the front of the CCL is the one with the most progress (least number of blocks remaining to finish migration). This means that the algorithm prioritizes completing migrations that are closest to finishing by themselves. This frees resources (ART entries and DRAM pages) that can then be used for migrating other pages.

For the case of demotion, the algorithm chooses from among all DRAM pages the least recently used one, which approximates the page that is less likely to be used next. This results in a likely long interval until the demoted page is needed again, which frees up time for the transfer of other pages.

For the case of rollback, the least recently used page among all pages that are currently under migration to DRAM is selected. This idea behind this selection is that the page that has not be accessed the longest since the start of migration is the most likely to not finish

migrating the remaining blocks. By rolling back its migration, an ART entry can be freed to be used by another page that migrates faster.

The idea behind demoting the least recently used page is that the algorithm chooses a page with a low probability of being used again in the near future

7.2.4 Analysis

In this subsection, I briefly discuss the complexity of updating and consulting data structures in TMP. Throughout this discussion, p is the number of pages in the system, n is the number of entries in the PACT table and m is the number of entries in the ART table. Since p is very large, it's critical that the algorithms have constant complexity with respect to p. In contrast, n and m are relatively small. Therefore, the algorithms can tolerate a higher complexity with respect to these values.

After reading monitoring information from the PACT, the DCL, RCL and ACT are updated. Updating the LRU lists (DCL and RCL) takes $O(n)$ time, since it requires moving each page's node to the front of the list. Updating the ACT takes $O(\text{n } log \text{ p})$ time. For each page in the table, a node is removed from the tree and inserted in a new location. Erasing takes constant time, since a pointer to the node is given. Inserting takes log p time, since the tree is ordered and binary search can be used. Note that finding the node in the middle of the list or the tree takes constant time with respect to p, since a hash table (PHT) is used.

7.3 EVALUATION

7.3.1 Methodology

For the experimental evaluation of TMP, I used HMMSim, the simulation infrastructure presented in Chapter 3. The main architectural parameters are the same used for the evaluation of CMMP in Chapter 6, except the size of the ART. The parameters are shown again in Table 10. Both the baseline (Multi-Queue)and TMP use CMMP, presented in Chapter 6.

TMP has several parameters, which were chosen empirically through experimentation.

Table 10: Architectural parameters

Parameter	**Value**
4GHz chip multiprocessor	4 4-issue wide, out-of-order cores, 128-entry reorder buffer
L1 I/D private cache	64KB per core, 4-way, LRU, 3 cycle hit, 16-entry queue
L2 unified shared cache	8MB, 16-way, LRU 32 cycle hit, 32-entry queue
128MB DRAM memory @ 1000MHz	64 banks, 32-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-12-12 (ns) Array energy (pJ/bit): 1.17 (reads), 0.39 (writes) Buffer energy (pJ/bit): 0.93 (reads), 1.02 (writes)
4GB PCM memory @ 400MHz	128 banks, 8-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-55-150 (ns) Array energy (pJ/bit): 2.47 (reads), 16.82 (writes) Buffer energy (pJ/bit): 0.93 (reads), 1.02 (writes)
PCM/DRAM bus	64-bit single-channel
ART	128 entries, 8-way, 0.4ns, 0.004nJ per access
PACT	8k entries, 32-way 0.6ns, 0.06nJ per access

AgingPeriod is the length of the period in either clock cycles or number of accesses used for determining whether old accesses should be discarded. *NumAgingPeriods* is the number of aging periods that an access remains part of the threshold calculation. The product of *AgingPeriod* and *NumAgingPeriods* determines the number of cycles or accesses that an access is retained for the threshold calculation. *AgingType* is the unit used for counting and can be either clock cycles or number of accesses. The default values used in the experiments are the following. *AgingPeriod*: 100 million; *NumAgingPeriods*: 10; *AgingType*: number of accesses.

