SHARED RESOURCE MANAGEMENT FOR NON-VOLATILE ASYMMETRIC MEMORY

by

Miao Zhou

B.S., Huazhong University of Science and Technology, 2004

M.S., Huazhong University of Science and Technology, 2007

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Miao Zhou

It was defended on

April 30, 2015

and approved by

Rami Melhem, PhD, Computer Science Department

Bruce R. Childers, PhD, Computer Science Department

Daniel Mossé, PhD, Computer Science Department

Yiran Chen, PhD, Electrical and Computer Engineering Department

Dissertation Director: Rami Melhem, PhD, Computer Science Department

SHARED RESOURCE MANAGEMENT FOR NON-VOLATILE ASYMMETRIC MEMORY

Miao Zhou, PhD

University of Pittsburgh, 2015

Non-volatile memory (NVM), such as Phase-Change Memory (PCM), is a promising energy-efficient candidate to replace DRAM. It is desirable because of its non-volatility, good scalability and low idle power. NVM, nevertheless, faces important challenges. The main problems are: writes are much slower and more power hungry than reads and write bandwidth is much lower than read bandwidth. Hybrid main memory architecture, which consists of a large NVM and a small DRAM, may become a solution for architecting NVM as main memory. Adding an extra layer of cache mitigates the drawbacks of NVM writes. However, writebacks from the last-level cache (LLC) might still (a) overwhelm the limited NVM write bandwidth and stall the application, (b) shorten lifetime and (c) increase energy consumption.

Effectively utilizing shared resources, such as the last-level cache and the memory bandwidth, is crucial to achieving high performance for multi-core systems. No existing cache and bandwidth allocation scheme exploits the read/write asymmetry property, which is fundamental in NVM. This thesis tries to consider the asymmetry property in partitioning the cache and memory bandwidth for NVM systems.

The thesis proposes three writeback-aware schemes to manage the resources in NVM systems. First, a runtime mechanism, *Writeback-aware Cache Partitioning* (WCP), is proposed to partition the shared LLC among multiple applications. Unlike past partitioning schemes, WCP considers the reduction in cache misses as well as writebacks. Second, a new runtime mechanism, *Writeback-aware Bandwidth Partitioning* (WBP), partitions NVM service cycles among applications. WBP uses a *bandwidth partitioning weight* to reflect the importance of writebacks (in addition to LLC misses) to bandwidth allocation. A companion *Dynamic Weight Adjustment* scheme dynamically selects the cache partitioning weight to maximize system performance. Third, *Unified Writeback-aware Partitioning* (UWP) partitions the last-level cache and the memory bandwidth cooperatively. UWP can further improve the system performance by considering the interaction of cache partitioning and bandwidth partitioning. The three proposed schemes improve system performance by considering the unique read/write asymmetry property of NVM.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ALGORITHMS

1.0 INTRODUCTION

Designing large capacity and energy-efficient memory systems becomes an increasingly important issue nowadays due to the high demands of modern data-intensive applications, the adoption of multicore processors and virtualization. Improving the energy efficiency of computing systems is a vital part towards a green society as the power consumed by computing systems is substantial. The U.S. Environmental Protection Agency estimated that the annual energy cost for data centers in 2011 was $7.4 billion. The energy/power demand of modern processors has been lowered through numerous architectural and design techniques, such as dynamic voltage and frequency scaling (DVFS) [39]. With energy efficient processors and the increasing size of the memory systems, main memory becomes a significant system power consumer [44, 1].

In past decades, DRAM has been the best candidate for main memory due to its low cost per bit. However, the memory capacity of data centers and servers has increased tremendously due to the large footprints of modern applications and the advent of multi-core systems. In these systems, DRAM static power accounts for a substantial portion of energy. As a result, the memory subsystem has become a major consumer of power, which has fueled research in alternative memory technologies. Non-volatile memory (NVM) is proposed as a replacement for DRAM or in addition to DRAM (using a smaller DRAM as a cache for the larger NVM). NVM is a desirable technology because of its non-volatility, low power for reads and very low idle power. Some of the NVM technologies include NAND Flash, Change Memory (PCM) [33, 15] and Spin-Transfer Torque RAM (STT-RAM) [50, 78, 20, 8, 36, 10].

NAND Flash is not suitable for serving as the main memory. It has very limited number of write/erase cycles: 10^5 rewrites [19] as opposed to 10^{16} for DRAM. NAND Flash also requires a block to be erased before writing into that block, which introduces considerably extra delay and energy. Moreover, NAND Flash can only be accessed in blocks and is not byte-addressable.

Attribute	DRAM	NAND Flash	PCM	STT-RAM
Non-Volatile	No	Yes	Yes	Yes
Byte-addressable	Yes	No	Yes	Yes
Read Latency	20-50 ns	25 μs	50 ns	2 ns
Write Latency	20-50 ns	500 μs	1 μs	10 ns
Endurance	10^{16}	10^4-10^5	10^7-10^8	10^{12}
Cell Area	6 F^2	5 F^2	5-8 F^2	37-40 F^2

Table 1: Basic attributes of different memory technologies

Therefore, NAND flash has been proposed as a disk cache [4, 35] or a replacement for disks [86] where writes are relatively infrequent, and happen mostly in blocks.

The other two NVM technologies, PCM and STT-RAM, are promising candidates for energy-efficient memory technology. Both of them share the common advantages of non-volatility, byte-addressability and scalability [24, 32]. STT-RAM is faster than PCM, and has nearly the same endurance as DRAM [79]. PCM, on the other hand, is much denser than STT-RAM. The cell area for DRAM, PCM and STT-RAM are $6F^2$ [81], 5-$8F^2$ [41], and 37-$40F^2$ [12] respectively, where F is the feature size. PCM also has excellent scalability within current CMOS fabrication methodology [71, 68, 41, 60, 70]. PCM's attributes, along with the common advantages of emerging memory technologies, make it a promising candidate of main memory [6, 42, 64].

Table 1 compares the basic attributes of NAND Flash, PCM and STT-RAM against DRAM [12, 22, 9, 41, 45, 79, 81, 96]. Unlike DRAM, NVM has a unique read/write asymmetry property (in terms of access latency, energy consumption and endurance).

In this dissertation, I explore using of PCM as main memory. Although PCM has many promising properties, it faces important challenges. It is wearable such that only 10^7 to 10^8 writes can be performed. A PCM device would fail in days or even hours without wear-leveling [96, 65, 7, 18]. Moreover, PCM is slower and consumes more energy on writes than DRAM. *Hybrid* main memory architectures [42, 17, 64] have been proposed to alleviate the problem. The hybrid architecture introduces a small DRAM as the *last-level cache* (LLC) before a large PCM, and is the baseline memory architecture in this thesis.

Last-level cache and memory bandwidth are two important resources impacting system performance. In a single-core system, cache reduces the average access latency and off-chip memory traffic, while the available memory bandwidth determines the processing rate of off-chip accesses. In a multi-core system, cores compete for these resources, drastically increasing their performance impact. Qureshi and Patt showed that cache partitioning improves the cache utility and system performance. They proposed to partition the LLC to minimize the total number of LLC misses to achieve better system performance, based on the observation that a reduction in LLC miss rate correlates with an improvement in IPC [63].

Bandwidth partitioning has also been shown to be beneficial for system performance. Liu et al. [47] proposed an analytic model to understand how off-chip bandwidth partitioning affects system performance. By allocating bandwidth according to the analytic model, they were able to improve the weighted speedup by up to 20% in a DRAM memory system.

The existing cache partitioning and bandwidth partitioning techniques [63, 47] do not consider the unique read/write asymmetry property of NVM. In this dissertation, we propose cache partitioning and bandwidth partitioning techniques that are tailored for NVM main memory systems by considering the read/write asymmetry property of NVM.

The rest of this dissertation is organized as follows: Chapters 1.1, 1.2 and 1.3 describe the problem we are solving, present our solution and contributions, respectively. Chapter 2 presents background information and related work. The proposed techniques are presented in Chapter 3 through Chapter 5. Chapter 6 concludes the thesis.

1.1 PROBLEM STATEMENT

Computing systems are part of our lives today, not only in a recognizable form but ranging from the large clusters in data centers to embedded systems in handy devices. Building energy-efficient and large capacity memory systems is very important. NVM technologies are promising candidates for energy-efficient memory. However, typical NVMs face certain challenges due to undesirable write properties. It is our goal to propose techniques to better utilize the shared resources (last-level cache capacity, and memory bandwidth) to mitigate the undesirable impacts of memory writes by exploiting the unique read/write asymmetric property of NVM. This work is important for enabling energy-efficient NVM as the main memory.

1.2 SOLUTION OVERVIEW

In this dissertation, we propose three techniques to improve the shared resource utility and mitigate the negative effects of non-volatile memory's write operations. First, *writeback-aware cache partitioning* (WCP) is proposed to divide the last-level cache among competing applications such that the number of misses and writebacks are reduced. A companion scheme, *write buffer balancing* (WBB) replacement, spreads writeback traffic among memory banks to reduce the delay due to congested memory write requests. Second, *writeback-aware bandwidth partitioning* (WBP) splits memory device service cycles among competing applications to optimize the overall system performance. Finally, we study the interaction between cache partitioning and bandwidth partitioning. *Unified Writeback-aware Partitioning* (UWP) allocates the cache and memory bandwidth cooperatively to further improve the shared resource utility.

1.3 CONTRIBUTIONS

This dissertation makes the following contributions:

- It introduces the concept of writeback-aware partitioning which is suitable for shared resource management in NVM systems.
- It proposes *Writeback-aware Cache Partitioning* (WCP) to partition the LLC of NVM. WCP takes into account writeback information, and outperforms a seminal technique, UCP, in terms of system throughput, fairness, memory lifetime and energy efficiency. It also proposes WBB to manage the cache partition of each application to ensure that the writeback traffic to write buffers is balanced.

- It proposes *Writeback-aware Bandwidth Partitioning* (WBP) to account for writeback information to partition the memory bandwidth. A companion scheme, *Dynamic Weight Adjustment* (DWA), is proposed to find the best bandwidth partitioning weight at runtime based on workload behavior.
- It demonstrates the benefits of managing the last-level cache and memory bandwidth cooperatively. An analytic model is introduced to quantify the performance impact of cache partitioning coordinated with bandwidth partitioning. Based on the analytic model, *Unified Writeback-aware Partitioning* (UWP) is proposed to partition cache and memory bandwidth cooperatively.

In conclusion, this thesis finds that LLC writeback information is important in managing the shared resources (e.g., LLC capacity and memory bandwidth) for systems with NVM due to the unique asymmetric property. By intelligently considering the LLC writeback information, we can improve the utility of the shared resources and achieve better system performance. The thesis also finds that coordinated management of the LLC capacity and memory bandwidth can further improve the overall shared resource utility and system performance. Although the thesis specifically considers PCM, most of the results can equally apply to other NVM technologies such as STT-RAM.

2.0 BACKGROUND AND RELATED WORK

2.1 PHASE CHANGE MEMORY

2.1.1 Phase Change Memory Background

Phase Change Memory (PCM) is a non-volatile memory that exploits the unique behavior of phase change material to store information. Although the technology emerged recently, the theory of phase change material has its origins early in 1960s when Ovshinsky reported a reversible change in resistivity upon a change in phase in certain glasses [58]. The first Phase Change Memory device was announced in Electronics by Neale et al. [66]. In the following years, the advance of semiconductor manufacturing technology enabled the development and application of PCM. Phase change material is already widely used in rewritable CDs and DVDs, in which the same alloy is used as the PCM memory developed by Numonyx. By exploiting the electrical resistivity of phase change material, PCM is drawing increasing interest recently, as it can be used as a memory cell and organized into memory array similar to DRAM.

A conventional DRAM cell uses a capacitor to store a bit of information. Analogously, a PCM cell uses phase change material to remember a bit. The phase change material is one type of chalcogenide alloy, such as Ge2Sb2Te5 (GST), which has two stable physical states: amorphous and crystalline. In the amorphous state, the material is highly disordered and exhibits high resistivity. In the crystalline state, the material has a regular crystalline structure and exhibits low resistivity. PCM exploits the difference in resistivity between these two states of the material to

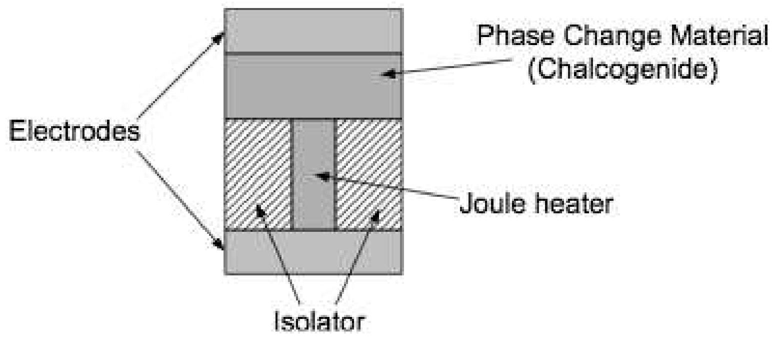


Figure 1: PCM cell structure

store data. Typically, a cell in the amorphous state (high resistance) is regarded as a logic “0” (RESET state), and a cell in crystalline state (low resistance) is regarded as a logic “1” (SET state). Unlike DRAM that relies on constant refresh to retain its data, the state of GST is preserved even after the cell is powered off, meaning that PCM is non-volatile.

A PCM cell is composed of several components (Figure 1). A layer of chalcogenide (GST) is sandwiched between two electrodes. A joule heater is placed between GST and the bottom electrode. The structure forms a PCM cell, which appears as a resistance in the circuit. Reading data from a PCM cell involves sensing the resistance level of the cell. This is done by applying a small voltage across the two electrodes so that the resistance of GST can be measured. This process is non-destructive and has negligible heat stress compared to write operations.

PCM write operations are done by changing the physical state of the GST material. The writing process requires controlling the temperature of the material to achieve the desired state. When heated above its crystallization temperature ($-300°$C) but below its melting temperature ($-600°$C) over a period of time, GST turns into a low-resistance crystalline state (which corresponds to logic “1” or SET state). When heated above its melting point and quenched quickly, GST turns into a high-resistance state (which corresponds to logic “0” or RESET state). Figure 2 illustrates the two process of PCM write operations. Figure 3 shows the example current-voltage behavior of PCM cells. Though writing a PCM cell incurs high operating temperature, the thermal cross-

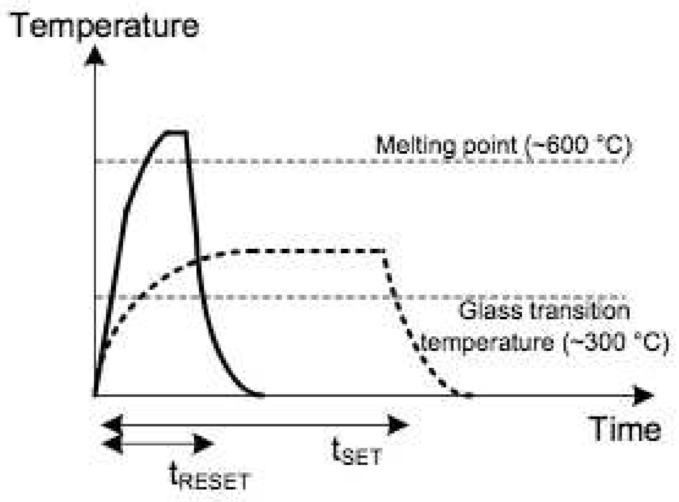


Figure 2: The process of PCM write operations

talk between adjacent cells at 65nm is shown to be negligible even without thermal insulation material [60]. Similar to the multi-level flash memory, the phase change material can also be programmed into four or more distinct states, forming a multi-level PCM cell that can represent four or more values [60, 9].

Despite the differences with DRAM cells, PCM can still use a similar array structure as DRAM arrays. It can use the same peripheral logic such as decoders, row buffers, request/reply networks etc. as the DRAM array [43]. In a typical PCM cell array structure, each PCM cell is connected between an access transistor and a bitline (BL). The access transistor is controlled by the wordline (WL). In order to access a PCM cell, its wordline is selected to enable the access transistor, which forms a path between the cell's bitline and ground. Read or write operations on the PCM cell are then done by applying different voltage pulses on the bitline. Selecting a wordline enables the access transistor of a series ("row") of PCM cells. By controlling the bitlines using a column mux, read or write operations can be performed on selected cells in that row.

There are two main options for the access of the device in a PCM cell: a transistor or a diode. A diode has a simpler structure, and hence is good for cell density. However, a diode cannot satisfy the high write current requirement beyond sub-100nm technology [84]. Also the scaling rule for a

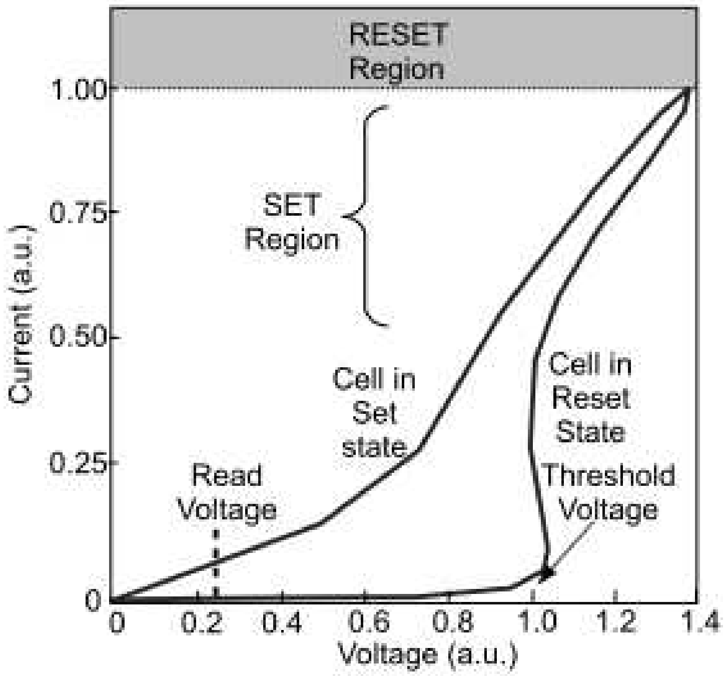


Figure 3: Example current-voltage behavior of PCM cells

diode is not as clear as in NMOS [60]. Lastly, the diode-based cell has been reported to be more vulnerable to errors induced by writing data to adjacent cells because of bipolar turn-on of the nearest-neighbor cells [29]. Taking all of the above, especially the scalability, into consideration, I assume transistor-based PCM cells in my experiments. My assumption was also confirmed by Li et al. [26].

PCM's advantages. Like DRAM, PCM memory is byte-addressable, giving it a big advantage over current NAND Flash technology that only supports block accesses. PCM offers comparable read latency as DRAM. Typical area size of PCM cell is 5-8F^2 [41], meaning that PCM has similar density to DRAM.

With shrinking feature size, DRAM is facing serious scaling problems as it is bounded by the limitation in cell-bitline capacitance ratio. DRAM has been found to be difficult to scale below 40nm [82]. PCM, on the other hand, offers much better scalability: When a PCM cell shrinks, the volume of the GST material shrinks as well, resulting in less write current. Hence PCM provides a truly scalable solution compared to conventional DRAM. A recent prototype by Liang et al. demonstrated the viability of the PCM cell reaching 2.5nm technology node [26].

Like NAND Flash, PCM is non-volatile. In theory, a PCM cell only consumes energy when it is accessed (read or write). This makes it possible to build memory chips with low leakage, which is crucial to meet the low-power requirements of future memory systems. Moreover, PCM uses physical states instead of electrical charge to represent data, making it much less vulnerable to the soft errors caused by alpha particles or cosmic radiation.

PCM's disadvantages. Due to repeated heat stress applied to the phase change material, PCM has limited number of write cycles (i.e., write endurance). A single cell can typically sustain 10^7-10^9 writes before a failure occurs [19, 9, 16]. While this is much better than NAND Flash, it could be a big concern if PCM is used in main memory. A PCM device without any lifetime improvement technique may last only about 100 days running a typical SPEC CPU program.

PCM's write operation is also slower than its read operation due to the heating/cooling process during a write. What makes things worse is that PCM write incurs high current injection (e.g., 0.6-1mA), resulting in high write power (2.88-4.8mW per bit) [43]. When PCM is used as memory, a multi-bit memory write (e.g., writing a 512-bit memory line) is usually completed in several rounds, with each round writing part of the line. This makes PCM's per-access write latency even longer. For example, Numonyx reported a 1us page write latency for its PCM prototype. This long per-access write latency and high write power have created some major issues when using PCM in main memory.

Another issue is the resistance drift in multi-level cells (MLC) [88, 23]. After a multi-level PCM cell is programmed, its resistance may increase and saturate over time, which is termed "resistance drift" [90]. Previous studies have indicated that the phenomena is caused by the structural relaxation of chalcogenide material, which is a thermally-activated, atomic rearrangement of the amorphous structure [88]. Resistance drift can cause a PCM cell to change to another state at run-time, causing a soft error. A study by Awasthi et al. has shown that the drift may occur as soon as 1.81 seconds after a cell is programmed [52]. In response to this problem, architectural techniques have been proposed to mitigate the issue without high overhead [52, 90]. In this thesis, I assume a single-level cell PCM which does not have the resistance drift issue. However, the techniques proposed in this thesis are independent of PCM cell technology. They can be extended to multi-level cell (MLC) and are orthogonal to the drift-mitigating schemes.

While PCM is favored due to its non-volatility, high scalability and low power consumption, it suffers from the undesirable properties such as its slow, power hungry and destructive write operation. Next, we will described the existing architectural techniques to mitigate the endurance and performance problem of PCM.

2.1.2 Architectural techniques for PCM

PCM has an endurance of 10^7 to 10^9 writes per cell [19, 9, 16]. If a specific cell is written once a second, it will take only 115 days for that cell to pass 10^7 writes. This shows that, even with a very low write rate per cell of one write per second, PCM endurance is not enough to sustain a lifetime of 7 to 8 years. which is the expected lifetime of a server assuming obsolescence and capacity limits.

A number of techniques have been proposed to increase lifetime of PCM devices and PCM memory systems. These techniques vary from write minimization (avoiding unnecessary writes), to various forms of wear-leveling, to changes in bit encoding. Some important techniques are presented in the following subsections.