For workloads, I use a subset of SPEC CPU2006 benchmarks run in rate mode as a multi-programmed workload. I treat each input of the same benchmark as a separate workload (in the figures, a number after the benchmark name is the input for the benchmark). Each workload is run for 1 billion instructions per core after a warm-up phase of 5 billion instructions. Each experiment consists of 4 copies of the same workload run in parallel on

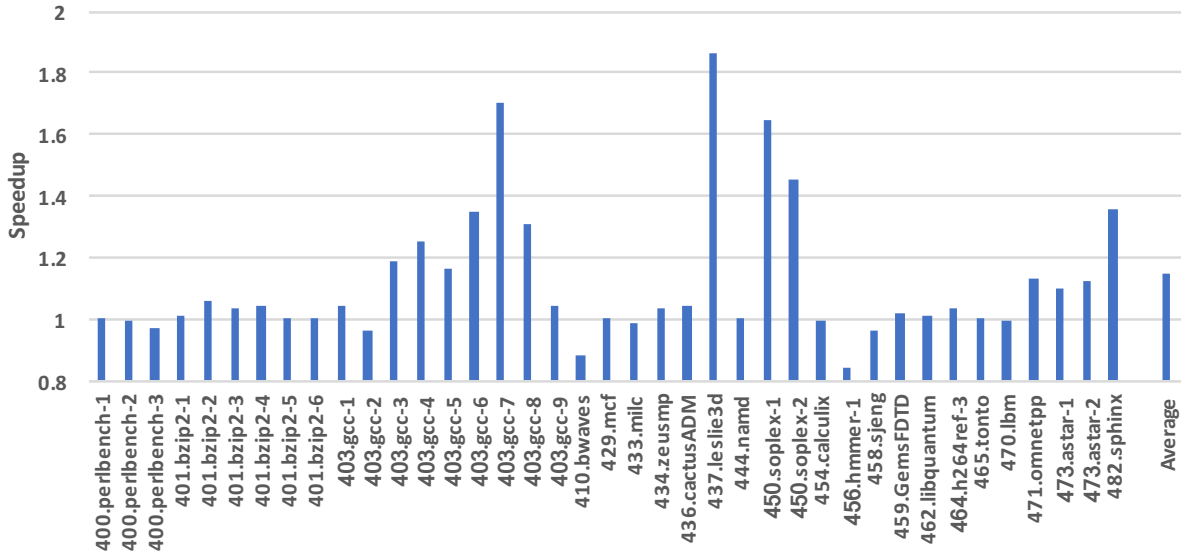

Figure 26: Speedup of Multi-Queue and TMP, normalized to No Migration.

the simulated 4-core system. To fully stress migration, I consider only multi-programmed workloads with working set sizes greater than the available DRAM memory (128MB). I also consider single-programmed workloads, where a single copy of a benchmarks is run on a single core, and the available DRAM is set to one quarter of available DRAM (32MB).

For the experiments, a baseline system running CMMP with the Multi-Queue migration policy is assumed.

7.3.2 Performance

Figure 26 shows the speedup of TMP relative to Multi-Queue for all benchmarks. On average, TMP performs 15% better than the baseline, with four benchmarks having a speedup of more than 40%. There are two main reasons for TMP's improved behavior over to Multi-Queue. First, when using CMMP, Multi-Queue does not have a mechanism for completing migrations. As a result, some migrations spend a long time in the ART, which prevents other migrations from taking place. In contrast, TMP either completes or rollbacks slow

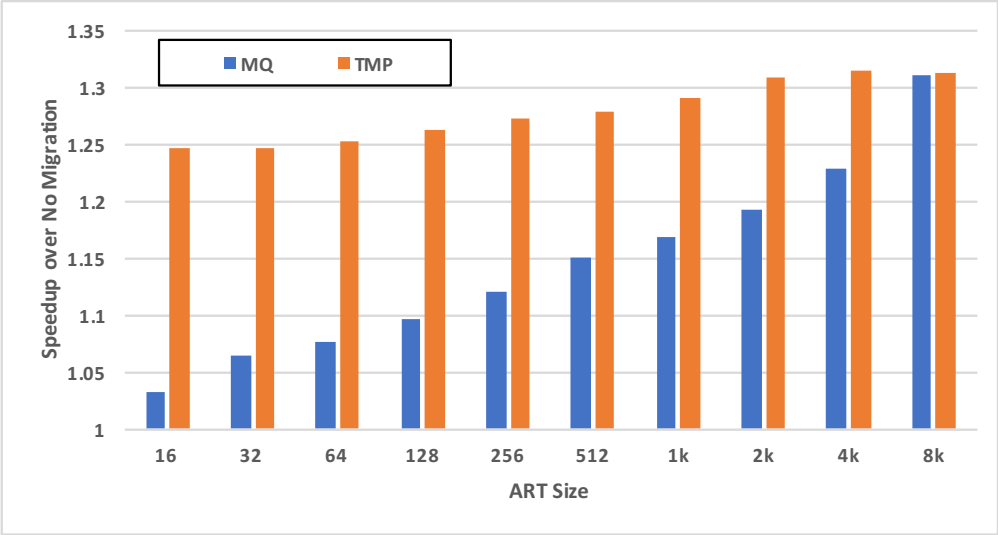

Figure 27: Speedup of TMP for different ART sizes, normalized to the baseline.

migrations. Although Multi-Queue does have mechanism for rolling back migrations (a page under migration can be selected for eviction from DRAM, resulting in a rollback), TMP is better at selecting pages to free space in the ART because it has two available options (complete and rollback).

The second reason is related to how the policies determine whether a page belongs in DRAM vs PCM. In Multi-Queue, a fixed threshold is used (pages above queue 5 are in DRAM, and 32 accesses are needed to get to queue 5). In TMP, the threshold moves depending on the number of accesses that the memory has seen, allowing the policy to determine the most heavily used pages. Coupled with an aging mechanism, the policy can adapt better to the access patterns of applications, resulting in better performance.

Figure 27 shows the average speedup of Multi-Queue and TMP relative to No Migration for different sizes of the ART. The performance of TMP relative to No Migration is high for all ART sizes (at least 25% speedup). This is not the case for Multi-Queue, which has low speedups when the ART is small. With large ART sizes, both Multi-Queue and TMP perform well. This behavior is consistent with the explanation given in the previous paragraph: when Multi-Queue has enough space in the ART for other migrations, it performs