The goal of the techniques presented below is to remove writes that are redundant or modify the value to be written to reduce the number of bits to be altered. This can be done at the cell, row or even at higher levels of aggregation.

Read-before-write. A read is executed before the write and only the cells that store values different from the values to be written are updated, and the cells that already have the final value are not modified. Many PCM devices [33, 37, 43] implement this scheme at the cell level.

Partial writes. In this technique [42], higher level caches (the one closest to the main memory) track dirty word or line and pass that information to the main memory. This additional information allows the PCM memory controller and devices to ignore the unmodified portions of the cache. In [64], a similar technique, called Line-Level Writes, is proposed that stores dirty information per cache line.

Flip-N-Write. Flip-N-Write uses an additional bit in each row to determine if the value is stored in a normal encoding or inverted [7]. The decision of which way to store is taken based on the lowest number of changed bits between the already stored data and the new data. This technique reduces the number of bits to write, thus increasing write bandwidth to the memory. PCM write performance is power limited so more bits can be written simultaneously within the same power budget. Less energy is consumed per transaction increasing energy-efficiency of the memory.

In a system in which all the cells have the same endurance, an uneven distribution of wear can cause cells that have more wear to fail prematurely, causing the whole memory to fail even though most of the cells are still healthy. Wear-leveling has the objective of distributing the wear evenly over the cells, avoiding premature cell failures and consequently increasing the memory lifetime. The memory access pattern of applications does not normally follow a regular pattern of an even distribution of writes over the whole memory, but it is highly skewed (see Chapter 3) directing a higher number of writes to a small number of memory locations. The memory subsystem have to counteract that access pattern to avoid premature failures and achieve a higher lifetime. Wear-leveling is applied to achieve this goal and can be implemented using different algorithms and granularities. A common implementation disconnects the physical address from a logical address using a mapping function that translates a logical address to the corresponding physical address. This mapping function allows the translation to be changed, pointing a logical address to different physical locations, throughout the lifetime of the memory effectively distributing the writes to the logical address over a larger number of physical addresses. Some of the wear-leveling algorithms use a configurable mapping function, while others implement a mapping table. Some of the algorithms require the physical memory to be larger than what is addressable by a logical address (excess capacity), while others do not require but support this implementation. The wear-leveling algorithms proposed to be used in PCM memories are described below.

Row Shifting. This technique implements wear-leveling at the row level [96]. The mapping of the physical cells and the logical cells in a row is shifted by a certain amount at each write, This technique is designed to mitigate the difference in wear of cells in a row. In [96], it is shown that writes to a specific row in general do not modify all bits in the row but tend to affect only a small subset (low order bits for example). By shifting the row, the physical location of a particular bit changes, hence distributing the writes to a single bit over a number of physical cells. It is proposed to use coarser shift instead of small ones (closer to 1bit shift), since modified bits tend to cluster [96].

Segment Swapping. In segment swapping, each bank of memory is subdivided into fixed size segments. Each segment is composed of a number of rows. By using of a mapping table, the mapping of a logical segment and a physical segment can be made flexible, thus allowing any logical segment to map to any physical segment in that bank. Segment swapping exchanges both the mapping between logical segments and the data, preserving the information but exchanging the physical segment that now holds the data. Segment swapping is proposed in [96] to create a more coarse level wear-leveling. The key to the technique is the selection of the segments that are swapped. The segments chosen in [96] are the ones with the least number of writes since the last swap (above a threshold). Segments that are 1MByte in size are used to reduce the overhead in counters and the size of the mapping table. The memory is unable to respond to requests whenever a segment is being swapped, so even when the number of swaps is small (every $2 \cdot 10^6$ writes) the latency of copying the data can be significant, since 2MBytes have to be transferred.

Fine-Grained Wear-Leveling. This technique is used to achieve intra-page wear-leveling [64]. Each page of 4KBytes is divided into 16 sub-pages and a shift number is added to the logical sub-page number to map to the corresponding physical sub-page. In [96], this shift number is changed only when the page is allocated by the operating system and it is not changed until the page is freed. The shift number is allocated randomly whenever it is changed.

Start-Gap. Start-gap is a low overhead wear-leveling technique [65]. The low overhead is achieved by using a programmable mapping function instead of a mapping table. The mapping function uses two counters, the start and gap counters. The start counter marks which physical address corresponds to the logical address 0 and the gap marks the start of the gap, a sequence of physical addresses that are not currently mapped to any logical address. The physical pages work as a circular buffer with the mapping determined by a start and gap counters. Periodically, based on number of writes, the gap counter is incremented and the information stored at the previous end of the gap is copied to the physical address pointed by the gap counter. This action remaps the logical address that corresponds to the gap entry to a new physical address. Since the physical memory is treated as a circular buffer, a particular logical address will point to all physical addresses during the lifetime of the memory.

Some memories utilize error correction, and in NAND Flash and PCM, ECC has been proposed as a way to deal with cell failures. ECC reserves additional bits to store redundant information allowing errors to be detected and corrected. ECC is heavily used in servers DRAM main memory where memory errors cannot be tolerated. In a write, the error correcting code is generated and stored in addition to the original information. In a read, the stored error correcting code is compared to the one generated on the fly, if the original error correcting code and the newly generated code do not agree, an error has occurred. The ECC is capable of correcting errors up to a determined number of incorrect bits and detecting but not correcting a larger number of incorrect bits. The limits are determined by the number of additional bits used for ECC. In [69], ECC is determined to be detrimental to PCM and Error-Correcting Pointers (ECP) is proposed. ECC is detrimental to PCM because it requires more bits to be written for each write, increasing power consumption and decreasing write bandwidth. ECC also increases the number of bits modified in each write, because a single data bit changed will affect some or all bits in the error correcting code. ECC is used to detect bits that changed after writes which is useful in DRAM or flash, but not so useful in case of PCM since after a bit is successfully written it will not fail. ECP repurposes the excess bits to be used as pointers and replacements bits. When a write to a memory cell fails, a set of pointer and replacement bit is allocated and the correct value is stored in the replacement bit and the pointer contains the number of the failed bit. This removes the need of computing the error correcting code both in the write and the read, and reading the correct bit is a matter of swapping the incorrect bits pointed by the ECP code. ECP also avoids increasing the number of bits to be written by changing the pointer only when a new failure is detected otherwise the pointer and replacement bits are kept at 0. It is shown that ECP leads to a longer lifetime than ECC.

PCM write latency is one of the major limitations of the technology. Even though PCM reads are expected to be several times slower than DRAM reads, PCM writes are one or more orders of magnitude slower than PCM reads [42, 96]. Write bandwidth is also severely limited by power, since each cell written consumes a significant amount of power, limiting the total number of cells that can be written simultaneously. Read bandwidth does not suffer from that limitation and it is one to two orders of magnitude higher than write bandwidth. One of the simple solutions that has been adopted is write avoidance: to reduce the number of writes to PCM, by both reducing the total number of write operations and the number of bits that are written. Write avoidance has a significant energy and latency impact. Other performance improving techniques have been proposed such as write cancellation and a improved PCM requests scheduler.

Write Avoidance. Flip-N-Write can reduce the write access traffic to the PCM devices by reducing the number of bits that are simultaneously written to the memory [7]. Partial writes and Read-before-write when used at a line/row level potentially can remove unnecessary writes [96]. The drawback of partial writes is the requirement of additional dirty bits to store the dirty status in a higher granularity. Read-before-write requires a read operation before a write and in worst case can have a small increase in latency since reads are much faster than writes.

Request Preemption and Pausing. PCM Request preemption is used to improve performance by reducing the impact of high latency operations. Both reads and writes can be preempted [61, 95]. The advantage of preemption is reduction of latency for higher priority requests even if another operation is already ongoing for that bank. Read preemption increases energy consumption since the operation has to be reexecuted in the future. Writes are the large latency offender in PCM, they are up to 10x slower than reads and are often not in the critical path. Write preemption (write cancellation) has the potential to improve latency at the cost of energy efficiency and memory lifetime [61]. The additional energy and wear are consequences of the reexecution of the write since the previous write was not completed and the stored value differs from the expected. These techniques try to limit the impact by not allowing preemptions to occur to the same request too many times [61, 95]. An alternative scheme, write pausing, is proposed to pause the write operation between steps in the loop (overprogramming) and resume the write later [61]. This operation does not incur in energy overhead, since the operation is not repeated but restarted from the point it was stopped. The request is paused so no additional latency is necessary beyond the delay caused by the requests interposed to the paused one.

Memory Request Scheduling. A new memory controller is proposed to implement a scheduling algorithm that is more amenable to PCM [95]. The scheduling algorithms proposed in [95] are a modification of PAR-BS [56], which is a fair scheduling algorithm designed for CMP systems and is modified to include write and read preemption in an effort to improve latency for higher priority requests. The preemption of requests that can be close to finishing does not improve latency significantly and can be detrimental to energy efficiency. Threshold limits are implemented to preclude preemption to requests that will take less than the threshold to finish. The advantage of specializing the memory controller to PCM is the ability to explore the difference in internal operations between PCM and DRAM. DRAM, contrary to PCM, requires that a read be followed by a write in a row, since reads are destructive while PCM reads are harmless. PCM can have preemptable reads or writes, while preemption in DRAM are not possible since that would provoke information loss in reads and a single write buffer prevents simultaneous operation (read with a write date buffered). This is possible in PCM since read data do not have to be stored to preserve information.

Zhou et al. [92] studied the memory scheduling algorithms suitable for embedded systems with PCM main memory. It is proposed to reorder PCM requests based on criticality of the requests to reduce latency and the number of missed deadlines.

2.2 MANAGEMENT OF SHARED CACHES

Cache is an important resource that impacts the application/system performance. For an application, the cache replacement policy decides which cache block to evict to make room for an inserted cache block. An effective cache replacement policy can keep the highly reusable data in the cache longer to achieve better cache utility and application performance. In a CMP system with a shared cache, the allocation of the shared cache capacity on applications/cores can greatly influence the overall utility of the shared cache. In this dissertation, we consider both cache replacement and cache partitioning policies which are effective and suitable for asymmetric NVM memory systems by taking into consideration the LLC writeback information. This section discusses the existing cache replacement and cache partitioning techniques.

2.2.1 Cache Replacement

TADIP [27] is a dynamic insertion policy (DIP) that dynamically identifies the application characteristic and inserts single-use blocks (dead on fill) in the LRU position to evict as early as possible. PIPP [87] pseudo partitions cache space to each application by having a different insert position for each application, which is determined using a utility monitor as in UCP. Upon hits, each block is promoted toward the MRU by one position. PIPP also considers the streaming behavior of an application. When an application shows streaming behavior, PIPP assigns only one way and allows promotion with a very small probability (1/128).

Pseudo-LIFO mechanisms are a new family of replacement policies based on the fill stack rather than the recency stack of the LRU [3]. The intuition of pseudo-LIFO is that most hits are from the top of the fill stack and the remaining hits are usually from the lower part of the stack. Pseudo-LIFO exploits this behavior by replacing blocks in the upper part of the stack, which are likely to be unused.

Jaleel et al. proposed re-reference interval prediction (RRIP) [28]. Cache replacement policies use some method of future reference prediction. For example, LRU predicts that all caches hits and misses will be re-referenced near-immediate. Dynamic insertion policies detect either a near-immediate or distance re-reference pattern in runtime, but not both at the same time. If an application shows a mixed pattern of temporal and non-temporal data, dynamic policies cannot hold near-immediate blocks in the cache. RRIP solves the problem of this mixed pattern by inserting incoming blocks in the not near-immediate position and promote blocks toward the near-immediate position upon hits. RRIP also uses a dynamic insertion policy to filter out non-temporal accesses. Wu et al. proposed prefetch-aware cache management [85], which is built on RRIP.

There is limited study on cache replacement for Non-volatile main memory systems. Ferreira et al. [18] proposed a clean-preferred page replacement algorithm named N-Chance for the DRAM cache in a PCM main memory system. N-Chance gives preference to evict clean pages, which reduces writebacks to PCM by coalescing writes in the DRAM cache.

2.2.2 Cache Partitioning

In an associative cache, the cache capacity can be partitioned by allocating either the cache ways or the cache sets. Existing cache partitioning techniques can be classified into two categories: way partitioning [63, 93] and set partitioning [30, 46]. This thesis studies the way partitioning techniques that are effective for NVM memory systems.

Suh et al. first proposed dynamic cache partitioning schemes in chip multi-processors that consider the cache utility (number of cache hits) using a set of in-cache counters to estimate the cache-miss rate as a function of cache size [75, 74]. However, since the utility information is acquired within a cache, information for an application cannot be isolated from other applications' intervention.

Utility-based cache partitioning (UCP) [63] addressed this problem by proposing a utility monitor (UMON) that uses separate structures, including ATD (auxiliary tag directory) and way counters. ATD maintains strict LRU-stack per application. Upon hits in ATD, the corresponding way counters will be incremented. The optimal partition for all applications is determined to maximize the overall number of cache hits.

Kim et al. considered the fairness problem from cache sharing such that slowdown due to the cache sharing is uniform to all applications [38]. Moreto et al. proposed MLP-aware cache partitioning, where the number of overlapped misses will decide the priority of each cache miss, so misses with less MLP will have a higher priority [55]. IPC-based cache partitioning [77] considered performance as the miss rate varies. Even though the cache miss rate is strongly related to performance, it does not always match the performance. However, since the baseline performance model is again based on the miss rate and its penalty, it cannot distinguish GPGPU-specific characteristics. Yu and Petrov considered bandwidth reduction through cache partitioning [89]. Srikantaiah et al. proposed the SHARP control architecture to provide QoS while achieving good cache space utilization [73]. Based on the cache performance model, each application estimates the cache requirement and central controllers collect this information and coordinate requirements from all applications.

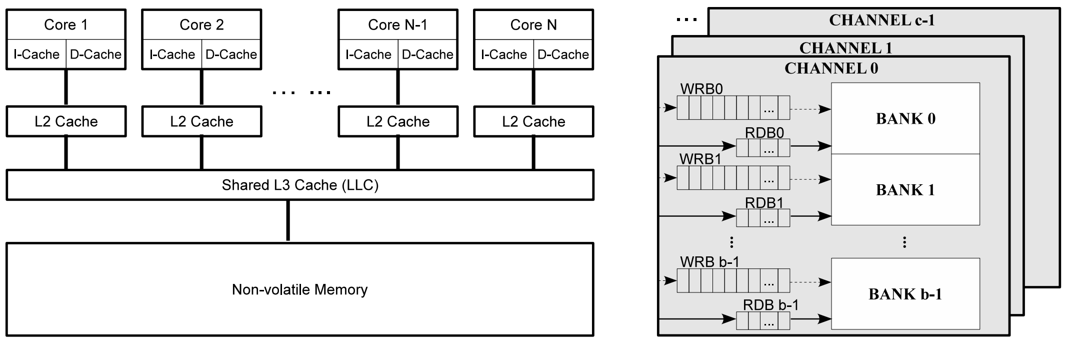
2.3 BANDWIDTH PARTITIONING

Similar to shared cache, memory bandwidth is also a critical shared resource for CMP systems. More specifically, memory bandwidth can be referred to as two different resources: the bus bandwidth and the bank bandwidth. In a DRAM system, the bus bandwidth is more likely to be the system bottleneck due to the fact that DRAM banks can process request at a relative high speed [47]. In contrast, the bank bandwidth becomes the system bottleneck for a system with NVM (e.g., PCM) [94]. An intelligent bandwidth partitioning policy should always target the system bottleneck. This thesis studies the partitioning of the bank bandwidth. This section discusses the existing memory bandwidth management techniques.

Past studies on bandwidth management took two directions. One direction focuses on memory request scheduling. Rixner et al. [59] proposed the FR-FCFS policy to favor requests that hit in the row buffer over other requests, and older over younger requests. Nesbit et al. [57] proposed Fair Queuing Memory Scheduler to prioritize requests according to the QoS objectives. Ipek et al. [25] proposed a reinforcement learning memory controller to adapt scheduling decisions. The other direction manages off-chip bandwidth allocation among applications. Liu et al. [47] used an analytic model to understand how off-chip bandwidth partitioning affects system performance. They also considered the impact of prefetching on bandwidth partitioning in [48]. Wang et al. [83] proposed a different model to reveal the relationship between various bandwidth partitioning schemes and system objectives. Kaseridis et al. [34] balanced bandwidth requirements among chips through workload migration. Ebrahimi et al. [13] proposed Fairness via Source Throttling (FST) to enable fair sharing of the memory. FST throttles cores causing unfairness by limiting the memory requests from the cores. FST is similar to WBP in certain ways. FST achieves fairness by throttling the most aggressive thread (with the smallest slowdown). WBP partitions memory bandwidth such that threads with higher bandwidth utility get more allocation, while threads with lower utility get less bandwidth share. FST throttles aggressive threads (with small slowdown), while WBP "throttles" less efficient threads (with small bandwidth utility). No existing research tries to incorporate writeback information in bandwidth partitioning decisions for non-volatile memory. The techniques proposed by Liu et al. [47] are the most related to ours. Our work differs in two aspects. First, we study partitioning of PCM device bandwidth which is the performance bottleneck in non-volatile memory. Second, our proposal takes into account writeback information which we show to be important to bandwidth partitioning.

2.4 COORDINATED CACHE AND BANDWIDTH PARTITIONING

Bitirgen et al. [2] proposed a global resource manager that uses Artificial Neural Networks (ANNs) to manage the allocation of shared cache capacity and off-chip bandwidth. The study showed that coordinated cache and bandwidth partitioning can outperform unmanaged cache and bandwidth



(a) Architecture (b) Organization of PCM

Figure 4: Hybrid Main Memory Architecture

partitioning. As ANN is a black box optimizer, fundamental insights of how cache and bandwidth partitioning interact remain undiscovered. Also, Bitirgen et al. [2] did not consider the unique asymmetry property of non-volatile memory.

2.5 NON-VOLATILE MAIN MEMORY ARCHITECTURES

To mitigate the shortcomings of non-volatile memory, many researchers propose to add an extra cache layer in front of the non-volatile memory [42, 17, 64]. This cache layer can be used in two ways. It can cache clean and dirty blocks for the memory. It could also be used as a write cache to store dirty blocks. The architecture that we study applies the first method because it benefits both read and write accesses, while the latter approach benefits only writes.

Figure 4(a) shows the main memory architecture that we consider throughout the thesis. Memory operations issued by the CPU are serviced by private L1 and L2 caches. Misses to the L2 cache, as well as writebacks from the L2 cache, are sent to the shared L3 cache, which is the last-level cache. The LLC works as a traditional write-allocate cache with a write-back policy (to the non-volatile memory). That is, when a modified cache line is evicted from the LLC, it must be written to the memory. The LLC is beneficial to PCM main memory: by coalescing a sequence of writes to the same line in the cache, the LLC partially mitigates the negative impacts of memory writes.

The hybrid main memory architecture can benefit non-volatile memory that suffers from the disadvantages of write operations. In this thesis, we study the hybrid memory architecture using Phase Change Memory as the main memory. Figure 4(b) presents the organization of PCM main memory with c channels of b banks each. Each bank has a r-entry read buffer (RDB) and a w-entry write buffer (WRB) for pending requests. When a writeback of a cache line from the LLC occurs, a write request is sent to the PCM, which is queued at a write buffer. The application progresses without delay if the write buffer has available entries since the writebacks are not on the critical path. However, the application stalls when the write buffer is full and cannot accept the request.

3.0 CACHE PARTITIONING USING WRITEBACK INFORMATION

3.1 MOTIVATION FOR WRITEBACK-AWARE CACHE PARTITIONING

Efficiently using the last-level cache (LLC) is crucial to improve system performance. For DRAM systems, Qureshi and Patt show that a reduction in LLC miss rate correlates with an improvement in IPC [63]. In their work, they classify applications as low utility, high utility and saturating utility depending on whether and how much an application benefits from a larger cache. A low-overhead circuit named UMON is introduced to monitor the utility information of each application. They propose Utility-based Cache Partitioning (UCP) to partition the LLC in a way that applications with high utility are given more cache quota, while applications with low utility get a small portion of the LLC. In this way, UCP outperforms LRU.

For hybrid main memory, however, reducing miss rate is insufficient to achieve high system performance. Reducing the number of LLC writebacks is also important due to the asymmetric property of non-volatile memory. In a hybrid memory architecture, memory writes are much slower than reads, and the memory write bandwidth is much lower than read bandwidth. These characteristics make it likely that the write buffers will become congested. Once the write buffers are full, the application stalls when a writeback from the LLC occurs. In this case, a writeback is more harmful than a cache miss because a memory write is much slower than a read.

Writebacks can also negatively affect memory lifetime and energy consumption. Non-volatile memory, such as NAND Flash, PCM and STT-RAM, has limited write endurance, and memory writes are more power hungry than reads. As a result, too many writebacks from the LLC shorten the memory lifetime and consume more energy.