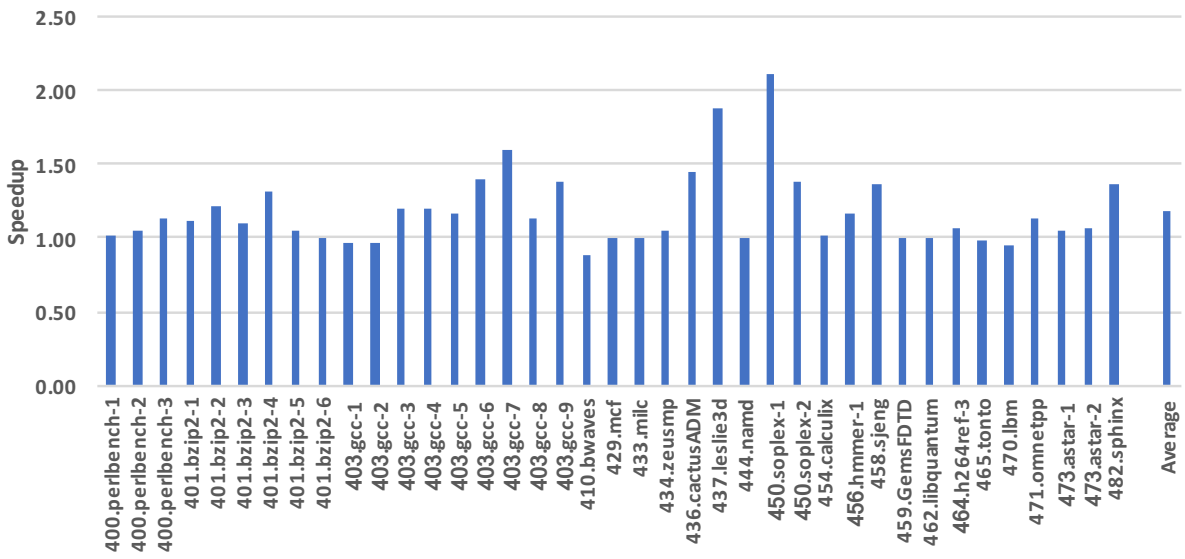

Figure 28: Speedup of TMP for multi-core workloads, normalized to the baseline

well. TMP with a 16-entry ART can achieve the same performance of Multi-Queue with an ART of 4k entries.

Figure 28 shows the speedup of TMP for multi-programmed workloads. The average speedup is 18% across all benchmarks, with some benchmarks having a speedup of more than 50%. The performance of individual benchmarks closely follows the results of the single-programmed case: in most cases, the multi-core speedup is within 3% of the speedup seen in the single-core case.

Figure 29 shows the average multi-programmed speedup for different sizes of the ART for Multi-Queue and TMP. TMP again performs well relative to No Migration (more than 25% speedup), especially for large ART sizes (more than 40% for ART sizes of 4k and greater). Multi-Queue also does well with large ART sizes, but struggles with small ones. In general, both Multi-Queue and TMP benefit from large ART sizes more with multi-programmed workloads than with single-programmed workloads. This is because there are more opportunities to select pages for migration than can benefit performance, and a large

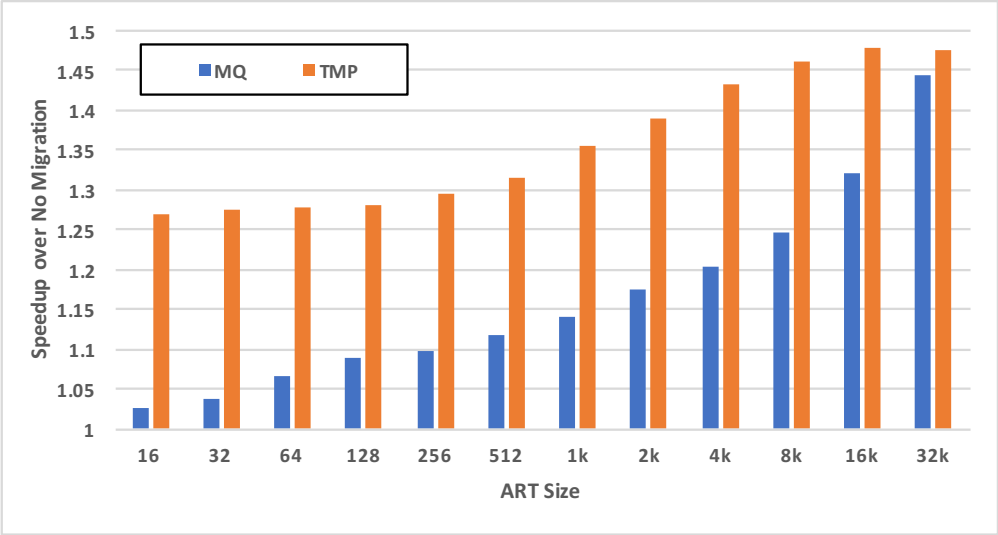

Figure 29: Speedup of Multi-Queue and TMP for different ART sizes for multi-core workloads, normalized to No Migration.

ART size takes advantage of this. However, with small ART sizes, Multi-Queue performs poorly relative to TMP. This is because when memory bandwidth is almost fully utilized and the ART size is small, TMP can select a better page to migrate.

7.4 COMPARISON TO IDEAL SYSTEM

Figure 1 in Chapter 1 showed a comparison of the state-of-the-art and ideal systems for several migration rates. In this last section, I revisit these results and compare them against the new mechanisms presented in this thesis. Figure 30 shows the speedup relative to No Migration of 6 systems: the state-of-the-art from Figure 1 (Multi-Queue without CMMP), CMMP with Multi-Queue and ART sizes of 128 and 2k entries, CMMP with TMP and ART sizes of 128 and 2k entries, and the ideal system from Figure 1 (Zero-Interference migration with Offline migration policy). This comparison does not show the speedups for migration rates lower than 100%, as did the previous figure. The reason is that limiting

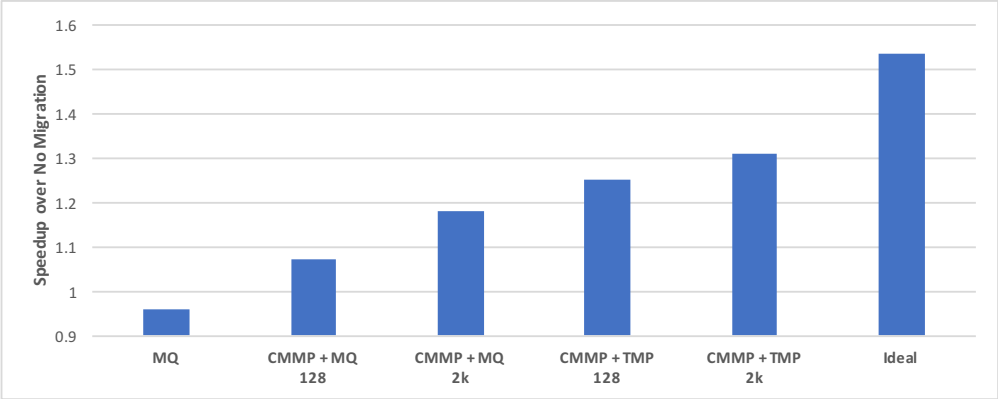

Figure 30: Comparison of CMMP with Multi-Queue and TMP and two ART sizes against state-of-the-art (Multi-Queue without CMMP) and ideal systems.

the migration rate is not compatible with OBM, since blocks are migrated on demand as a result of application requests and the rate is set by how fast the application accesses blocks. Limiting the rate at which migrations are started does not slow down migrations, because a migration will simply be triggered later and adopt the migration rate that is natural to the application.

CMMP alone (without a better migration policy) can improve performance significantly (between 7% and 18% depending on the ART size) relative to No Migration (the state-of-the-art does worse than No Migration at 100% migration rate because of high interference in the memory system). CMMP achieves this by using memory bandwidth more efficiently: it allows policies to identify good migration candidates before paying the cost of a full migration, and it coalesces application accesses to memory with migration traffic.

CMMP with a better migration policy (TMP is designed specifically for CMMP) can improve performance even more (between 25% and 31% depending on the ART size) relative to No Migration. TMP's algorithm for selecting pages to migrate adapts better to application behavior. In particular, CMMP has a *moving* threshold to identify hot and cold pages that changes as pages are accessed (Multi-Queue's threshold is fixed at 32 accesses). Compared to Multi-Queue, TMP makes better use of ART entries, a limited resource in CMMP. TMP

with 128 entries performs better than Multi-Queue with 16 times as many entries (2k). TMP achieves this by taking advantage of CMMP's support for *completion* and *rollback* of ongoing migrations, which frees resources for other migrations (Multi-Queue does not do this).

The best non-ideal system (CMMP with TMP and a 2k-entry ART) can achieve 58% of the potential improvement of an ideal system. CMMP cannot match the performance of a memory system with no interference due to migration. In addition, a real policy like TMP cannot anticipate changes in application behavior as the Offline policy does. For these two reasons, CMMP and TMP cannot achieve all of the potential improvement of the ideal system. However, it does come close.

7.5 SUMMARY

This chapter introduced TMP, a new migration policy specifically designed to take advantage of hardware mechanisms presented in Chapter 6. TMP keeps track of the most frequently used pages, and easily adapts to changes in application behavior. TMP relies on CMMP's PACT table to keep an accurate count of page accesses and uses it to update internal OS data structures with low complexity. TMP also uses progress information about migrations from CMMP's ART table to decide whether to rollback or complete a page's transfer, which can free resources. An evaluation of the proposed policy shows that TMP performs 15% better than Multi-Queue for single-programmed workloads and 18% for multi-programmed workloads.