To the best of our knowledge, no existing cache partitioning and replacement scheme accounts for writeback information. Figure 5 shows the architectural overview of my approach to support

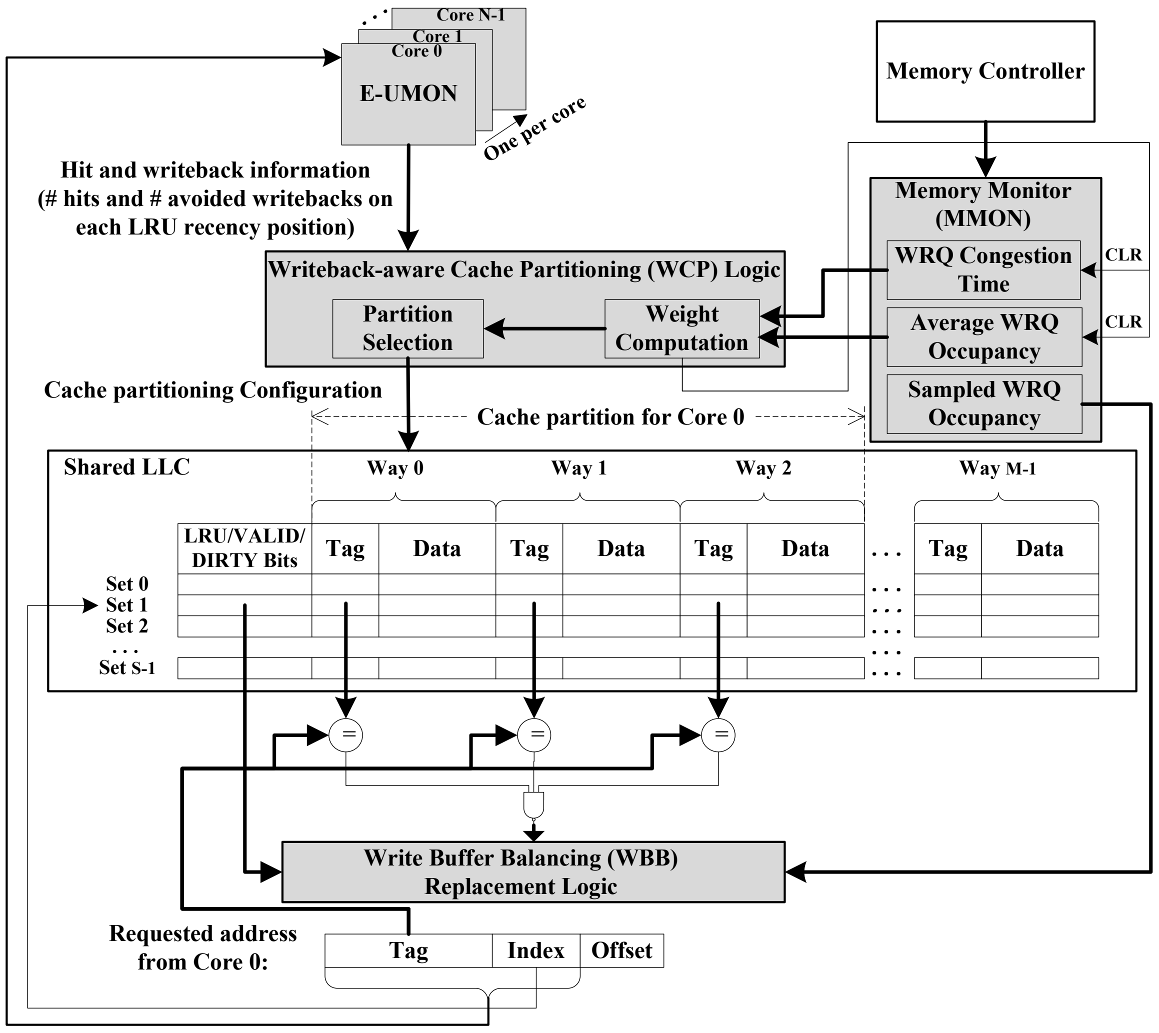


Figure 5: Architectural components to enable WCP

writeback-aware cache partitioning and replacement. It presents four major architectural components (gray colored in Figure 5), as follows. The *extended utility monitor* (E-UMON) is an extension of UMON that tracks hit and writeback information about an application. The *memory monitor* (MMON) tracks information about the write buffers of the memory devices. The *writeback-aware cache partitioning* logic uses information collected by E-UMON and MMON to select a cache partitioning configuration. It includes two modules: *partition selection* and *weight computation*. The *write buffer balancing replacement* logic uses information from MMON to pick a victim entry when a cache miss occurs. Chapter 3.2.2 describes E-UMON. Chapter 3.2.3 explains the cache partition selection scheme. Chapter 3.2.4 describes MMON and the weight computation. WBB replacement is described in Chapter 3.3.

3.2 WRITEBACK-AWARE CACHE PARTITIONING

Writeback information is important for efficient partitioning of the LLC in a hybrid main memory. This section describes the mechanism to track writeback information, and proposes writeback-aware cache partitioning.

3.2.1 Monitoring of Writebacks

The writeback information that E-UMON monitors is based on the concepts of *stack distance* and *stack property*. Stack distance was introduced to study the behavior of storage hierarchies by Mattson et al. [53]. In a replacement policy that has the stack property, such as Least Recently Used (LRU) replacement, each set of a set-associative cache is viewed as a stack. All cache lines in the set are ordered by the last access time, where the top of the stack is the Most Recently Used (MRU) cache line, and the bottom is the LRU cache line. Stack distance is defined as the LRU recency position of a cache line when it is accessed again. It is critical for monitoring miss rate. An access is a hit if the cache associativity is no less than the stack distance.

Figure 6 shows a sequence of references to an M-way set-associative cache. The five accesses are to the same cache line X. Assume that a write-back policy is used by the cache. A dirty cache line X exists in the cache after time t_1 when the first write access to X happens. Assume also that subsequent reads to the same cache line occur at t_2, t_3 and t_4, and another write access occurs at t_5. The stack distances for these five accesses are 4, 3, 5, 2 and 7. For instance, the cache line X has a LRU recency position of 3 before the read access happens at t_2, and is promoted to the top of the LRU stack after t_2. The following are some observations about the five references to the same cache line X:

- If $1 \leq M < 3$, then the dirty cache line X is evicted from the cache. The eviction causes a writeback to the next level cache/memory between t_1 and t_2.

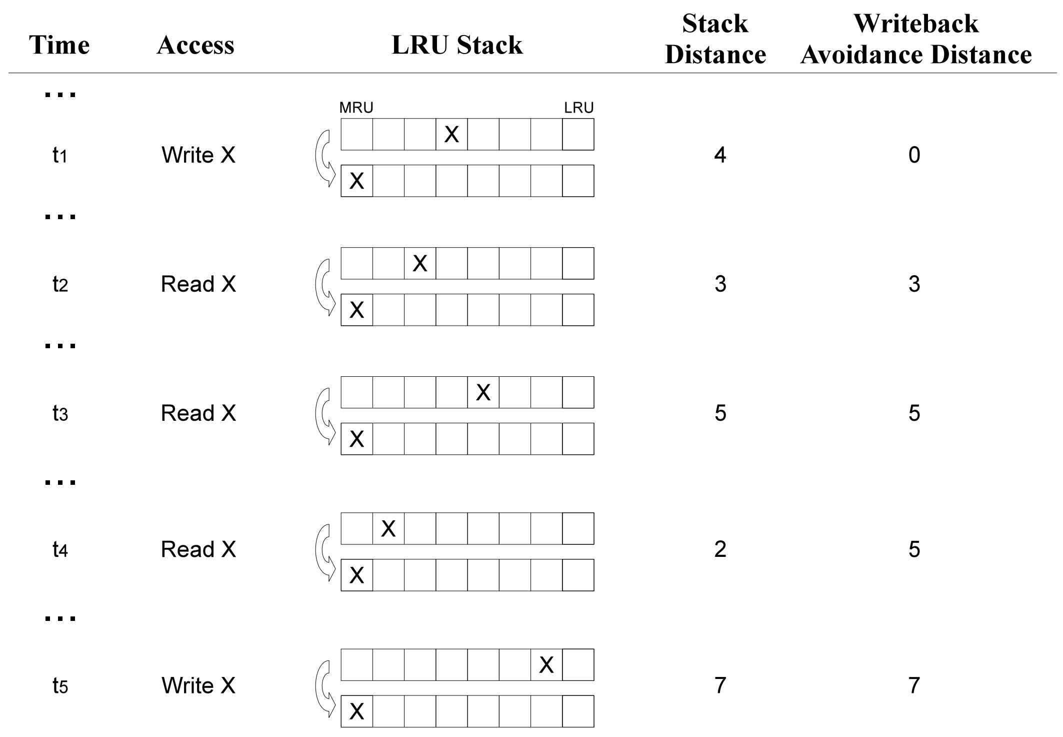


Figure 6: An example of tracking writebacks from a cache

- If $3 \leq M < 5$, then the writeback of cache line X happens between t_2 and t_3.
- Finally, if $5 \leq M < 7$, then cache line X is written back to the next level cache/memory between t_4 and t_5.

Hence, to avoid the writeback of cache line X between t_1 and t_5, the cache associativity should be at least 7, which is the maximum value among 3, 5 and 7.

Here are two conclusions derived from the example. First, each write access leads to a potential writeback. A dirty line exists in the cache after a write access. The dirty line might eventually be evicted and cause a writeback. Second, a writeback is avoided if the dirty line stays in the cache until the next write access happens to the same line.

Consider a general example for an M-way set-associative cache. There is a sequence of accesses to the same cache line X, $\{W_0, R_1, R_2, ..., R_n, W_{n+1}\}$, including two write accesses (W_0 and W_{n+1}) and n read references ($R_1, R_2, ..., R_n$). The stack distances for these accesses are

$SD_0, SD_1, SD_2, ..., SD_n, SD_{n+1}$. Write access W_0 causes a writeback when the dirty cache line X is evicted before the next write access W_1 arrives. Cache line X will not be written back to the next level cache/memory between two write accesses W_0 and W_1 if and only if accesses $R_1, R_2, ..., R_n$ and W_{n+1} are hits on the cache. In other words, a writeback of the cache line X could be saved if and only if $M \geq max(SD_1, SD_2, ..., SD_n, SD_{n+1})$.

The *writeback avoidance distance* of write access W_0 to cache line X is defined as the maximum stack distance of all the accesses to cache line X which happens between W_0 and the next write access to cache line X. A write access causes a writeback if the cache associativity is less than its writeback avoidance distance.

It is known that the LRU policy obeys the stack property, which means that an access that hits in a LRU managed cache containing M ways is guaranteed to also hit if the cache has more than M ways. The stack property also applies to writebacks: a write that does not cause a writeback in an M-way cache is guaranteed to also not cause a writeback in a cache with more than M ways. The reason is that all accesses between two consecutive writes are hits on a cache with more than M ways, if they are all hits on an M-way cache.

3.2.2 Extended Utility Monitors (E-UMON)

To make cache partitioning decisions among applications, the partitioning scheme needs to know the hit and writeback information of applications with different number of ways. Since LRU obeys the stack property, it is possible to track this information with a shadow tag array containing the same number of ways as the shared cache.

Mechanisms to monitor hit information have been described by many researchers [54, 63]. Similar to UMON [63], E-UMON monitors hit information of an application executing on core c through a shadow tag array and hit counters. The shadow tag array has the same associativity as the shared LLC. Each shadow tag entry has three fields: valid bit (V), tag address (TAG) and LRU bits (LRU) that track stack distance of each access. Dynamic set sampling [62] is applied to reduce the hardware overhead. Our evaluation shows that sampling 32 sets is sufficient for a cache with 2048 sets.

The shadow tag array is updated on every access to the LLC, using LRU replacement. Of

Algorithm 3.1 Monitoring Hits

```
parameters:
$i$: Core id
$st$: The accessed shadow tag entry
$addr$: The accessed address

if $st.TAG == GetTagAddress(addr)$ then
    $st.SD \leftarrow GetLruRecencyPosition()$
    $HIT^i_{st.SD} \leftarrow HIT^i_{st.SD} + 1$
end if
```

Algorithm 3.2 Monitoring Writebacks

```
parameters:
$i$: Core id
$st$: The accessed shadow tag entry
$addr$: The accessed address

if $st.TAG == GetTagAddress(addr)$ then
    $st.SD \leftarrow GetLruRecencyPosition()$
    if $st.D == TRUE$ then
        $st.WAD \leftarrow max(st.WAD, st.SD)$
    end if
    if the access is a write operation then
        $AWB^i_{st.WAD} \leftarrow AWB^i_{st.WAD} + 1$
        $st.WAD \leftarrow 0$
    end if
else
    $st.WAD \leftarrow 0$
end if
```

course, E-UMON only updates the tag entries that belong to the sampled sets. E-UMON tracks the hit information through a set of counters, based on the information in the shadow tag array (see Algorithm 3.1). For an M-way cache, M counters $HIT^i_0, HIT^i_1, ..., HIT^i_{M-1}$ are used to get hit information. HIT^i_c is the number of additional hits when the c^{th} way is available for core i, where $0 \leq c \leq M-1$. Counter HIT^i_{sd} is incremented on each hit on the shadow tag array, where sd is the stack distance.

To obtain writeback information, E-UMON adds two additional fields to each shadow tag entry: dirty bit (D) and the writeback avoidance distance (WAD). Writeback information is recorded through the avoidable writeback counters $AWB^i_0, AWB^i_1, ..., AWB^i_{M-1}$, where AWB^i_c is the number of avoidable writebacks if the c^{th} way is available for $0 \leq c \leq M-1$ (see Algorithm 3.2). The writeback avoidance distance for each write access is tracked through the WAD field of the shadow tag entry. Initially, the WAD fields of all shadow tag entries are initialized to 0. Once the shadow tag entry is modified (i.e., the dirty bit D has been set), WAD records the maximum stack distance of the following accesses to the entry until the next write occurs. When a write hit happens, counter AWB^i_{wad} is incremented, where wad is the writeback avoidance distance. The WAD field is reset after each write hit, and starts tracking the writeback avoidance distance of the newly-arrived write access.

With the knowledge of hit and writeback information, the partitioning scheme can compute the

Optimization Objective	Partitioning Weights
Cache Miss Rate [63]	$w_{Hit} = 1, w_{Awb} = 0$
Memory Dynamic Energy	$w_{Hit} : w_{Awb} = E_R : E_W$, where E_R and E_W are energy consumption of a memory read and write request.
Memory Writes	$w_{Hit} = 0, w_{Awb} = 1$

Table 2: Partitioning weights for different optimization objectives

number of hits ($hits_{M'}$) and avoidable writebacks ($avoidablewritebacks_{M'}$) for an M'-way cache, where $M' < M$, as follows.

$$hits_{M'} = \sum_{c=0}^{M'-1} HIT_c^i \tag{3.1}$$

$$avoidablewritebacks_{M'} = \sum_{c=0}^{M'-1} AWB_c^i \tag{3.2}$$

3.2.3 Cache Partition Selection

Let π_i be the number of the ways used by core i. For cache way partitioning, $\bar{\pi} = \{\pi_0, ..., \pi_{N-1}\}$ is a valid partition for a system with N cores and an M-way shared LLC, when it satisfies:

$$\sum_{i=0}^{N-1} \pi_i = M, \text{where } 1 \leq \pi_i \leq M - N + 1. \tag{3.3}$$

Based on the hit and avoidable writeback counters in E-UMON, *Writeback-aware Cache Partitioning* (WCP) tries to select a valid partition that maximizes a weighted sum of the number of misses and the number of writebacks:

$$\sum_{i=0}^{N-1} \sum_{c=0}^{\pi_i-1} (w_{Hit} \cdot HIT_c^i + w_{Awb} \cdot AWB_c^i) \tag{3.4}$$

Two parameters w_{Hit} and w_{Awb} are *partitioning weights* assigned to hit counters and write-back counters. These two parameters determine the relative importance of reducing misses and minimizing writebacks. The suitable partitioning weights differ for various system optimization objectives.

Table 2 gives the partitioning weights for three optimization objectives. UCP was proposed to improve the system performance of a DRAM system by reducing the overall LLC miss rate [63]. UCP is a specific case of WCP with $w_{Hit} = 1$ and $w_{Awb} = 0$. If the dynamic power efficiency of the memory is the criteria, then the partitioning weights should satisfy $w_{Hit} : w_{Awb} = E_R : E_W$, where E_R and E_W are energy consumption of a memory read and write request. The reason is that a memory read is avoided for a hit, while a memory write is saved for each avoided writeback. If reduction of memory writes is the objective, then $w_{Hit} = 0$ and $w_{Awb} = 1$ should be used.

3.2.4 Weight Computation for Performance Optimization

The appropriate partitioning weights for performance optimization are not obvious. Different workloads might favor different weights. Consider two extreme scenarios. For applications with few write memory accesses, there are limited writebacks from the LLC, which is not enough to saturate the memory write bandwidth. In this case, the system performance is affected by the hit rate—not the number of writebacks, and thus, the best choice is $w_{Hit} = 1$ and $w_{Awb} = 0$ (WCP behaves the same as UCP). At the other extreme, for applications with many write accesses, many writebacks cause applications to stall due to overwhelmed memory bandwidth. The penalty of incurring a writeback is much higher than a cache miss as a memory write operation is much slower than a read. Therefore, the parameters should satisfy $w_{Awb} > w_{Hit}$. More importantly, these two cases might happen for the same applications from one execution phase to another. To simplify the problem, a new parameter w_c is defined as $w_c = w_{Awb}/w_{Hit}$, which is used in the following maximization problem:

$$\sum_{i=0}^{N-1} \sum_{c=0}^{\pi_i - 1} (HIT_c^i + w_c \cdot AWB_c^i) \tag{3.5}$$

It is experimentally shown in Chapter 3.5.1 that a universal partitioning weight does not exist. A partitioning weight that works the best for one workload might be a bad choice for another workload. A good solution should adjust w_c dynamically for different workloads, or even for different phases of the same workload. The adaptation of the cache partitioning weight is based on two observations.

First, w_c should be proportional to the urgency of reducing writebacks. It is not desirable to reduce (or increase) the total number of misses at the cost of much higher (lower) number of

Algorithm 3.3 Adjusting Cache Partitioning Weight (w_c) Dynamically

```
parameters:
$WB$: Number of write buffers
$AWBO_i$: The average occupancy of write buffer $i$
$WBCT$: Number of cycles when there exists a congested write buffer
$EPOCH\_CYCLES$: Number of cycles per epoch

$MaxAWBO = \max_{i=0}^{WB-1} AWBO_i$

if $MaxAWBO \geq ThreshOccuHigh$ then
    if ($WBCT/EPOCH\_CYCLES \geq ThreshWBCongest$) and ($w_c < wBoostValue$) then
        $w_c \leftarrow wBoostValue$
    else
        $w_c \leftarrow w_c + 1$
    end if
else if $MaxAWBO \leq ThreshOccuLow$ then
    $w_c \leftarrow w_c - 1$
end if
```

writebacks. For example, if the write buffers are rarely congested, then it is unimportant to reduce the number of writebacks for performance optimization. In this case, a small weight is favorable. However, a large weight should be used to reduce miss rate as well as writeback rate, if the write buffers are congested.

Second, applications tend to have bursts of writes that cause bursts of writebacks from the LLC. It is beneficial to recognize the burst periods as soon as possible and choose a big weight to mitigate the negative impact of saturated memory bandwidth.

The cache partitioning weight w_c is adjusted periodically at *epochs* that last 5 million cycles. The weight computation module selects a weight at the beginning of each epoch based on information from MMON. MMON has three types of information about the phase change main memory: write buffer congestion time, sampled write buffer occupancy, and average write buffer occupancy. *Write buffer congestion time* (WBCT) is the number of cycles when a write buffer is full. *Sampled write buffer occupancy* (SWBO) records the occupancy of write buffers. The occupancy of a write buffer is defined as the ratio of the number of pending requests to its capacity. *Average write buffer occupancy* (AWBO) is the average occupancy of write buffers during an epoch. WBCT and AWBO are reset at the beginning of each epoch, while SWBO is updated at fixed-length time intervals called quanta (i.e., 10 thousand cycles).

The weight is initialized to 0 when the system starts and changed by the cache partitioning weight adjustment policy as described in Algorithm 3.3. The proposed scheme uses the parameters $ThreshOccuHigh = 80\%$, $ThreshOccuLow = 50\%$, $wBoostValue = 5$ and $ThreshWBCongest = 20\%$.

- If there is a write buffer with high occupancy ($AWBO \geq ThreshOccuHigh$) and the write buffers are congested for a long time ($WBCT/EPOCH\_CYCLES \geq ThreshWBCongest$), then the weight is either boosted to $wBoostValue$ or incremented. If the weight is already larger than $wBoostValue$, then it is incremented. Otherwise, the weight is set to $wBoostValue$.

- If a write buffer has high occupancy ($AWBO \geq ThreshOccuHigh$) but the write buffers are not congested for a long time ($WBCT/EPOCH\_CYCLES < ThreshWQCongest$), then the weight is incremented.

- If all write buffers are underutilized ($AWBO < ThreshOccuHigh$), then the weight is decremented.

WCP partitions the LLC after the selection of the partitioning weight. WCP selects the cache partition that produces the maximum weighted sum of total hits and saved writebacks (Equation (3.5)). Our scheme guarantees that each partition has at least one way ($\pi_i \geq 1$ for $0 \leq i \leq N-1$). To better blend past and recent information, the hit counters and saved writeback counters in E-UMONs are halved after each epoch.

To enforce the cache partitioning decisions of WCP, a ($\log N$)-bit core identifier field is introduced to the tag-store entry of each cache line for a N-core processor. The core identifier indicates which core owns the cache line. The cache replacement engine ensures that each core does not use more cache resources than its quota. On a cache miss, the replacement engine counts the number of cache blocks within the set that belong to the application that causes the miss. If the application uses more cache blocks than its quota, the replacement engine replaces the LRU block among all the blocks that belong to that application. Otherwise, the LRU block among all the blocks that do not belong to the application is evicted.

Algorithm 3.4 Write Buffer Balancing Replacement

```
output:
$VictimEntry$: the victim entry

Populate $S$ with candidate entries
Set $S \Leftarrow SortByLastTimeUsed(S)$ $\{S[1] \leftarrow LRU\}$
if there exists invalid entries then
    $VictimEntry \leftarrow$ an invalid entry
else
    $VictimEntry \leftarrow S[1]$
    for $i \leftarrow 1$ to $LengthOf(S)$ do
        $WriteBufferId \leftarrow GetMappedWriteBufferId(S[i])$
        if $(IsEntryClean(S[i]) == TRUE)$ or
        $(GetSampledOccupancy(WriteBufferId) < ThreshOccuHigh)$ then
            $VictimEntry \leftarrow S[i]$
            break
        end if
    end for
end if
```

3.3 WRITE BUFFER BALANCING REPLACEMENT

When a dirty cache line is evicted from the LLC, it will cause a memory write. The memory write request will be queued in the corresponding write buffer when possible, and the application continues without stalling. However, the application stalls if the corresponding write buffer is full. *Write Buffer Balancing* (WBB) replacement avoids these delays by evicting (a) a clean line or (b) a dirty cache line which is mapped to a write buffer with fewer pending requests. By distributing writebacks evenly among write buffers, WBB reduces the delays due to writebacks from LLC. Note that WBB is in charge of replacing cache lines within cache partition (i.e., for each application) and is orthogonal to the cache partitioning policy.

Algorithm 3.4 shows the pseudo-code for WBB that finds a victim entry. When a cache miss happens, an invalid entry is used whenever possible. When all entries are valid, starting from the LRU to the MRU entry, WBB searches the candidate entries, and checks whether there exists one that is (a) clean or (b) mapped to a write buffer under relatively low load (i.e., with a sampled occupancy less than $ThreshOccuHigh$). Such an entry is selected as a victim, if it exists. Otherwise, the LRU entry is selected as the victim. WBB behaves the same as LRU when the applications generate few writes. In this case, each write buffer has few pending requests.

3.4 EXPERIMENTAL METHODOLOGY AND METRICS

Simics [51] is used to model the processor, L1 and L2 caches, and generate memory traces, which are input to an in-house cycle-accurate simulator that models the shared DRAM last-level cache and PCM. Our baseline system is a 8-core CMP with a three level cache hierarchy. The L1 and L2 caches are private to each core. Each L1 instruction and data cache is a 4-way 64KB cache and each L2 cache is a unified 2MB 8-way associative cache. The sizes of the L1 and L2 caches are unchanged throughout our study. The DRAM cache for a 8-core system is a 64-way 32MB shared last-level cache (4MB per core). Each core issues two instructions per cycle, and has a 168-entry instruction window. An evaluation of the proposed schemes on 2-, 4- and 16-core systems is included to validate the scalability of the proposed schemes. The DRAM LLC size (associativity) is proportional to the number of cores. For example, a 4-core system has a 32-way 16MB DRAM cache. Table 3 shows the parameters in the default configuration. The permutation-based page interleaving scheme [91] is used as the address mapping scheme for phase change main memory. Assuming there are 2^q write buffers, the lower order q bits of the DRAM LLC tag and the lower order q bits of the DRAM cache set index are used as the input to a q-bit bitwise XOR logic to generate the write buffer index. This interleaving scheme requires negligible area overhead compared to E-UMON and WCP, which dominate in total overhead (see Chapter 3.5.7). Note that

Processor	8-core CMP, 4GHz, Out-of-Order 168-entry instruction window, 2 instructions per cycle in each core
L1 Icache, Dcache	private, 64KB, 64B line, 4-way, LRU
L2 Cache	private, 2MB, 256B line, 8-way, LRU 15 cycles access latency
DRAM LLC	shared, 32MB, 256B line, 64-way, writeback policy 50 cycles access latency
Main Memory	64GB PCM, 4 channels of 8-banks each 32-entry write buffer per bank 8-entry read buffer per bank read priority scheduling via write pausing [61]
PCM	read: 200 cycles (50ns) latency, 1 J/GB energy consumption write: 4000 cycles (1μs) latency, 6 J/GB energy consumption

Table 3: Baseline configuration

Type	Benchmark
W-L	400.perlbench, 403.gcc, 416.gamess, 434.zeusmp, 435.gromacs, 436.cactusADM, 444.namd, 447.dealII, 453.povray, 454.calculix, 456.hmmer, 458.sjeng, 464.h264ref, 465.tonto, 471.omnetpp, 481.wrf, 482.sphinx, 483.xalancbmk
W-ML	401.bzip2, 410.bwaves, 433.milc, 437.leslie3d, 445.gobmk, 473.astar
W-MH	450.soplex, 459.GemsFDTD, 463.libquantum
W-H	429.mcf, 470.lbm

Table 4: Benchmark classification based on writebacks per 1000 instructions

the NVM is provisioned with write pausing [61] to enable fast processing of the memory reads.

Benchmarks. The SPEC CPU2006 benchmarks are used for evaluation. Table 4 characterizes the benchmarks in terms of writebacks. Based on writebacks per 1K instructions (WBPKI), the benchmarks are classified into four types: Heavy (W-H), Medium-Heavy (W-MH), Medium-Light (W-ML), and Light (W-L). From this classification, fifteen workloads are created, as described in Table 5. The WBPKI and the percentage of memory bandwidth consumed by write requests (WB%) for each workload for the baseline scheme (UCP) are also listed. *Light*, *Medium*, and

Workload	WBPKI	WB%	Benchmarks
Light1	1.42	11.3%	cactusADM, hmmer(3), perlbench(2), sjeng(2)
Light2	1.51	12.1%	cactusADM(2), bwaves(2), hmmer, sjeng(3)
Light3	1.98	16.7%	bzip2(2), bwaves, zeusmp(3), xalancbmk(2)
Medium1	5.38	22.1%	astar, bzip2(3), GemsFDTD(3), libquantum
Medium2	5.42	22.9%	bzip2(3), leslie3d, libquantum(2), soplex(2)
Medium3	5.63	23.7%	astar(2), leslie3d(2), soplex(2), GemsFDTD(2)
Medium4	6.10	25.4%	astar(2), cactusADM(2), GemsFDTD(2), libquantum(2)
Medium5	6.68	27.5%	bzip2(2), bwaves, GemsFDTD(3), libquantum(2)
Medium6	7.29	34.8%	GemsFDTD(2), lbm(2), soplex(2), xalancbmk(2)
Medium7	7.89	36.3%	bwaves(3), mcf(3), xalancbmk, zeusmp
Medium8	8.87	38.4%	mcf(2), soplex(2), xalancbmk(2), zeusmp(2)
Medium9	9.71	43.2%	leslie3d, mcf(3), soplex(3), xalancbmk
High1	13.11	67.5%	lbm(2), mcf(2), omnetpp(2), xalancbmk(2)
High2	19.37	74.2%	mcf(2), omnetpp(3), xalancbmk(3)
High3	22.41	89.4%	gobmk(3), mcf(3), xalancbmk(2)

Table 5: Multiprogrammed workloads of Light, Medium and High

Heavy are workloads with light, medium and heavy LLC writebacks.

Each benchmark was run in Simics. Simulation for a 8-core system is continued until each benchmark executes at least 1 billion instructions. If one benchmark finishes its 1 billion instructions before the other benchmarks finish, it is restarted so that the eight benchmarks continue competing for shared resources such as DRAM LLC and the memory bandwidth.

Evaluation Metrics. To understand the performance impact, three performance metrics are measured in the experiments. The three performance metrics are weighted speedup [72], throughput and fairness, which are defined as follows:

$$\text{Weighted Speedup} = \sum (CPI_{alone,i}/CPI_i); \tag{3.6}$$

$$\text{Throughput} = \sum 1/CPI_i; \tag{3.7}$$

$$\text{Fairness} = N/\sum (CPI_i/CPI_{alone,i}), \tag{3.8}$$

where CPI_i is the CPI (average number of cycles per instruction) of application i when it executes with other applications, and $CPI_{alone,i}$ is the CPI of the same application when it executes alone. The weighted speedup metric indicates reduction in execution time. The throughput of the system is indicated by the sum of the IPCs (e.g., the inverse of CPI). The fairness metric is the harmonic mean of normalized IPCs [31]. This section also evaluates the impact of the proposed schemes on memory lifetime and energy consumption.

3.5 EVALUATION RESULTS AND ANALYSIS

To understand the effectiveness of our schemes, WCP is compared against the state-of-the-art LLC partitioning policy, UCP [63]. Sensitivity studies on memory write latency, number of memory banks, write buffer length and the epoch length are also included.

3.5.1 Dynamic Weight Adjustment for WCP

Figure 7 shows the weighted speedup and throughput of WCP with different partitioning weights normalized to SHARE. SHARE is the baseline scheme which allows all applications compete for

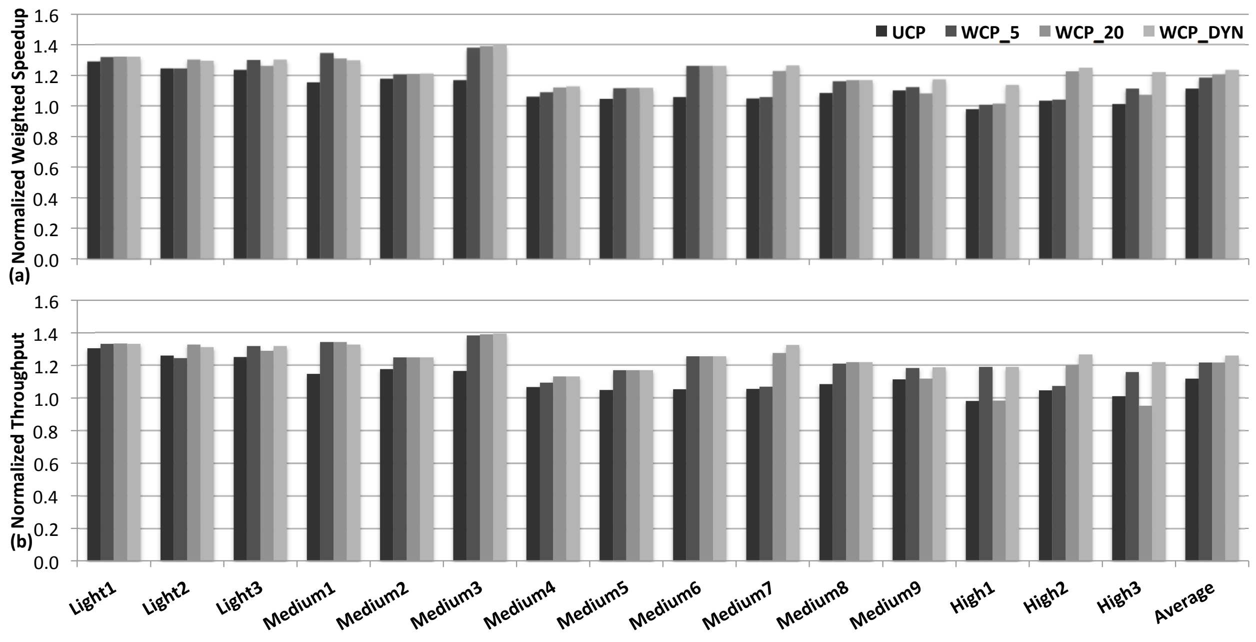


Figure 7: Performance impact of cache partitioning weight

the shared LLC capacity and memory bandwidth without management. That is, SHARE assumes LRU as the cache replacement policy. Results are shown normalized to SHARE, unless otherwise noted. The bar labeled "average" is the average result of all 15 workloads. Note that WCP with $w = 0$ is identical to UCP. The results clearly show that there is no universal static weight for different workloads. From the weighted speedup perspective, there are four workloads that favor a static weight of 5, four benefit from a weight of 20, and seven workloads perform similarly with a weight of 5 or 20. WCP_DYN (WCP with dynamic weight adjustment) outperforms UCP, WCP_5 and WCP_20 for all but three workloads. On average, WCP_5 improves weighted speedup by 6.5%, WCP_20 improves weighted speedup by 8.3%, and WCP_DYN improves weighted speedup by 11.1%, compared to UCP. For *High1*, the dynamic weight adjustment policy outperforms the best static weight by more than 10%. This is because WCP_DYN adapts to different workloads and different execution phases of the same workload. This illustrates the importance of adjusting partitioning weight based on the urgency of reducing writebacks. For the remainder of the thesis, WCP by default means WCP_DYN.

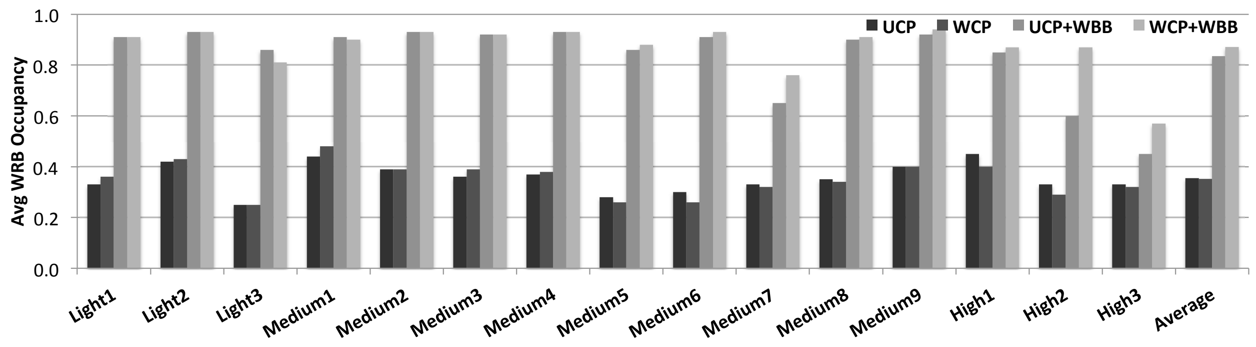


Figure 8: Average occupancy of write buffers when there are congested write buffer(s)

3.5.2 Unbalanced Write Buffers

Figure 8 shows the average occupancy of all write buffers when at least one write buffer becomes congested. The instantaneous occupancy of a write buffer is defined as the ratio of the number of pending requests to its capacity. Each time a write buffer becomes full, the instantaneous occupancy of the write buffers are averaged. The average instantaneous occupancy of write buffers are then averaged over time. The result shows that the distribution of writebacks among write buffers is highly skewed without write buffer balancing. On average, when one write buffer is full, the other write buffers are underutilized (with occupancy less than 0.4) for UCP and WCP. When applying WBB, all write buffers are evenly utilized. WBB helps both UCP and WCP to reduce the chances of filling up write buffers. For the remainder of the thesis, WBB is the default cache replacement policy unless otherwise specified.

3.5.3 Memory Lifetime

Figure 9 shows the writebacks per 1K instructions (WBPKI) of UCP, WCP, UCP+WBB and WCP+WBB normalized to SHARE. On average, UCP has similar WBPKI as SHARE. As an extreme example, for *Medium5*, UCP incurs more than 20% more writebacks than SHARE. This result happens because UCP tries to minimize the total number of misses at the cost of possibly generating more writebacks. Unlike UCP, WCP takes into account information of both misses and writebacks, and reduces the total number of misses and writebacks. For eight out of the fifteen

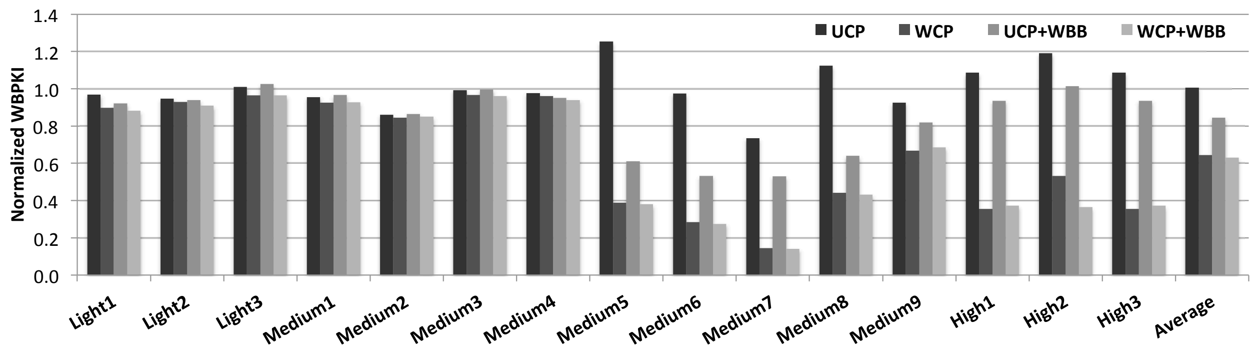


Figure 9: Impact on memory lifetime

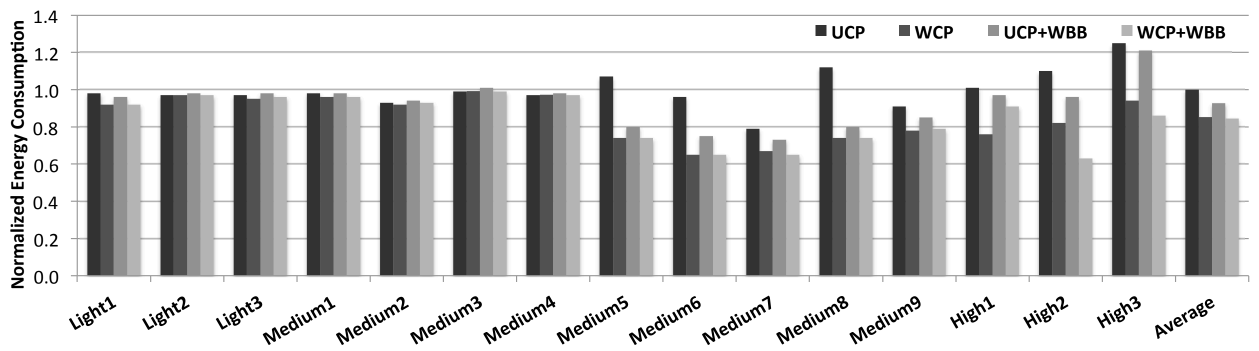


Figure 10: Impact on memory energy consumption

workloads, WCP reduces WBPKI by at least 25% over UCP. For Medium7, WCP achieves a reduction of 80% on WBPKI over UCP. Intuitively, for workloads with few writebacks, WCP is not very effective in reducing writebacks. Assuming that a perfect wear-leveling scheme is applied, the reduction in writebacks directly translates into a prolonging of the memory lifetime. As a result, on average, WCP improves lifetime by almost 56% compared to UCP.

3.5.4 Energy Consumption

Figure 10 shows the energy consumption of UCP, WCP, UCP+WBB and WCP+WBB normalized to SHARE. Overall, SHARE and UCP have similar energy consumption. WCP saves energy by

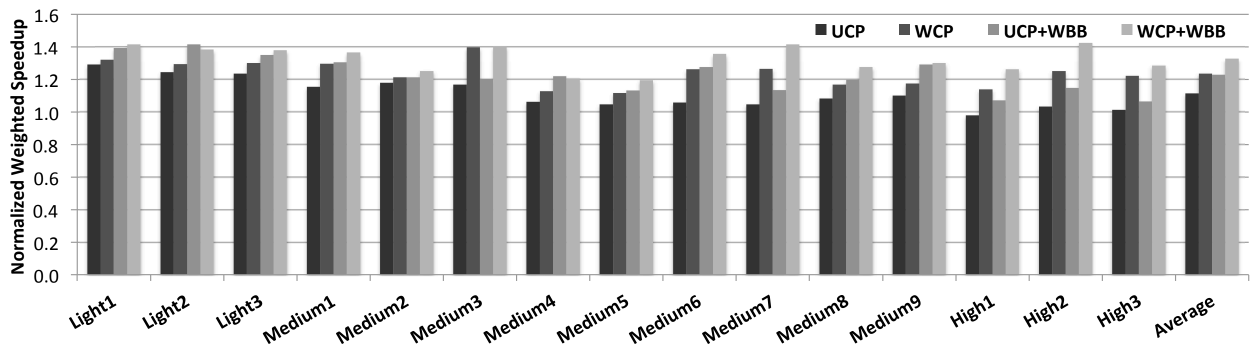


Figure 11: Impact on weighted speedup

minimizing writebacks (i.e., memory writes) over both SHARE and UCP. WCP reduces the energy for eight of the fifteen workloads by reducing memory writes which are energy hungry. WBB saves energy for nine workloads, and increases energy consumption marginally for three workloads. It is noticable that UCP+WBB can also reduce energy consumption. The reason is that applications with many writebacks stall less frequently, and get a relatively larger cache quota when WBB is applied. The writebacks generated by these applications are reduced as a result of a larger cache partition. On average, WCP, UCP+WBB and WCP+WBB reduce energy by 14.8%, 7.3% and 15.5% over UCP, respectively.

3.5.5 Performance

Figure 11 shows the weighted speedup for different policies normalized to SHARE. On average, UCP outperforms SHARE by 11.2% in terms of weighted speedup. For *High1*, however, UCP reduces performance marginally compared to SHARE. This result happens because UCP changes the cache partition configurations once every epoch (5 million cycles [63]) and, therefore, it cannot respond to the phase changes that occur at a finer granularity. LRU (SHARE), on the other hand, naturally responds to such a fine-grained phase change. WCP improves weighted speedup by more than 15% for six out of the fifteen workloads over UCP. For *High2*, WCP outperforms UCP by 21%. The benefits of using WBB are as follows. By distributing writebacks evenly among write buffers, WBB benefits the weighted speedup in spite of the cache partitioning policy. When UCP

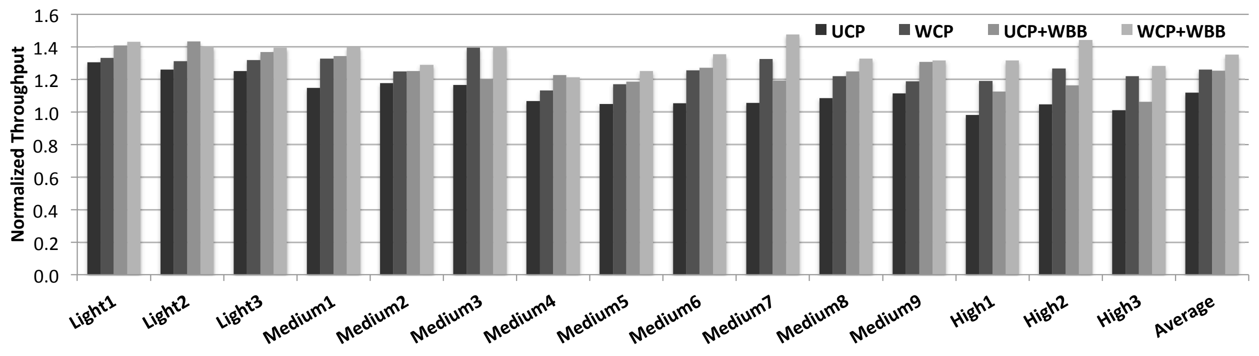


Figure 12: Impact on system throughput

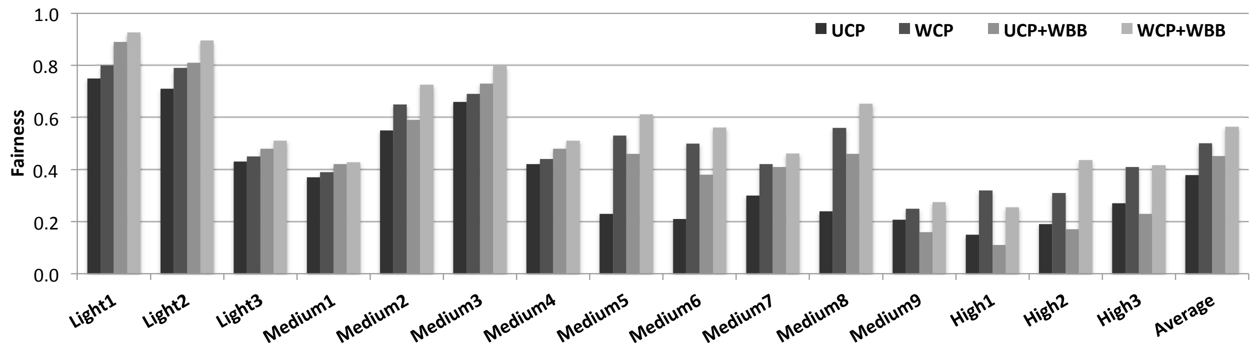


Figure 13: Impact on fairness

is used to partition the cache, WBB improves the weighted speedup by 10% on average. When WCP is applied as the cache partitioning policy, WBB result in an average improvement of 7.4% in weighted speedup.