8.0 CONCLUSIONS

This thesis proposes hardware support for enabling page migration in software-managed hybrid main memory systems. First, I built a simulation framework for studying and evaluating hybrid main memory. This framework allows users to test new hardware mechanisms and migration policies for managing hybrid memory. In addition, the framework provides various tools for characterizing the behavior of programs running on hybrid memory systems. Based on the results of characterization studies using this framework, I proposed mechanisms several mechanisms for enabling efficient page migration in hybrid main memory. The following conclusions can be drawn from the qualitative analysis and the experimental results.

- Current commodity hardware for main memory does not allow for efficient migration of pages by the OS. As a result, applications must be paused during page migration, considerably hurting performance. I propose concurrent page migration (CPM) to pin the contents of pages under migration in the last-level cache, which avoids the need to stall an application.
- Although support for page migration improves performance, allowing only one concurrent migration is detrimental to performance because the memory system can not react quickly enough to an application's changing behavior. However, multiple concurrent migrations can saturate the memory with migration requests, which can also impact performance. Based on these observations, I performed a characterization study of the overhead of migration of multiple pages and identified the factors that limit performance in hybrid main memory.
- Based on the results of this study, I propose a novel mechanism for efficient concurrent migration of multiple pages (CMMP) in software-managed hybrid memory. This mecha-

nism reduces contention at the memory system caused by migration and provides a simple and elegant interface for communicating access counts to software. CMMP reduces migration bandwidth by copying blocks on demand as they are accessed by applications and by eliding the transfer of untouched blocks during migrations to PCM.

- I also propose a new migration policy specifically tailored for CMMP. The new policy relies on access counts collected by CMMP to continually determine the best set of pages to place in DRAM. The new policy also relies on CMMP's ability to complete and roll back migrations to free up resources for use by other pages. The new policy outperforms the current state-of-the-art migration policy by a large margin.

BIBLIOGRAPHY

[1] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 37–48, ACM, 2012.

[2] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 237–248, ACM, 2013.

[3] "Process integration, devices and structures," in *Int'l. Technology Roadmap for Semiconductors*, 2012.

[4] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, (New York, NY, USA), pp. 175–186, ACM, 2010.

[5] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: a hybrid PRAM and DRAM main memory system," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pp. 664–669, 2009.

[6] A. P. Ferreira, B. Childers, R. Melhem, D. Moss and, and M. Yousif, "Using PCM in next-generation embedded space applications," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pp. 153–162, April 2010.

[7] T. Liu, Y. Zhao, C. J. Xue, and M. Li, "Power-aware variable partitioning for DSPs with hybrid PRAM and DRAM main memory," in *Proceedings of the 48th Annual Design Automation Conference*, DAC '11, pp. 405–410, 2011.

[8] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 24–33, ACM, 2009.

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

[10] W. Zhang and T. 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, (Washington, DC, USA), pp. 101–112, IEEE Computer Society, 2009.

[11] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 126–136, Feb 2015.

[12] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+DRAM hybrid main memory," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pp. 14–14, 2009.

[13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[15] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 105–118, ACM, 2011.

[16] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Concurrent page migration for mobile systems with OS-managed hybrid memory," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '14, 2014.

[17] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Characterizing the overhead of software-managed hybrid main memory," in *Proceedings of the IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '15, 2015.

[18] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Hmmsim: A simulator for hardware-software co-design of hybrid main memory," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2015 IEEE*, NVMSA '15, 2015.

[19] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Concurrent migration of multiple pages in software-managed hybrid main memory," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, ICCD, 2016.

[20] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[21] G. Burr, M. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, B. Rajendran, S. Raoux, and R. S. Shenoy, "Phase change memory technology," *Journal of Vacuum Science Technology B: Microelectronics and Nanometer Structures*, vol. 28, no. 2, pp. 223–262, 2010.

[22] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 2–13, ACM, 2009.