Figure 12 shows the throughput of the proposed schemes normalized to SHARE. UCP achieves an average of 11.7% throughput improvement over SHARE. WCP further improves the throughput over UCP by 12.7% on average. For nine out of the fifteen workloads, WCP improves the throughput by more than 10% over UCP. WCP+WBB improves the throughput by 21.1% on average over UCP.

Cache partitioning and replacement policies might improve the performance of some applications by severely penalizing others. The harmonic mean of the normalized IPCs considers both

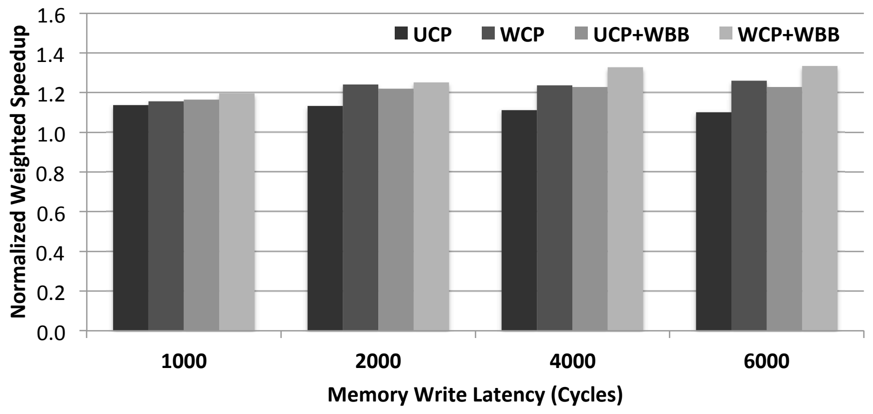


Figure 14: Sensitivity of WCP and WBB to memory write latency

fairness and performance [31]. Figure 13 shows the performance of UCP, WCP, UCP+WBB and WCP+WBB in terms of fairness. On average, WCP+WBB improves the fairness metric by 48.8% compared to UCP.

3.5.6 Sensitivity Study

To analyze the benefit of our schemes under different system configurations, this section describes a study of sensitivity to the memory write latency, the number of memory channels, the write buffer size and the epoch length. This section shows only the results for weighted speedup since the trends for throughput, fairness, lifetime and energy consumption are similar.

The write latency of the memory devices is varied to analyze the impact of the proposed schemes. The memory read latency is unchanged (i.e., 200 cycles). The memory write latency is varied from 1000 cycles (5x read latency) to 6000 cycles (30x read latency). Figure 14 shows the weighted speedup of UCP, WCP, UCP+WBB and WCP+WBB normalized to SHARE as the write latency varies. It only shows the average weighted speedup among 15 workloads. The improvement in weighted speedup decreases as memory writes become faster. This is expected as the importance of using writeback information is proportional to the write latency, which is the penalty of incurring a writeback when the memory bandwith is saturated. The weighted speedup improvement of WCP+WBB over UCP is 5.3% when the write latency is 1000 cycles, 10.6% for

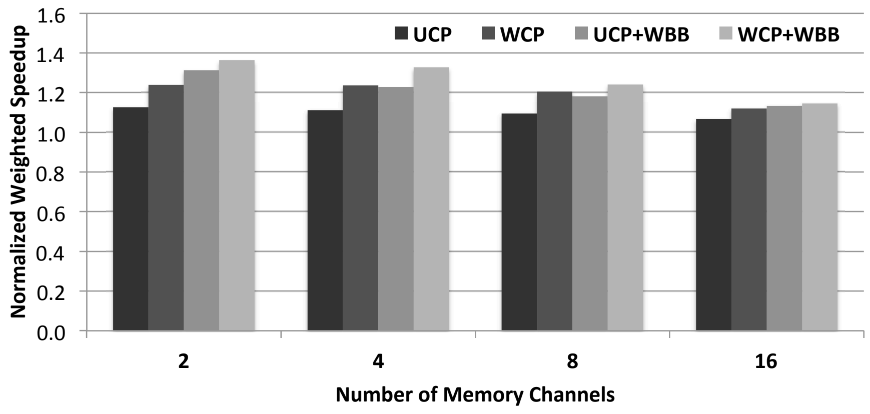


Figure 15: Sensitivity of WCP and WBB to # of memory channels

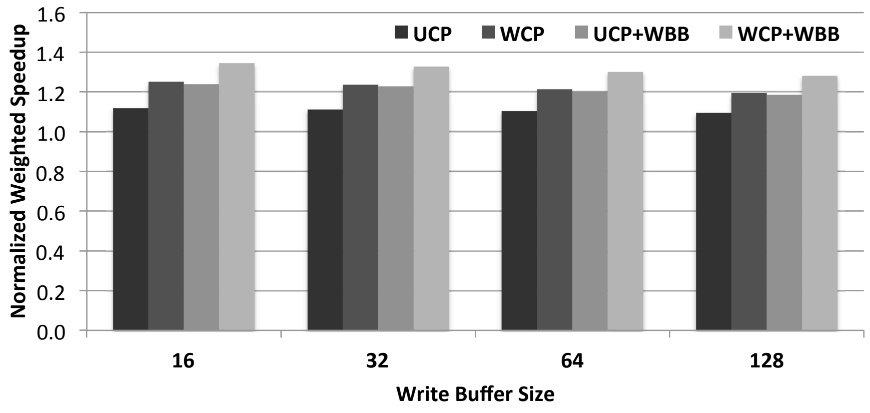


Figure 16: Sensitivity of WCP and WBB to write buffer size

2000 cycles, 19.3% for 4000 cycles and 21.1% for 6000 cycles.

Our baseline configuration assumes 4 memory channels. Figure 15 shows the weighted speedup impact as the number of memory channels is varied from 2 to 16. All policies benefit from more memory channels, as the number of write buffers is increased. The effectiveness of the proposed schemes reduces as the number of channels increases because more memory channels can naturally reduce the likelihood of congested write buffers. On average, WCP+WBB improves weighted speedup by 21.1%, 19.3%, 13.3% and 7.3% for 2, 4, 8 and 16 memory channels, respectively.

The baseline contains a 32-entry write buffer for each memory bank. Figure 16 shows the

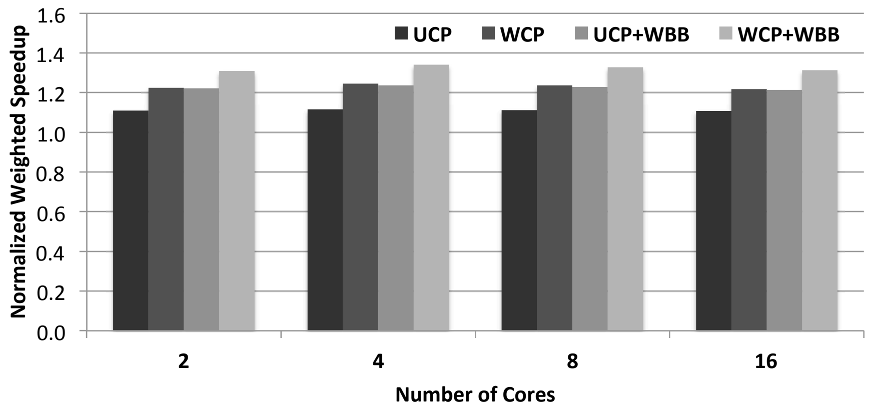


Figure 17: Sensitivity of WCP and WBB to # of cores

weighted speedup for UCP, WCP, UCP+WBB and WCP+WBB normalized to SHARE as the write buffer size is varied from 16 to 128. All policies benefit very little when the write buffer size increases, and the improvement of WCP+WBB over UCP varies slightly for various write buffer sizes. The average weighted speedup improvement of WCP+WBB compared to UCP is 20.2%, 19.3%, 17.9% and 17.1% for 16-, 32-, 64- and 128-entry write buffers.

A sensitivity study on epoch length by varying it from 1 million to 10 million cycles shows that the results are similar for different epoch lengths. The average weighted speedup improvement of WCP+WBB compared to UCP is 22.1%, 18.9%, 19.3% and 21.7% for 1, 2, 5 and 10 million cycles epoch length, respectively.

To understand the scalability of the proposed schemes, we evaluate WCP and WBB for a large number of workloads on 2-, 4-, 8- and 16-core systems with a shared DRAM cache of size 8MB, 16MB, 32MB and 64MB respectively. For each CMP configuration, we use 30 randomly generated workloads. WCP uses the lookahead algorithm [63] when scaled to systems with more than 4 cores. Figure 17 shows the summarized weighted speedup among all workloads. On average, WCP+WBB improves weighted speedup for 2-, 4-, 8- and 16-core systems by 18.1%, 20.1%, 19.3% and 18.5% over UCP respectively. These results show that WCP and WBB are scalable.

Component	Description	Storage Overhead
Shadow tag array per core	32 sets, 64 entries per set, each entry (V 1b, D 1b, TAG 21b, LRU 6b, WAD 6b)	8960 B
HIT counters per core	64 counters, 4B for each counter	256 B
AWB counters per core	64 counters, 4B for each counter	256 B
E-UMON per core	shadow tag array, HIT counters, AWB counters	9472 B
Total E-UMON overhead	8 E-UMONs	75776 B
WRB congestion time		4 B
Average WRB occupancy	4 memory channels, 8 WRBs per channel, 4B for each WRB	128 B
Sampled WRB occupancy	4 memory channels, 8 WRBs per channel, 4B for each WRB	128 B
Total MMON overhead		260 B
WCP	2048 sets, 64 tag entries per set, 3b core id in each tag entry	49152 B
Overall overhead		125188 B

Table 6: Storage overhead of WCP in a 8-core system

3.5.7 Hardware Overhead

E-UMON and WCP are the two largest sources of hardware overhead in our scheme. Table 6 details the storage overhead of E-UMON, MMON and WCP, assuming a 40-bit physical address space. Recall that our design of E-UMON only samples 32 sets. For the baseline 8-core configuration with 32MB shared LLC, our schemes require less than 0.4% storage overhead. None of the structures or operations required by WCP are on the critical path for cache hits. WBB introduces negligible latency overhead to cache replacement.

3.6 CONCLUSION

In this chapter, we study the intelligent cache partitioning and cache replacement techniques to better utilize the shared cache resource for asymmetric NVM systems. It is shown that LLC write-back information is important and should be taken into consideration. WCP reduces the number of overall LLC writebacks and LLC misses to improve the system performance. By distributing the slow memory write requests more evenly among write queues, WBB further improves the system performance.

4.0 BANDWIDTH PARTITIONING USING WRITEBACK INFORMATION

4.1 LIMITED MEMORY DEVICE BANDWIDTH

Non-volatile memory is a promising candidate for energy efficient memory due to its nonvolatility, scalability and low energy consumption. However, due to limited power, the memory access operations of non-volatile memory are usually slower than DRAM. For instance, PCM writes are 10-30 times slower than DRAM, while reads are 2-4 times slower [5]. As a result, non-volatile memory devices have extremely limited write bandwidth compared to DRAM.

With increasing core count in multi-core processors, performance is limited by the bandwidth of the off-chip bus and memory devices. For DRAM systems, the off-chip bus bandwidth is the bottleneck due to DRAM's relatively small access latency. Figure 18 shows device bandwidth utilization (percentage) for several SPEC CPU2006 benchmarks in DRAM and hybrid memory systems[1]. The bandwidth utilization is defined as the ratio of the consumed memory bandwidth to the available memory bandwidth. The figure plots the benchmarks in two groups with different y-axis scales (left: 0–5%, right: 0–40%). Each benchmark has two bars: the utilization for DRAM memory (first bar) and the utilization for hybrid memory (second bar). Each bar is divided into the amount of bandwidth consumed by reads (light shading) and writes (dark shading).

As the figure shows, for the DRAM system, all benchmarks, except *omnetpp*, have very small device bandwidth utilization ($< 2\%$). The low utilization indicates that device bandwidth for DRAM is not likely to be a system bottleneck. However, there are cases, illustrated by *omnetpp*, in which bandwidth utilization can be problematic. In these situations, memory reads typically consume the majority of the bandwidth due to the higher rate of read access. To address these problematic cases in DRAM main memory, Liu et al. propose off-chip bus bandwidth partition-

[1]Simulation details are described in Chapter 3.4.

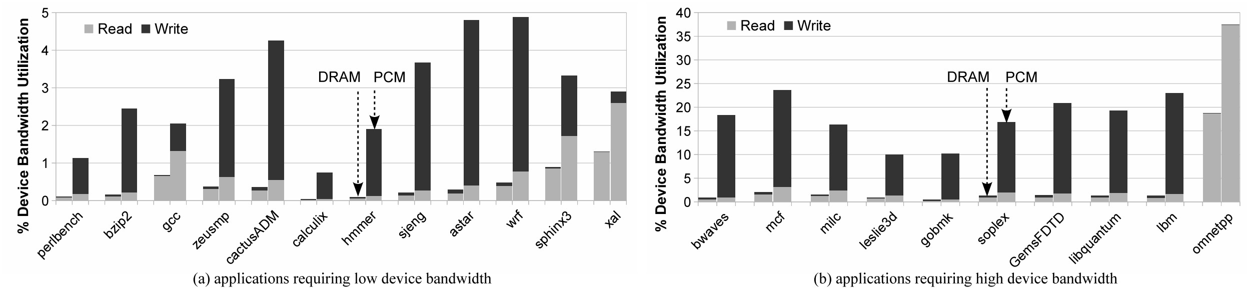


Figure 18: Percentage of device bandwidth for DRAM and hybrid memory systems

ing to maximize the overall weighted speedup with read information [47]. We call this class of technique "*Read-only Bandwidth Partitioning*" (RBP) as it takes into account only reads, ignoring LLC writebacks.

In hybrid main memory, the situation is different: the limited bandwidth available for write requests is a system bottleneck due to long write access latency. Figure 18 illustrates that device bandwidth utilization is much higher for PCM than DRAM. Although each benchmark consumes less than 37% of the device bandwidth, the overall bandwidth utilization of multiple applications in a multi-core system (i.e., 8-core) can be very high. This indicates that bandwidth is considerably more likely to be a system performance bottleneck, particularly as multi-core systems simultaneously execute multiple applications or threads. In addition, writes consume a substantial portion of device bandwidth. As a result, the device bandwidth consumed by writes is no longer negligible and must be taken into consideration in bandwidth partitioning.

To demonstrate that RBP is not always a good solution for hybrid memory, we evaluate the throughput of RBP [47] relative to the conventional sharing scheme in a 8-core baseline system for 15 workloads (see Figure 19). We characterize the workloads by the percentage of device bandwidth which are consumed by memory write requests (ranging from 10% to 90% from left to right). For workloads with a small portion of memory write traffic, RBP achieves good performance. For workloads with higher writeback traffic, however, RBP *hurts* performance due to inappropriate partitioning of the memory bandwidth using only read information. To better utilize the bandwidth, the writeback information is taken into consideration in partitioning decisions. For

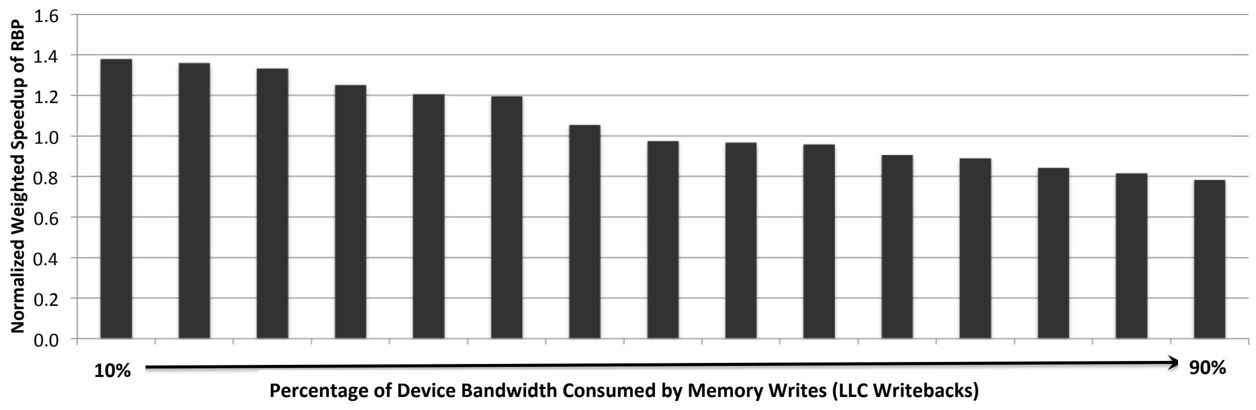


Figure 19: Throughput of RBP normalized to SHARE

the remainder of the thesis, "bandwidth" refers to the collective bandwidth of the memory devices.

4.2 ARCHITECTURE SUPPORT FOR WRITEBACK-AWARE BANDWIDTH PARTITIONING

4.2.1 Bandwidth Partitioning

Figure 20 shows an abstraction of a hybrid main memory architecture. DRAM is the LLC for hybrid main memory. Figure 20(a) shows the baseline memory architecture without bandwidth partitioning. Multiple cores share the bandwidth of the memory devices through the memory controller. Memory requests are served in First Come First Serve (FCFS) order.

Figure 20 (b) shows the baseline augmented with *bandwidth partitioning*. Instead of sharing bandwidth in a "free for all" fashion, bandwidth partitioning allocates and enforces specific shares (fractions) of the bandwidth for applications. For bandwidth partitioning, we use the token bucket algorithm [80].

From the token bucket algorithm, we derive an analytic model to determine the impact of bandwidth partitioning. The model establishes the relationship between token allocation and performance. The model uses the following terminology and assumptions. A *token* refers to a *memory*

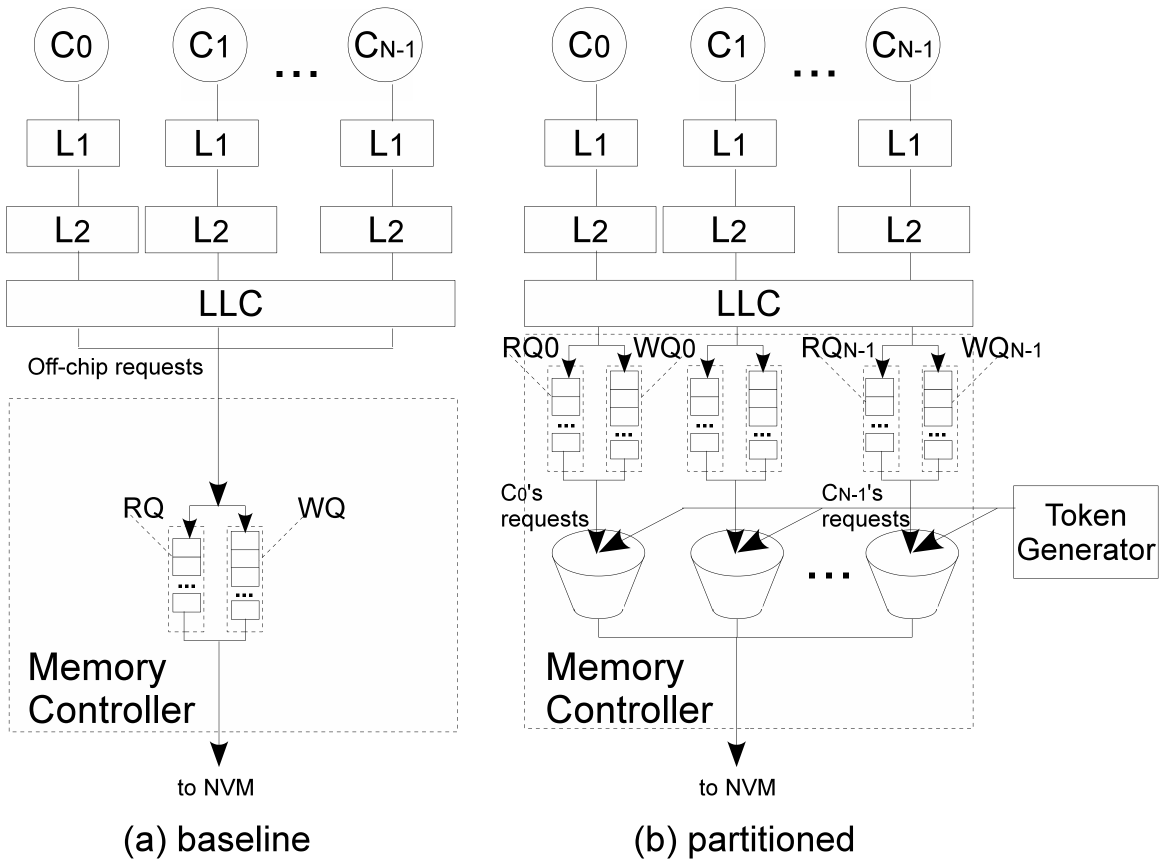


Figure 20: Architecture abstractions for bandwidth partitioning

device service cycle, that is, one service unit of the memory device. We use "device service cycle" and "token" interchangeably. Assuming read latency is d_r cycles and write latency is d_w ($d_w > d_r$) cycles for the non-volatile memory, each read request requires d_r tokens and each write request requires d_w tokens. A memory request from application i must acquire the necessary service tokens before it can be sent to the memory. Otherwise, pending requests are stored in either the *read queue* (RQ) or the *write queue* (WQ) of the corresponding core, depending on request type. The application stalls if a read request is pending. When a write request is pending, the corresponding application i continues unless there is no slot available in WQ_i. In other words, application i is delayed by pending read requests in RQ_i, and might be delayed by pending write requests when WQ_i is overwhelmed. Table 7 summarizes the notation.