[23] S. Bock, B. Childers, R. Melhem, D. Mosse, and Y. Zhang, "Analyzing the impact of useless write-backs on the endurance and energy consumption of pcm main memory," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pp. 56–65, April 2011.

[24] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," in *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on*, pp. 3014–3017, 2007.

[25] S. Cho and H. Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 347–357, Dec 2009.

[26] M. Qureshi, M. Franceschini, and L. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–11, Jan 2010.

[27] T. Nirschl, J. Philipp, T. Happ, G. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. Chen, Y. Zhu, R. Bergmann, H. L. Lung, and C. Lam, "Write strategies for 2 and 4-bit multi-level phase-change memory," in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pp. 461–464, 2007.

[28] S. Ahn, Y. Song, C. Jeong, J. Shin, Y. Fai, Y. Hwang, S. Lee, K. Ryoo, S. Lee, J.-H. Park, H. Horii, Y. Ha, J. Yi, B. Kuh, G. Koh, G. Jeong, H. Jeong, K. Kim, and B.-I. Ryu, "Highly manufacturable high density phase change memory of 64mb and beyond," in *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*, pp. 907–910, 2004.

[29] F. Bedeschi, R. Fackenthal, C. Resta, E. Donze, M. Jagasivamani, E. Buda F. Pellizzer, D. Chow, A. Cabrini, G. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande, "A multi-level-cell bipolar-selected phase-change memory," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 428–625, 2008.

[30] K.-J. Lee, B.-H. Cho, W.-Y. Cho, S. Kang, B.-G. Choi, H.-R. Oh, C.-S. Lee, H.-J. Kim, J. min Park, Q. Wang, M.-H. Park, Y.-H. Ro, J.-Y. Choi, K.-S. Kim, Y.-R. Kim, I.-C. Shin, K. won Lim, H.-K. Cho, C.-H. Choi, W. ryul Chung, D.-E. Kim, K.-S. Yu, G.-T. Jeong, H.-S. Jeong, C.-K. Kwak, C.-H. Kim, and K. Kim, "A 90nm 1.8v 512mb diode-switch pram with 266mb/s read throughput," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pp. 472–616, Feb 2007.

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

[32] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, (3001 Leuven, Belgium, Belgium), pp. 914–919, European Design and Automation Association, 2010.

[33] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. 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, (New York, NY, USA), pp. 14–23, ACM, 2009.

[34] J. Dong, L. Zhang, Y. Han, Y. Wang, and X. Li, "Wear rate leveling: Lifetime enhancement of pram with endurance variation," in *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 972–977, 2011.

[35] A. Ferreira, S. Bock, B. Childers, R. Melhem, and D. Moss, "Impact of process variation on endurance algorithms for wear-prone memories," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pp. 1–6, March 2011.

[36] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 454–464, ACM, 2011.

[37] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 235–246, IEEE Computer Society, 2012.

[38] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 404–415, ACM, 2013.

[39] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, (Berkeley, CA, USA), pp. 187–200, USENIX Association, 2004.

[40] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS IX, (New York, NY, USA), pp. 105–116, ACM, 2000.

[41] M. M. Tikir and J. K. Hollingsworth, "Using hardware counters to automatically improve memory performance," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 46–, IEEE Computer Society, 2004.

[42] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," *J. Parallel Distrib. Comput.*, vol. 68, pp. 1186–1200, Sept. 2008.

[43] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[44] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, pp. 16 –19, jan.-june 2011.

[45] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module a simulation infrastructure for the jwac memory scheduling championship," 2012.

[46] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: Architectural simulator to model (non-)volatile memory systems," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2015.

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[48] NVidia, "Variable smp (4-plus-1^{TM}) - a multi-core cpu architecture for low power and high performance," white paper, NVidia, 2011.

[49] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pp. 81–90, Nov 2011.

[50] K. Hoste and L. Eeckhout, "Characterizing the unique and diverse behaviors in existing and emerging general-purpose and domain-specific benchmark suites," in *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pp. 157–168, April 2008.