System Parameters	
N	Number of cores
f_{clk}	CPU clock frequency of each core (Hz)
R	Number of memory ranks
K	Cache block size of the last-level cache ($Bytes$)
T	Peak device service token rate ($Cycles/sec$)
d_r	memory read latency (cycles)
d_w	memory write latency (cycles)
p	Probability that memory write stalls the computation

Application-specific Parameters for Application i	
M_i	LLC miss ratio
W_i	LLC writeback ratio
A_i	LLC access rate (sec^{-1})
$\lambda_{m,i}$	rate of LLC misses (sec^{-1}); $\lambda_{m,i} = M_i A_i$
$\lambda_{w,i}$	rate of LLC writebacks (sec^{-1}); $\lambda_{w,i} = W_i A_i$
$CPI_{LLC\infty,i}$	CPI with an infinite LLC size
$t_{m,i}$	Average LLC miss penalty ($Cycles$)
α_i	Bandwidth allocated to application i for read requests
β_i	Bandwidth allocated to application i for write requests
π_i	Number of cache ways allocated to application i

Table 7: Parameters for the analytic model

Bandwidth is measured by the amount of data that can be served in a unit of time [47], i.e., Bytes/sec. For DRAM memory, the bandwidth does not distinguish between read and write requests because the operations can be served at the same speed. However, a non-volatile memory device serves read requests faster than write requests. That is, a write request consumes more device service cycles than a read. As a result, in non-volatile memory, typically, the bandwidths for reads and writes are treated independently. Tokens are allocated to applications as a proxy for a share of the read and write bandwidth. The read and write token service rates are defined as the maximum rates at which reads and writes can be served. For hybrid main memory with R memory ranks, the total available token rate is $T = R \cdot f_{clk}$ tokens/sec, where f_{clk} is the CPU clock frequency.

The read and write service token rates are divided among applications to partition bandwidth. Let α_i and β_i denote the token rate allocated to application i for read and write requests, respectively. The bandwidth for application i can be expressed as $K \cdot (\frac{\alpha_i}{d_r} + \frac{\beta_i}{d_w})$ Bytes/sec, where each memory request size is K Bytes. The overall bandwidth can be expressed as $K \cdot \sum_{i=0}^{N-1}(\frac{\alpha_i}{d_r} + \frac{\beta_i}{d_w})$

Bytes/sec. Under the assumption that 100% of the service tokens are consumed, a partition of the bandwidth requires α_i and β_i to be determined, subject to

$$\sum_{i=0}^{N-1}(\alpha_i + \beta_i) = T, \tag{4.1}$$

where T is the peak memory device service token rate. Because memory read and write requests operate at different rates (i.e., $d_r \neq d_w$), the overall bandwidth for both request types can change. As a result, bandwidth partitioning must distribute tokens at rates proportional to the needs of each application.

By varying the fraction of the token rate allocated to application i, the application experiences delays due to pending requests in RQ_i and WQ_i to different degrees. Therefore, the token rate must be allocated wisely to ensure that the applications which can benefit the most receive more tokens.

The goal of bandwidth partitioning is to find a bandwidth allocation, $\bar{\alpha} = \{\alpha_0, ..., \alpha_{N-1}\}$ and $\bar{\beta} = \{\beta_0, ..., \beta_{N-1}\}$, to maximize the system *weighted speedup* (Equation (3.6)) under the constraint of Equation (4.1). It is known that cache partitioning improves performance by reducing cache misses and memory traffic [63, 93]. Bandwidth partitioning can further improve performance by allocating bandwidth intelligently among applications. In this chapter, it is assumed that the LLC is shared among cores. Coordinating bandwidth and cache partitioning is discussed in Chapter 5.

4.2.2 Architecture Overview

Due to the bursty nature of memory requests and the fact that memory traffic changes from time to time, it is natural to partition the bandwidth periodically in epochs. Figure 21 depicts our architecture for bandwidth partitioning. The architecture adds two components to the memory controller and two components to the Operating System (OS): (1) the *bandwidth utility monitor* (BUMon), (2) the *weight selection* logic, (3) the *bandwidth partitioning* logic, and (4) the *bandwidth regulator*. These components are shaded in the figure.

The bandwidth partitioning approach has four steps. Each step uses one of the architecture components from Figure 21. Figure 22 shows the steps for the k^{th} epoch on a timeline. First, BUMon tracks the number of executed instructions, cache misses and writebacks for the previous

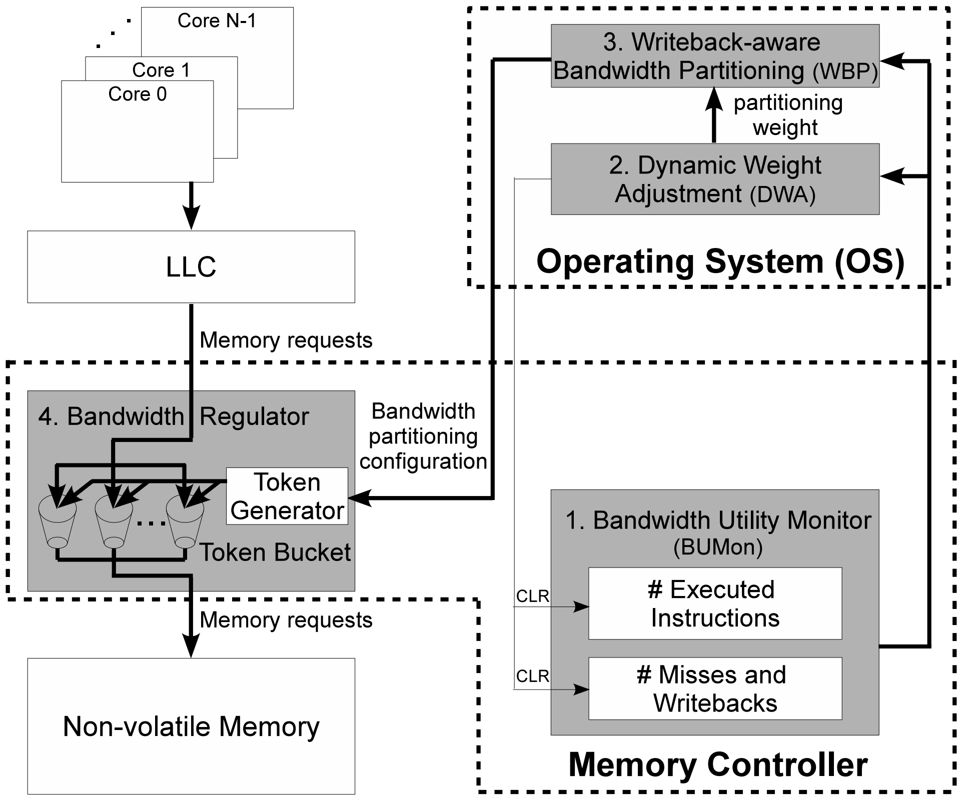


Figure 21: Components added to memory controller and OS to support WBP

epoch. Second, at the beginning of the k^{th} epoch, the weight selection component, DWA, uses the information collected by BUMon to select a *bandwidth partitioning weight* that reflects the relative importance of misses and writebacks to partitioning. Third, the bandwidth partitioning weight is fed into the bandwidth partitioning component, WBP, to allocate the bandwidth. Finally, the bandwidth partition is enforced by the bandwidth regulator which includes the *token generator* and one *token bucket* per core. The token generator distributes the tokens (memory device service cycles) among token buckets based on decisions from WBP. A memory request is ready for scheduling when the corresponding bucket has enough tokens to fulfill the request.

Figure 22 also shows how bandwidth allocation progresses in time and can be overlapped with workload execution. In the figure, each epoch lasts for t_e. DWA and WBP take Δt. Note that applications will not be interrupted or delayed during allocation. The allocation is computed con-

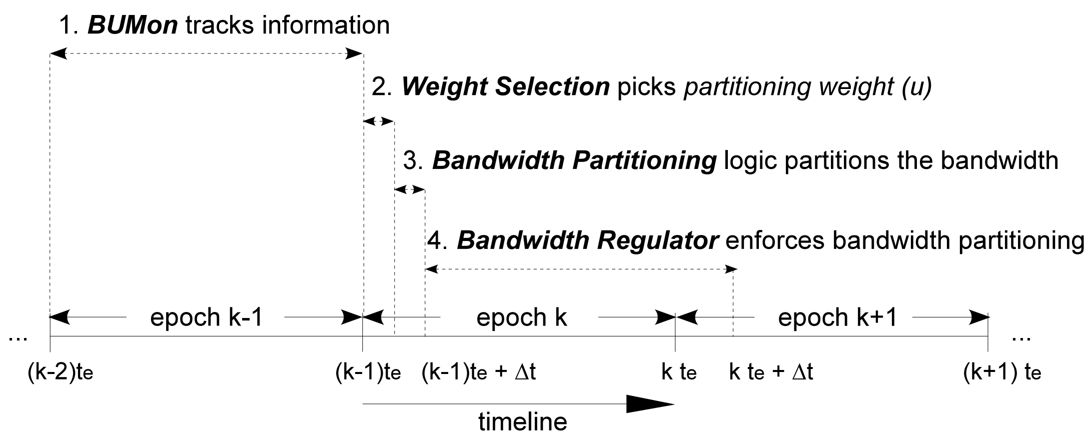

1. BUMon tracks information
2. Weight Selection picks partitioning weight (u)
3. Bandwidth Partitioning logic partitions the bandwidth
4. Bandwidth Regulator enforces bandwidth partitioning
epoch k-1
epoch k
epoch k+1
(k-2)te
(k-1)te
(k-1)te + Δt
k te
k te + Δt
(k+1) te
timeline

Figure 22: Writeback-aware bandwidth partition steps/timeline

currently with application execution by the OS and applied immediately once computed. However, the length of Δt impacts the responsiveness of bandwidth partitioning. For large Δt, the bandwidth has the potential to be allocated inappropriately due to stale application information (gathered by BUMon).

We experimentally examine the importance of responsiveness, as it is an indicator of the latency requirements for the OS to compute the bandwidth allocation. We measure the time to compute the partitioning configuration in the OS, which is less than $0.05t_e$ for $t_e = 5$M cycles. By varying the responsiveness (i.e., the length of Δt) from 0 to $0.1t_e$, the evaluation results show that the weighted speedup for the evaluated workloads differs by no more than 3% between $\Delta t = 0$ and $\Delta t = 0.1t_e$. Thus, implementing DWA and WBP in the OS avoids additional hardware cost without sacrificing the responsiveness and effectiveness of bandwidth partitioning.

4.3 WRITEBACK-AWARE BANDWIDTH PARTITIONING

In this chapter, an analytic model is proposed to derive the allocation strategy of WBP (see Figure 21) in three steps. First, we extend the additive CPI formula used in analyzing cache performance of uniprocessor systems [14, 49] to multi-core systems by considering contention on bandwidth. Second, RQ/WQ are modeled as queuing systems, in which contention results in queuing delay. Lastly, the weighted speedup maximization problem is solved using Lagrange multipliers.

4.3.1 CPI Formula Extension

The additive CPI formula [14, 49] expresses the average number of cycles to execute an instruction in a uniprocessor as the sum of the CPI assuming an infinite LLC ($CPI_{LLC\infty}$) and the extra CPI due to LLC misses ($h_m \cdot t_m$), where h_m is the average number of LLC cache misses per instruction, and t_m is the average LLC miss penalty.

$$CPI = CPI_{LLC\infty} + h_m t_m \tag{4.2}$$

Equation (4.2) is extended to multiple applications with separate pending queues for misses and writebacks. For application i, RQ_i and WQ_i can be viewed as independent single-server queuing systems. Therefore, the impact of bandwidth contention can be expressed as the extra queueing delay on these queues. However, the two queues impact applications differently. Every memory read request is on the critical path. Therefore, the queuing delay of RQ_i directly delays the application. On the other hand, memory write requests are on the critical path when write buffers are full or almost full. Qureshi et al. [61] propose write pausing to prioritize memory reads over writes when the occupancy of the write buffer is below a threshold (e.g., 85%). Otherwise, writes are serviced ahead of read requests. Memory writes are also on the critical path when WQ_i is overwhelmed. To derive a more accurate formula, it is assumed that memory writes are on the critical path with a probability p ($0 \leq p \leq 1$). Chapter 4.4 discusses estimating the effect of p.

Let $\Delta t_{m,i}$ and $\Delta t_{w,i}$ be the average expected queuing delay of the queues. The CPI for application i can be formulated as:

$$CPI_i = CPI_{LLC\infty,i} + h_{m,i}(t_{m,i} + \Delta t_{m,i}) + p\; h_{w,i}\Delta t_{w,i}\;, \tag{4.3}$$

where $h_{m,i}$ and $h_{w,i}$ are the average number of LLC misses and writebacks (i.e., memory reads and writes) per instruction. Let $\lambda_{m,i}$ and $\lambda_{w,i}$ denote the rate of LLC misses and writebacks as measured by the BUMon (see Table 7). Expression $h_{m,i}$ and $h_{w,i}$ can be expressed as:

$$h_{m,i} = \frac{\lambda_{m,i} CPI_i}{f_{clk}};\; h_{w,i} = \frac{\lambda_{w,i} CPI_i}{f_{clk}}. \tag{4.4}$$

By substituting $h_{m,i}$ and $h_{w,i}$ into Equation (4.3), the CPI of application i can be expressed as:

$$CPI_i = \frac{CPI_{LLC\infty,i}}{1 - \frac{\lambda_{m,i}}{f_{clk}}(t_{m,i} + \Delta t_{m,i}) - p\,\frac{\lambda_{w,i}}{f_{clk}}\Delta t_{w,i}}. \tag{4.5}$$

4.3.2 Queuing Delay Derivation

Little's law states that the average queue length is equal to the arrival rate multiplied by the system service time. The average waiting time of an request in the system is the total service time of the requests that are ahead of it in the queue.

For application i, the service time of a memory request on queues RQ_i and WQ_i are d_r/α_i and d_w/β_i. Hence, applying Little's law, the average queuing delay in RQ_i and WQ_i are:

$$\Delta t_{m,i} = \frac{\lambda_{m,i} d_r^2 f_{clk}}{\alpha_i^2};\ \Delta t_{w,i} = \frac{\lambda_{w,i} d_w^2 f_{clk}}{\beta_i^2}. \tag{4.6}$$

By substituting $\Delta t_{m,i}$ and $\Delta t_{w,i}$ into Equation (4.5), the CPI expression is derived as:

$$CPI_i = \frac{CPI_{LLC\infty,i}}{1 - \frac{\lambda_{m,i} t_{m,i}}{f_{clk}} - \frac{\lambda_{m,i}^2 \cdot d_r^2}{\alpha_i^2} - p\frac{\lambda_{w,i}^2 \cdot d_w^2}{\beta_i^2}}. \tag{4.7}$$

From the above discussion, it is possible to reduce the queuing delay of requests and, in turn, improve the IPC of application i by allocating more bandwidth to it (i.e., α_i and β_i). However, with limited overall bandwidth, it might be unable to allocate enough bandwidth to all applications. Instead, we try to partition the bandwidth efficiently among applications to achieve good overall system performance.

4.3.3 Maximizing Weighted Speedup

As explained in Chapter 4.2.1, the goal is to optimize weighted speedup, which by substituting Equation (4.7) into Equation (3.6) is expressed as:

$$\sum_{i=0}^{N-1} \frac{CPI_{alone,i}}{CPI_{LLC\infty,i}}\left(1 - \frac{\lambda_{m,i} t_{m,i}}{f_{clk}} - \frac{\lambda_{m,i}^2 d_r^2}{\alpha_i^2} - p\frac{\lambda_{w,i}^2 d_w^2}{\beta_i^2}\right). \tag{4.8}$$

Given that the first and second terms of expression (5.1) are not affected by bandwidth partitioning, minimizing the sum of the third and fourth terms is equivalent to maximizing the weighted speedup. To simplify the derivation, we assume that $\frac{CPI_{alone,0}}{CPI_{LLC\infty,0}} \approx ... \approx \frac{CPI_{alone,N-1}}{CPI_{LLC\infty,N-1}} \approx 1$. This approximation is reasonable because, in a system with N (e.g., $N = 8$) cores, the LLC capacity allocated to an application when it executes alone is N times larger than when it competes with others. In such a case, the miss rate and the CPI of an application by itself (executes alone) is close to that of the application running with an infinite size LLC. As a result, maximizing weighted speedup is equivalent to minimizing the function $F(\bar{\alpha}, \bar{\beta})$:

$$F(\bar{\alpha}, \bar{\beta}) = \sum_{i=0}^{N-1} \frac{\lambda_{m,i}^2 d_r^2}{\alpha_i^2} + p\frac{\lambda_{w,i}^2 d_w^2}{\beta_i^2}. \tag{4.9}$$

Minimizing $F(\bar{\alpha}, \bar{\beta})$ is a constrained optimization problem, with the objective function in Equation (4.9) and the constraints on α_i and β_i given by Equation (4.1). To solve this problem, we apply Lagrange multipliers by introducing a new variable (γ) and a Lagrange function:

$$L(\bar{\alpha}, \bar{\beta}, \gamma) = F(\bar{\alpha}, \bar{\beta}) + \gamma\big(\sum_{i=0}^{N-1}(\alpha_i + \beta_i) - T\big). \tag{4.10}$$

By differentiating $L(\bar{\alpha}, \bar{\beta}, \gamma)$ with respect to α_i, β_i and γ, and solving the differential equations, we arrive at the bandwidth allocation ($\bar{\alpha}$ and $\bar{\beta}$) to maximize weighted speedup for application i :

$$\alpha_i = \frac{T \cdot \lambda_{m,i}^{\frac{2}{3}} d_r^{\frac{2}{3}}}{\sum_{j=0}^{N-1}(\lambda_{m,i}^{\frac{2}{3}} d_r^{\frac{2}{3}} + w_b \lambda_{w,i}^{\frac{2}{3}} d_w^{\frac{2}{3}})}; \tag{4.11}$$

$$\beta_i = \frac{T \cdot w_b \lambda_{w,i}^{\frac{2}{3}} d_w^{\frac{2}{3}}}{\sum_{j=0}^{N-1}(\lambda_{m,i}^{\frac{2}{3}} d_r^{\frac{2}{3}} + w_b \lambda_{w,i}^{\frac{2}{3}} d_w^{\frac{2}{3}})}, \tag{4.12}$$

where $w_b = p^{\frac{1}{3}}$. w_b is the *bandwidth partitioning weight*. In the above equations, $\lambda_{m,i}$ and $\lambda_{w,i}$ are monitored by BUMon, while d_r and d_w are system parameters known at design time. w_b is the only unknown parameter.

To reflect the dependences of α_i and β_i on w_b, we use $\alpha_i(w_b)$ and $\beta_i(w_b)$ to denote the allocated tokens to read and write requests for application i. WBP (see Figure 21) selects a bandwidth partition that maximizes system performance by allocating $\alpha_i(w_b) + \beta_i(w_b)$ tokens to application i according to Equations (4.11) and (4.12). WBP computes these equations, given the monitored information and w_b. The next chapter discusses how to select the bandwidth partitioning weight (w_b).

4.4 DYNAMIC WEIGHT ADJUSTMENT

An issue for WBP is the difficulty of estimating the probability, p, that a write request will cause the computation to stall, and hence finding a good partitioning weight w_b statically. Different workloads prefer different weights. Consider two extreme scenarios. For applications with few write accesses, there are limited LLC writebacks, which are not enough to saturate the write buffers. In this case, the probability p that a writeback stalls the computation is zero, and hence $w_b = 0$. In this case, writeback-aware bandwidth partitioning behaves the same as read-only bandwidth partitioning [47] by ignoring memory writes. For applications with many write accesses, the writebacks cause applications to stall due to limited write bandwidth with a probability $p > 0$. More importantly, the probability p, and hence, the choice of weight w_b will change as applications go through phases. While the cache partitioning weight (w_c) indicates the performance impact of LLC writebacks in cache partitioning, the bandwidth partitioning weight (w_b) reflects how LLC writebacks influence the system performance in bandwidth partitioning.

A universal bandwidth partitioning weight does not exist as shown in Chapter 4.5.1. A bandwidth partitioning weight that works the best for one workload might be a bad choice for another workload. A good solution is to adjust w_b at run-time.

The weight w_b is adjusted dynamically at epoch boundaries. The weight selection module, DWA, picks a weight at the beginning of each epoch based on the information from BUMon. During the $(k-1)^{th}$ epoch, BUMon tracks the number of misses and writebacks for each application. At the end of the $(k-1)^{th}$ epoch, the weight selection module computes the rate of misses and writebacks ($\lambda_{m,i}^{k-1}$ and $\lambda_{w,i}^{k-1}$) which are used to compute three additional metrics: *consumed bandwidth* (CB), *instruction per device service token* (IPT) and *bandwidth utilization ratio* (BU). To better blend past and recent information, the counters for the number of executed instructions, misses and writebacks in BUMon are halved every epoch.

CB records bandwidth using device service tokens consumed by the application. The consumed bandwidth of application i in epoch k is:

$$CB_i^k = \lambda_{m,i}^k d_r + \lambda_{w,i}^k d_w. \tag{4.13}$$

IPT is a metric defined as the number of executed instructions divided by the consumed device service tokens. It reflects the bandwidth utility of an application. For application i in epoch k, the number of executed instructions per second is expressed as f_{clk}/CPI_i^k. Therefore, IPT could be expressed as:

$$IPT_i^k = \frac{f_{clk}}{CPI_i^k CB_i^k}. \tag{4.14}$$

An intelligent bandwidth partitioning policy that maximizes system performance should allocate the bandwidth such that the application with a higher bandwidth utility (i.e., IPT) receives a larger share of the memory bandwidth. For application i, the expected number of executed instructions is the product of its bandwidth utility and the allocated bandwidth, which could be expressed as $IPT_i^{k-1} \cdot \big(\alpha_i^k(w_b) + \beta_i^k(w_b)\big)$.

We try to find the weight w_b that maximizes the expected number of executed instructions (EEI) of the overall system, which can be expressed as

$$EEI^k(w_b) = \sum_{i=0}^{N-1} IPT_i^{k-1}\big(\alpha_i^k(w_b) + \beta_i^k(w_b)\big). \tag{4.15}$$

However, maximizing Equation (4.15) might lead to a pathological problem. Assume applications A_1 and A_2 compete for memory bandwidth. A_1 is allocated a large portion of the bandwidth while A_2 gets a small portion of the bandwidth in an epoch. A_1 has the potential of executing more instructions than A_2. Maximizing Equation (4.15) results in a larger bandwidth allocation to A_1 and a smaller allocation to A_2 in the next epoch. This feedback loop can have a detrimental effect, leading to the starvation of A_2 while A_1 is allocated more bandwidth than it can utilize. To resolve this problem, we consider how well an application uses its allocated bandwidth in the past. Specifically, the bandwidth utilization ratio, BU, is defined as the ratio of consumed bandwidth to allocated bandwidth. Given the allocated bandwidth for reads and writes in the k^{th} epoch (α_i^k and β_i^k), the bandwidth utilization ratio of application i in epoch k is expressed as

$$BU_i^k = \frac{CB_i^k}{\alpha_i^k + \beta_i^k}. \tag{4.16}$$

The feedback loop is prevented by bounding the bandwidth allocated to an application in epoch k by its allocation in epoch $k-1$, if the application did not effectively utilize the bandwidth in epoch $k-1$ (e.g., with a bandwidth utilization $< 85\%$). Moreover, even if the application had effectively utilized its allocated bandwidth in epoch $k-1$ (e.g., with a bandwidth utilization $\geq 85\%$), the increase in bandwidth allocation is limited to 20% to dampen the positive re-enforcement of the feedback loop. Specifically, we find w_b that maximizes

$$EEI^k(w_b) = \sum_{i=0}^{N-1} IPT_i^{k-1} min(\alpha_i^k(w_b) + \beta_i^k(w_b),\ s_i\, CB_i^{k-1}), \tag{4.17}$$

where $s_i = 1$ when $BU_i^{k-1} < 85\%$ and $s_i = 1.2$ otherwise.

At runtime, DWA selects the weight w_b from a candidate list $\{w_b^0, ..., w_b^{q-1}\}$. For each candidate w_b^i ($i = 0, ..., q-1$), DWA computes the expected number of executed instructions for the overall system $EEI^k(w_b^i)$. Finally, DWA selects the partitioning weight as the candidate that has the highest expected number of executed instructions. The partitioning weight is input to WBP to allocate the memory bandwidth.

4.5 EVALUATION RESULTS

To validate the effectiveness of our schemes, WBP is compared against the state-of-the-art bandwidth partitioning policy [47]. Sensitivity studies are also included. The experimental methodology and system parameters are the same as described in Chapter 3.4.

4.5.1 Performance

We compared the performance of several bandwidth allocation schemes to a non-partitioned baseline scheme (SHARE) to estimate the benefit of bandwidth partitioning. The proposed writeback-aware bandwidth partitioning with a partitioning weight of w_b is denoted as WBP_w_b. We show results for (1) read-only bandwidth partitioning (RBP, equivalent to WBP_0) [47], (2) WBP_0.5 that gives partial weights to memory writes, (3) WBP_1.0 that treats memory reads and writes with equal importance, and (4) WBP with dynamic weight adjustment (WBP+DWA), where for DWA $w_b \in \{0, 0.25, 0.5, 0.75, 1\}$. The partitioning schemes allocate bandwidth among applications at the beginning of each epoch which lasts 5 million cycles (t_e = 5M). Figure 23 shows weighted speedup normalized to the SHARE baseline for the workloads from Table 5.

On average, RBP outperforms the baseline by 6%. For workloads with high memory write traffic, RBP has worse performance than the baseline because memory writes consume a substan-

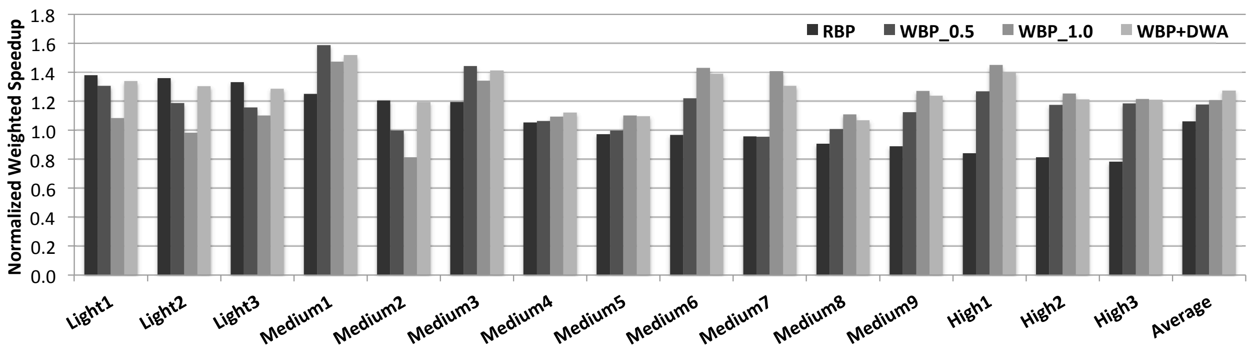


Figure 23: Weighted speedup normalized to SHARE

tial memory bandwidth (e.g., $> 59\%$ of the overall bandwidth) and RBP ignores the writeback information.

Among the policies with static partitioning weights (i.e., RBP, WBP_0.5 and WBP_1.0), each policy performs the best in one group of workloads. For *Light* workloads, memory reads consume a significant portion (i.e., $> 80\%$) of the memory device service cycles. For these workloads, memory writes should be ignored in bandwidth partitioning, and thus, RBP (WBP_0) is the best. On the other hand, WBP_0.5 is best for *Medium1* and *Medium3* because memory writes consume more bandwidth. Lastly, the memory bandwidth consumption due to writes accounts for more than 60% of the overall bandwidth for the *High* workloads. WBP_1.0 considers both memory reads and writes and gives them the same importance in the partitioning decision, and hence, it is the best choice.

These results demonstrate that there is no universal partitioning weight suitable for all workloads. By dynamically adapting the weight, WBP+DWA matches the performance of the best static partitioning weight w_b. Further examination of w_b over time reveals that WBP+DWA picks the best static partitioning weight more than 90% of the time. On average, WBP_0.5, WBP_1.0 and WBP+DWA improve weighted speedup by 11.1%, 14.0% and 20.1%, compared to RBP.

Figure 24 shows the impact on throughput. This metric has similar result as the weighted speedup. By avoiding interference among applications, RBP achieves an average of 6.5% throughput improvement over the baseline. WBP_1.0 improves the throughput by an average of 23.4%

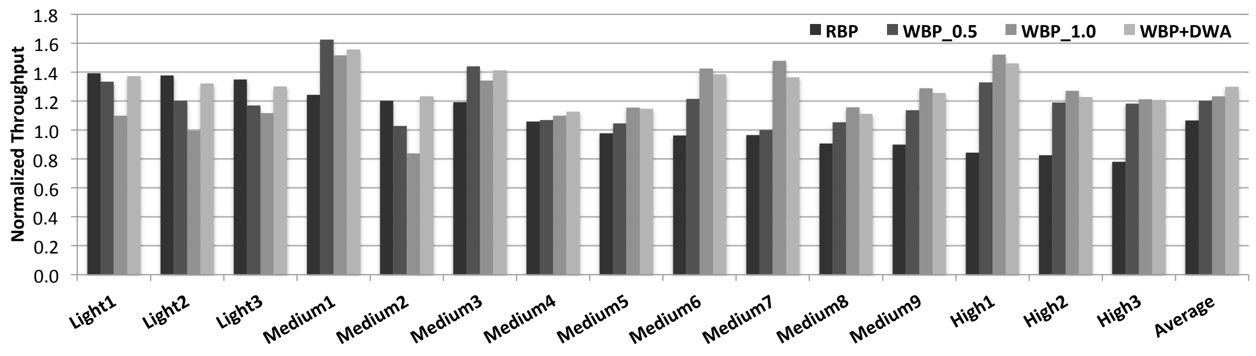


Figure 24: Throughput normalized to SHARE

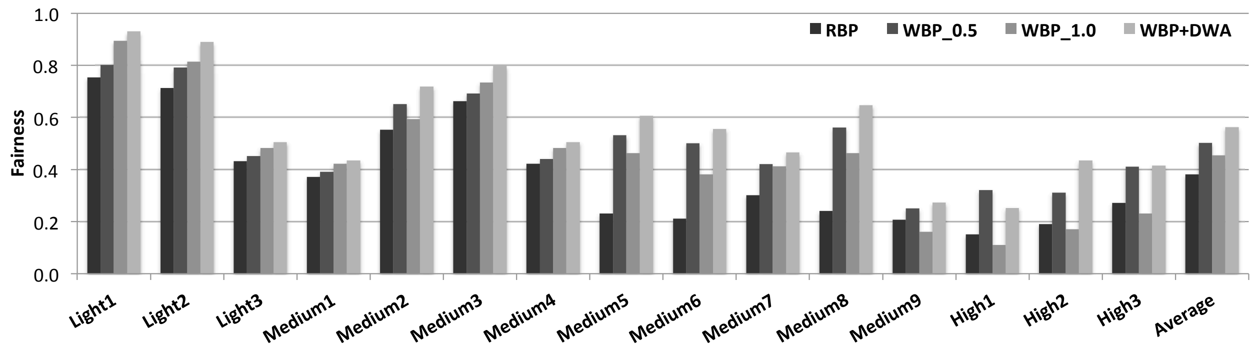


Figure 25: Fairness normalized to SHARE

over the baseline, and an average of 15.9% over RBP.

For seven of the fifteen workloads, WBP_1.0 improved throughput by more than 25% over RBP. WBP_1.0 does worse than RBP for four benchmarks. By selecting the partitioning weight dynamically, WBP+DWA improves throughput by 21.9% over RBP, and by 29.8% over the baseline on average.

Figure 25 shows the normalized performance of RBP, WBP_0.5, WBP_1.0, WBP+DWA in terms of fairness. On average, WBP+DWA improves the fairness metric by 47.8% compared to RBP.

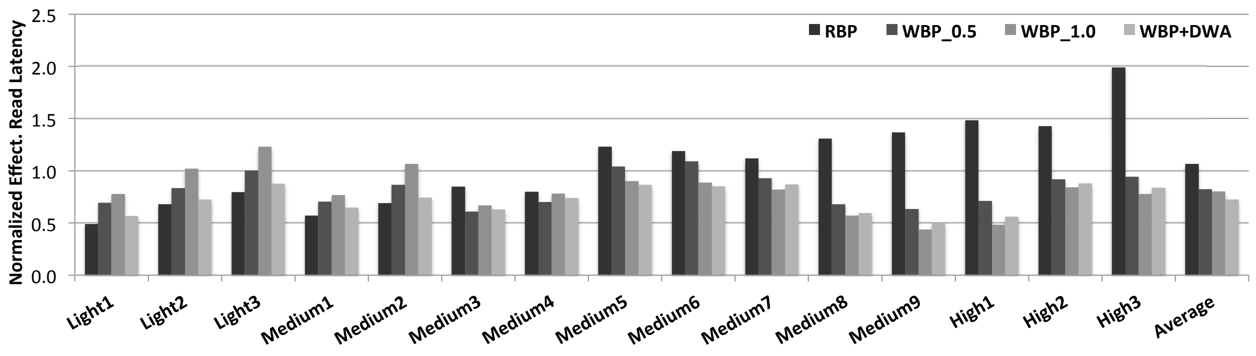


Figure 26: Average effective read latency normalized to SHARE

4.5.2 Effective Read Latency

Bandwidth partitioning impacts the memory request processing speed of an application by distributing available memory bandwidth among applications. It is evaluated how bandwidth partitioning influences memory latency. The *effective read latency* is defined as $t_{complete} - t_{arrive}$, where $t_{complete}$ and t_{finish} are the time when a read request arrives and completes.

Figure 26 shows average effective read latency for the bandwidth partitioning policies normalized to SHARE. For all workloads, WBP+DWA reduces the effective read latency by 32% over RBP, while the gains of the static WBP algorithms are slightly smaller. Our WBP algorithms reduce latency because they consider writebacks that are crucial for write-intensive workloads. By reducing the effective read latency, the system can process applications faster, and thus, improve overall performance (see Chapter 4.5.1).

4.5.3 Dynamic Weight Adaptation Granularity

Chapter 4.5.1 shows that WBP+DWA outperforms static WBP by selecting the cache partitioning weight w_b among $\{0, 0.25, 0.5, 0.75, 1\}$ dynamically. That is, DWA adapts $w_b \in [0, 1]$ with a step of 0.25. We examine how the step size influences performance. A large step is desirable as it reduces the number of iterations for the DWA algorithm. However, a small step may lead to a better allocation from a finer granularity of sharing. We consider several granularities: DWA_2

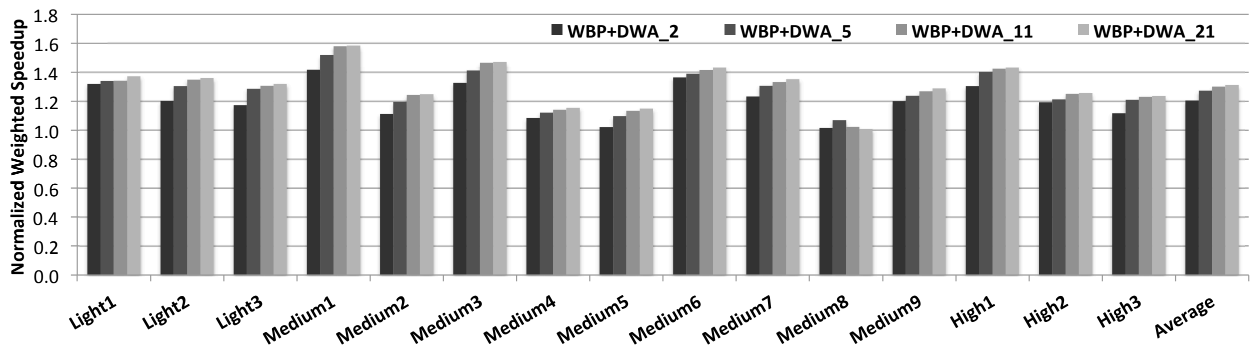


Figure 27: Performance impact of dynamic weight adaptation granularity

picks w_b from $\{0, 1\}$; DWA_5 selects w_b among $\{0, 0.25, 0.5, 0.75, 1\}$; DWA_11 adapts w_b among $\{0, 0.1, ..., 1\}$; and DWA_21 chooses w_b from $\{0, 0.05, ..., 1\}$.

Figure 27 shows the effect of weight adaptation granularity on the weighted speedup. On average, WBP+DWA_2, WBP+DWA_5, WBP+DWA_11 and WBP+DWA_21 improves the weighted speedup by 20.5%, 27.3%, 30% and 31%, respectively, over SHARED. As expected, a fine granularity (small step) for the candidate list leads to better performance. However, we consider DWA_5 to be the point of diminishing return.

4.5.4 Sensitivity Study

To analyze the benefit of the proposed schemes under different system configurations, we study the sensitivity of different schemes to memory write latency, the number of memory channels and the number of cores. We show only the results for weighted speedup since the results for throughput and fairness are similar.

The memory write latency is varied from 1000 cycles (5$\times$ read latency) to 6000 cycles (30$\times$ read latency). The read latency is unchanged (i.e., 200 cycles). Figure 28 shows the normalized weighted speedup (average over 15 workloads) of RBP, WBP_0.5, WBP_1.0 and WBP+DWA as the write latency varies. The benefit of our WBP schemes increase as memory writes become slower. This result is expected as the importance of writeback information is proportional to write latency, which is the penalty of incurring a writeback when the memory bandwidth is saturated.

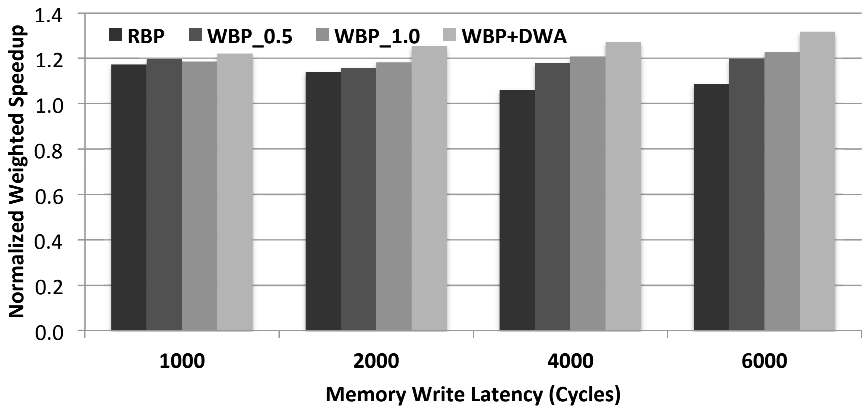


Figure 28: Sensitivity of bandwidth partitioning schemes to memory write latency

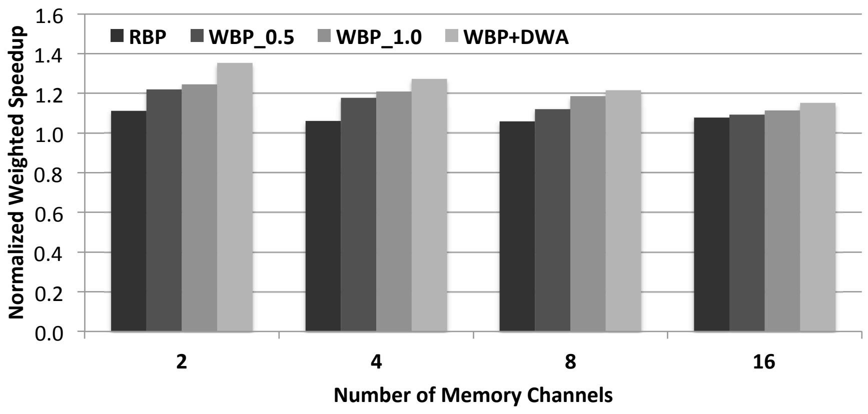


Figure 29: Sensitivity of bandwidth partitioning schemes to # of memory channels

Compared to RBP, WBP+DWA is 4.1%, 10%, 20.1% and 21.3% better when the write latency is 1000, 2000, 4000 and 6000 cycles, respectively.

The baseline assumes 4 channels with 8 memory banks per channel. Figure 29 shows the performance impact by varying the number of channels. The effectiveness of our schemes reduces with more channels; with more channels (more memory banks), there is more available bandwidth, which reduces the likelihood of bandwidth saturation. On average, WBP+DWA improved weighted speedup by 21.7%, 20.1%, 14.8% and 6.9% for 2, 4, 8 and 16 channels over RBP, respectively.

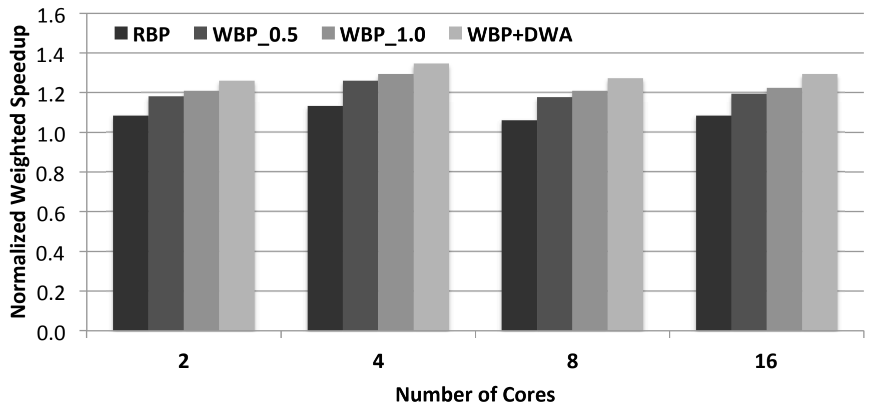


Figure 30: Sensitivity of bandwidth partitioning schemes to # of cores

To understand the scalability of our schemes, we evaluate them on 2-, 4-, 8- and 16-core systems with a hybrid memory of 1, 2, 4 and 8 memory channels, respectively. For each CMP configuration, we use 30 randomly generated workloads. On average, WBP+DWA improves weighted speedup for 2-, 4-, 8- and 16-core systems by 16.1%, 18.9%, 20.1% and 19.2% over RBP, respectively (see Figure 30).

WBP partitions the bandwidth at epochs which last for 5 million cycles. In order to understand the impact of epoch length, the epoch length is varied from 2 million to 10 million cycles. The results differ by less than 3% over these epoch lengths.

4.6 CONCLUSION

This chapter explores effective techniques to partition the memory bandwidth to achieve better system performance for asymmetric NVM systems. The LLC writeback information is shown to be important and beneficial to manage the memory bandwidth. By allocating the memory bandwidth based on an analytic model that considering the LLC writeback information, WBP outperforms the stat-of-the-art bandwidth partitioning techniques.

5.0 COORDINATING THE PARTITIONING OF CACHE AND BANDWIDTH

5.1 INTERACTION OF CACHE AND BANDWIDTH PARTITIONING

A well-known theory about the relationship between cache and bandwidth partitioning is that the performance impact of bandwidth partitioning is *secondary* to that of cache partitioning [76, 47]. The reason is that while cache partitioning can significantly reduce the total number of cache misses and writebacks, thus reducing the memory traffic, bandwidth partitioning cannot reduce the memory traffic. It seems natural to only consider the performance impact of cache partitioning while ignoring that of bandwidth partitioning. A closer examination of the interaction between cache and bandwidth partitioning, however, raises questions on such claim.

The management of the last-level cache and the memory bandwidth interact with each other. On the one hand, cache partitioning can reduce the total number of cache misses and writebacks, thus reducing the overall memory access traffic and alleviating the pressure on the memory system. On the other hand, although bandwidth partitioning does not directly impact the total number of cache misses and writebacks, it decides the speed of processing the off-chip memory requests for each application and, in turn, affects the rate at which each application accesses the cache. In a given time period, an application would benefit more from its cache partition by absorbing more cache misses and writebacks when the requests of the application are issued and processed at a higher rate. In other words, bandwidth partitioning affects the cache utility.

In order to demonstrate the interaction between cache and bandwidth partitioning, Figure 31 shows the throughput (sum of IPCs) of a workload, mcf-$bzip2$, under different static cache and bandwidth partition configurations on a 2-core system where one core runs *mcf* and the other core runs *bzip2*[2]. Not surprisingly, the system performance varies under different partition con-

[2]Simulation details are described in Chapter 3.4.

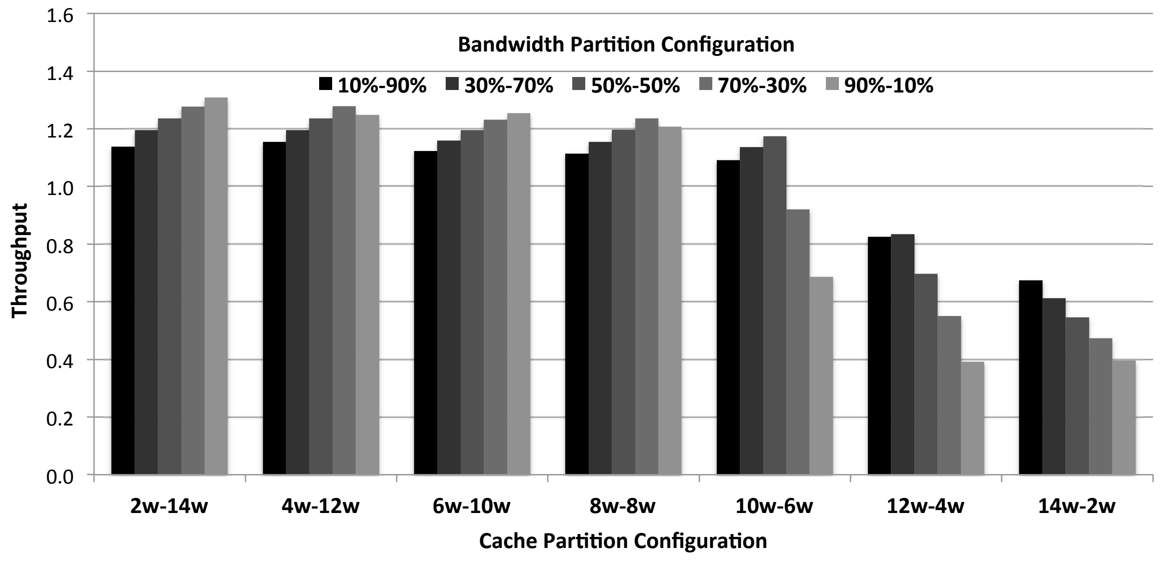


Figure 31: Performance of *mcf-bzip2*

figurations. The best performance is obtained when mcf and $bzip2$ get 90% and 10% share of the bandwidth (90%-10%), respectively, when 2 ways of the cache are allocated to mcf and 14 ways are used by $bzip2$ ($2w$-$14w$). Interestingly, the best choice of bandwidth partition changes under different cache partition configurations. The reason is that the off-chip memory traffic and bandwidth demand of each application changes under different cache sizes, which impacts the best bandwidth partitioning decision. This validates our reasoning that the choice of cache partitioning influences the bandwidth partitioning decision, and vice versa.

Given that the best static partition configuration for mcf-$bzip2$ is 90%-10% and $2w$-$14w$, a natural question would be whether other workloads favor the same partition configuration. Towards that end, I measured the performance of 150 randomly selected workloads under different static cache and bandwidth partition configurations on a 2-core system. Figure 32 shows the number of workloads that perform the best under particular static partition configurations. For instance, 9 workloads perform the best with a 10%-90% and $4w$-$12w$ configuration. The important point to retain is that most workloads benefit the most when the cache is partitioned in an unbalanced fashion. This is reasonable when one of the applications dominates the cache accesses, or when one application is cache friendly while the other is a streaming application. It is also clear that

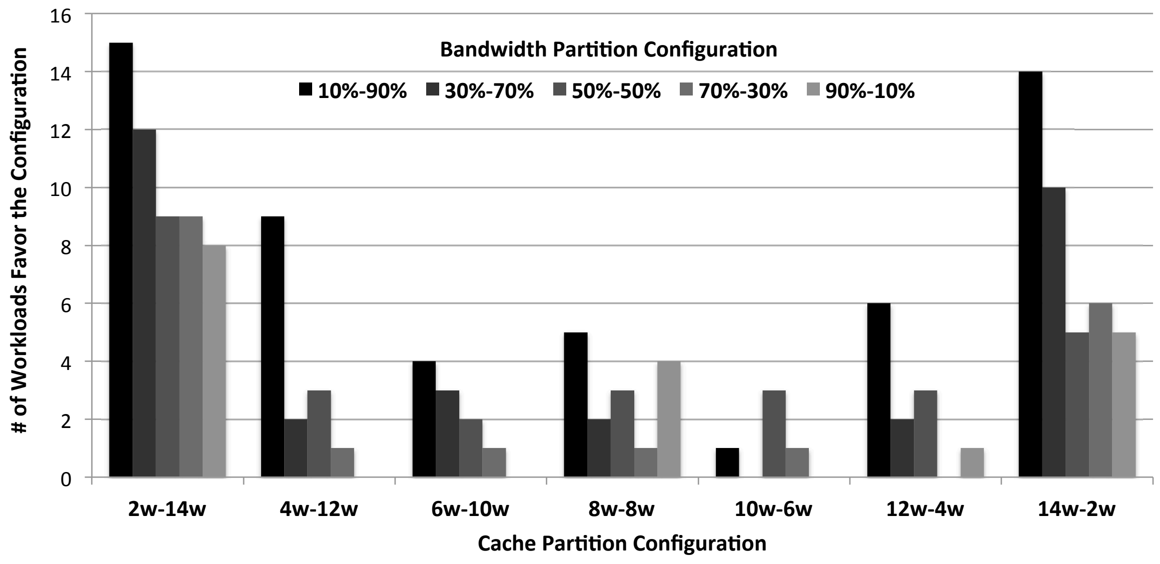


Figure 32: Number of workloads favoring each static partitioning configuration

different workloads favor different static partition configurations. As a result, a dynamic scheme is necessary to better partition the cache and bandwidth.

Unfortunately, simply adopting both WCP and WBP does not guarantee good system performance because doing so ignores the interaction between cache and bandwidth partitioning, and each of these schemes makes decisions based on different information. WCP manages the cache relying solely on information about cache misses and writebacks (i.e., HIT_c^i and AWB_c^i from E-UMON) without any knowledge about bandwidth allocation. Similarly, WBP allocates memory bandwidth based on information about the measured rates of LLC misses and writebacks of each application (i.e., $\lambda_{m,i}$ and $\lambda_{w,i}$ from BUMon) without any knowledge about the effect of changing the cache partition on $\lambda_{m,i}$ and $\lambda_{w,i}$.

Another reason is because WBP assumes that the memory traffic information, $\lambda_{m,i}$ and $\lambda_{w,i}$, does not change much between consecutive epochs, which is reasonable if the cache partition does not change. However, the assumption is no longer valid when both the cache and bandwidth are partitioned dynamically as proposed in this paper. Therefore, a new coordinated way of partitioning the cache and memory bandwidth is needed.

5.2 COORDINATED PARTITIONING OF CACHE AND BANDWIDTH

This section defines the problem of coordinated partitioning and introduces two approaches to solve it: (a) a search driven technique; (b) a scheme based on an analytic model.

5.2.1 Problem Definition

In a N-core system with M-ways of shared cache and memory bandwidth of T, the problem of *Coordinated Partitioning* is to find a cache partition $\bar{\pi} = \{\pi_0, ..., \pi_{N-1}\}$ and a bandwidth partition, $\bar{\alpha} = \{\alpha_0, ..., \alpha_{N-1}\}$ and $\bar{\beta} = \{\beta_0, ..., \beta_{N-1}\}$, to maximize the system weighted speedup (Equation (3.6)), under two constraints specified in Equations (3.3) and (4.1). Note that π_i is the number of cache ways allocated to application i, and α_i and β_i are the bandwidth allocation for memory reads and writes of application i.

By decoupling the additive CPI formula and modeling the memory system as queuing systems (see Section 4.3), the coordinated partitioning problem is transformed to the problem of minimizing a function $F(\bar{\pi}, \bar{\alpha}, \bar{\beta})$:

$$F(\bar{\pi}, \bar{\alpha}, \bar{\beta}) = \sum_{i=0}^{N-1} \left(\frac{(\lambda_{m,i}(\pi_i))^2 \ d_r^2}{\alpha_i^2} + p\frac{(\lambda_{w,i}(\pi_i))^2 \ d_w^2}{\beta_i^2}\right) \tag{5.1}$$

where d_r and d_w are latencies for memory reads and writes, $\lambda_{m,i}(\pi_i)$ and $\lambda_{w,i}(\pi_i)$ are the rates of LLC misses and writebacks for application i with an allocation of π_i cache ways, p is the probability that memory writes are on the critical path (the relative importance of writeback information), and α_i and β_i are the bandwidth allocation for memory reads and writes of application i, respectively. In the above equation, d_r and d_w are system parameters known at design time, while p can be determined dynamically at runtime (see Section 4.4). The rates of LLC misses and writebacks, $\lambda_{m,i}(\pi_i)$ and $\lambda_{w,i}(\pi_i)$, are the only unknown parameters. For presentation, we abbreviate these functions as $\lambda_{m,i}$ and $\lambda_{w,i}$, given a partition of π_i ways.

We can measure values of $\lambda_{m,i}$ and $\lambda_{w,i}$ periodically (in epochs) when executed with a cache partition of π_i ways, and use the measured values in one epoch to estimate their values for the next epoch. However, as mentioned above, this relies on an implicit assumption that the cache partition does not change from epoch to epoch.

Given a cache management scheme, bandwidth allocation can be applied as long as $\lambda_{m,i}$ and $\lambda_{w,i}$ reflect the rates of cache misses and writebacks for the incoming epoch. One way of coordinating cache partitioning and bandwidth allocation is to estimate $\lambda_{m,i}$ and $\lambda_{w,i}$ for different cache partition configurations, $\bar{\pi} = \{\pi_0, ..., \pi_{N-1}\}$, use the estimated values of $\lambda_{m,i}$ and $\lambda_{w,i}$ to derive the corresponding bandwidth allocations and then chose the cache partition and bandwidth allocation that maximizes Equation (5.1). The search method relies on runtime measurements to estimate the relation between $\lambda_{m,i}$, $\lambda_{w,i}$ and $\bar{\pi}$, and the model-driven method relies on an analytic approximation of $\lambda_{m,i}$ and $\lambda_{w,i}$ as functions of $\bar{\pi}$.

5.2.2 Search Driven Partitioning

This scheme optimizes Equation (5.1) by searching many different cache partition configurations. To estimate $\lambda_{m,i}$ and $\lambda_{w,i}$ for different cache partition configurations, we rely on monitoring the number of hits and avoidable writebacks on each cache way using the well known technique of emulating the LRU stack for each application [76, 63, 93]. Specifically, using a shadow tag array, we can record the number of additional hits (HIT_c^i) and avoidable writebacks (AWB_c^i) on way c for $0 \leq c \leq W - 1$. In other words, by allocating the cache way c to application i, we can increase the cache hits by HIT_c^i and reduce the cache writebacks by AWB_c^i. Note that the number of additional hits and avoidable writebacks for application i is still non-zero even when all W cache ways are allocated to application i. With such information, we can estimate $\lambda_{m,i}$ and $\lambda_{w,i}$ more accurately. Using the cache partition configuration $\{\pi_0, ..., \pi_{N-1}\}$ and the epoch length, $EpochLength$, we estimate the rate of LLC misses and writebacks:

$$\lambda_{m,i} = \sum_{c=\pi_i}^{M} HIT_c^i / EpochLength; \ \lambda_{w,i} = \sum_{c=\pi_i}^{M} AWB_c^i / EpochLength. \tag{5.2}$$

Algorithm 5.1 shows the search-driven approach to examine all possible cache partition configurations (Line 2). For each cache partition configuration, the algorithm computes $\lambda_{m,i}$ and $\lambda_{w,i}$ according to Equation (5.2) based on information from the shadow tag arrays (Line 3). It then calculates the bandwidth allocation by applying Lagrange multipliers to solve the minimization problem of function F (Equation (5.1)) (Line 4) and computes the value of function F (Line 5). Finally, after iterating over all possible cache partition configurations, the algorithm finds the

Algorithm 5.1 Search Driven Partitioning

```
1: $F_{min} \leftarrow MAX\_VALUE$
2: for every possible cache partition configuration $\bar{\pi}'$ do
3:     Compute $\lambda_{m,i}$ and $\lambda_{w,i}$ under partition $\bar{\pi}'$ according to Equation (5.2).
4:     Calculate bandwidth allocation $\bar{\alpha}'$, $\bar{\beta}'$ by minimizing function $F$ (Equation (5.1))
5:     Compute the value of function $F(\bar{\pi}', \bar{\alpha}', \bar{\beta}')$ using Equation (5.1)
6:     if $F(\bar{\pi}', \bar{\alpha}', \bar{\beta}') < F_{min}$ then
7:         $\bar{\pi} \leftarrow \bar{\pi}'$, $\bar{\alpha} \leftarrow \bar{\alpha}'$, $\bar{\beta} \leftarrow \bar{\beta}'$
8:         $F_{min} \leftarrow F(\bar{\pi}', \bar{\alpha}', \bar{\beta}')$
9:     end if
10: end for
```

cache partition and bandwidth allocation that minimizes function $F(\bar{\pi}, \bar{\alpha}, \bar{\beta})$, yielding the maximum weighted speedup (Lines 6-10).

A possible problem for the search-driven partitioning scheme is the long search process. The algorithm needs to compute the bandwidth allocation under all possible cache partition configurations. For example, for a 8-core system with 32-ways shared cache, it needs to iterates over more than 10 million cache partition configurations. The resource allocation decision will probably be obsolete after such a long computational delay.

To limit the search space, we start the search from the cache partition configuration used in the last epoch (epoch k-1), $\{\pi_{0,k-1}, ..., \pi_{N-1,k-1}\}$, with a limited search radius of r. A radius of r means SCRAM searches only partition configurations $\{\pi'_0, ..., \pi'_{N-1}\}$ when $|\pi'_i - \pi_{i,k-1}| \leq r$ for $0 \leq i \leq N-1$. In other words, the algorithm examines allocations that vary (up or down) up to r ways for each application. Section 5.3.1 details the latency overhead for various values of r.

5.2.3 Model Driven Partitioning

To avoid the long search process, we devise an analytic model that considers the performance impact of cache partitioning and bandwidth allocation together. Using the analytic model, we propose a novel coordinated scheme, *Unified Writeback-aware Partitioning* (UWP). In comparison to the search driven method, the trade-off is UWP relies on approximation and may not always find the best partitioning decision but it has much less overhead. We examine this trade-off in Section 5.3.1.

Hartstein et al. [21] predicted that the cache miss rate should vary with cache size as an inverse power law, and the exponent in the power law is (between -0.3 and -0.7) directly related to the time dependence of cache accesses. The writeback traffic is a proper subset of the cache misses, and the cache writeback rate should also follow an inverse power relation to the cache size. By reasonably approximating the cache miss and writeback rates, we avoid the search process. The rates of LLC misses and writebacks, $\lambda_{m,i}$ and $\lambda_{w,i}$, can be expressed as the product of the rate of LLC accesses, A_i, and the LLC miss and writeback rates, which can themselves be approximated as exponential functions of the number of dedicated cache ways:

$$\lambda_{m,i} = A_i u_i \pi_i^{s_i};\ \lambda_{w,i} = A_i v_i \pi_i^{t_i}. \tag{5.3}$$

Note that s_i, t_i, u_i and v_i are application-specific coefficients that are obtained through a curve fitting process. Substituting $\lambda_{m,i}$ and $\lambda_{w,i}$ in Equation (5.3) into the expression in Equation (4.9), function F can be expressed as:

$$F(\bar{\pi}, \bar{\alpha}, \bar{\beta}) = \sum_{i=0}^{N-1} f_i(\pi_i, \alpha_i, \beta_i), \tag{5.4}$$

where $f_i(\pi_i, \alpha_i, \beta_i)$ is defined as:

$$f_i(\pi_i, \alpha_i, \beta_i) = \frac{A_i^2 u_i^2 \pi_i^{2s_i} d_r^2}{\alpha_i^2} + p\frac{A_i^2 v_i^2 \pi_i^{2t_i} d_w^2}{\beta_i^2}. \tag{5.5}$$

As explained in Section 4.3.3, maximizing the weighted speedup is transformed to the problem of minimizing $F(\bar{\pi}, \bar{\alpha}, \bar{\beta})$, which is a constrained optimization problem, with the objective function in Equation (5.4) and the constraints on π_i, α_i and β_i given by Equations (3.3) and (4.1). To solve this problem, we apply Lagrange multipliers by introducing two new variables (γ_1, γ_2) and a Lagrange function:

$$L(\bar{\pi}, \bar{\alpha}, \bar{\beta}, \gamma_1, \gamma_2) = F(\bar{\pi}, \bar{\alpha}, \bar{\beta}) + \gamma_1\big(\sum_{i=0}^{N-1}(\alpha_i + \beta_i) - T\big) + \gamma_2\big(\sum_{i=0}^{N-1}\pi_i - M\big). \tag{5.6}$$

Because the problem of coordinated management involves way partitioning which is a discrete problem, and Lagrange multipliers is used to solve the maximization (minimization) problem of continuous functions, it is impossible to directly solve the coordinated management problem by Lagrange multipliers. Instead, we use Lagrange multipliers to derive the relations between way partitioning and bandwidth allocation decisions, and then find a near-optimal solution using a look-ahead algorithm. By differentiating $L(\bar{\pi}, \bar{\alpha}, \bar{\beta}, \gamma_1, \gamma_2)$ with respect to π_i, α_i, β_i, γ_1 and γ_2, and solving the differential equations, we can express the bandwidth allocation configuration as functions of the way partition configuration:

$$\alpha_i(\pi_i) = T \cdot g_i(\pi_i) / \sum_{j=0}^{N-1} \big((1 + h_j(\pi_j)) \cdot g_j(\pi_j)\big); \tag{5.7}$$

$$\beta_i(\pi_i) = T \cdot h_i(\pi_i) \cdot g_i(\pi_i) / \sum_{j=0}^{N-1} \big((1 + h_j(\pi_j)) \cdot g_j(\pi_j)\big), \tag{5.8}$$

where

$$g_i(\pi_i) = \sqrt{s_i A_i^2 \pi_i^{\frac{2}{3}s_i + \frac{4}{3}t_i - 1} u_i^{\frac{2}{3}} d_r^{\frac{2}{3}} v_i^{\frac{4}{3}} d_w^{\frac{4}{3}} p^{\frac{2}{3}} + t_i A_i^2 \pi_i^{2t_i - 1} v_i^2 d_w^2 p}, \tag{5.9}$$

$$h_i(\pi_i) = \pi_i^{\frac{2}{3}s_i - \frac{2}{3}t_i} u_i^{\frac{2}{3}} d_r^{\frac{2}{3}} v_i^{\frac{-2}{3}} d_w^{\frac{-2}{3}} p^{\frac{-1}{3}}. \tag{5.10}$$

In the above equations, d_r and d_w are system parameters known at design time, A_i can be monitored and estimated at runtime, while s_i, t_i, u_i and v_i are application-specific coefficients computed by fitting curves of the LLC miss and writeback rates into exponential curves of the number of cache ways. Similar to WBP, p is determined by maximizing the expected number of executed instructions (see Section 4.4 for details). However, the value of π_i remains unknown. Given Equations (5.7)-(5.10), cache partitioning remains an unsolved problem. A lookahead algorithm is introduced to solve the problem.

In Equations (5.7) and (5.8), the bandwidth allocated to an application could be expressed as a function of π_i, that is, $\alpha(\pi_i)$ and $\beta(\pi_i)$. By substituting α_i and β_i in Equations (5.7) and (5.8) into Equation (5.4), function F can be expressed as a function of only the cache partition $\bar{\pi} = \{\pi_0, ..., \pi_{N-1}\}$:

$$F(\bar{\pi}) = \sum_{i=0}^{N-1} f_i(\pi_i), \tag{5.11}$$

where $f_i(\pi_i)$ is derived by substituting α_i and β_i in Equations (5.7) and (5.8) into the expression in Equation (5.5).

The problem of minimizing $F(\bar{\pi})$ in Equation (5.11) could be translated into a problem of allocating a total of M ways among N applications to maximize the overall utility, where the utility of application i, U_i, is defined as the negative of the function $f_i(\pi_i)$ in Equation (5.5).

$$U_i(\pi_i) = -f_i(\pi_i). \tag{5.12}$$

Algorithm 5.2 Lookahead Cache Partitioning Algorithm

```
1:  for $i = 0$ to $N-1$ do
2:      $\pi[i] \leftarrow 1$
3:  end for
4:  while $M > \sum_{i=0}^{N-1} \pi[i]$ do
5:      $AppToAlloc \leftarrow 0$
6:      $WaysToAlloc \leftarrow 0$
7:      $MaxMarginalUtility \leftarrow MIN\_VALUE$
8:      for $i = 0$ to $N-1$ do
9:          for $\delta = 1$ to $M - \sum_{i=0}^{N-1} \pi[i]$ do
10:             Compute marginal utility $\Delta U_i(\pi[i], \delta)$
11:             if $\Delta U_i(\pi[i], \delta) > MaxMarginalUtility$ then
12:                 $AppToAlloc \leftarrow i$
13:                 $WaysToAlloc \leftarrow \delta$
14:                 $MaxMarginalUtility \leftarrow \Delta U_i(\pi[i], \delta)$
15:             end if
16:         end for
17:     end for
18:     $\pi[AppToAlloc] \leftarrow \pi[AppToAlloc] + WaysToAlloc$
19: end while
```

Marginal Utility is defined as the utility per unit allocation resource. Given that the utility for application i with π_i cache ways is $-f_i(\pi_i)$, the marginal utility of increasing the cache allocation from π_i to $\pi_i + \delta$ for application i is defined as:

$$\Delta U_i(\pi_i, \delta) = \frac{f_i(\pi_i) - f_i(\pi_i + \delta)}{\delta}. \tag{5.13}$$

While the classic greedy algorithm allocates one unit of resource at each step, a lookahead algorithm is able to find a better solution by enabling allocating more than one unit of resource if it achieves a higher marginal gain. Algorithm 5.2 shows the process of the lookahead algorithm to partition the cache among applications. First, the cache partition is initialized such that each application gets 1 allocated cache way (Lines 1-3), assuming that each application should use some cache. Then, the remaining cache ways will be allocated among applications iteratively (Lines 4-18). In each allocation iteration, the marginal utility of each application, $\Delta U_i(\pi[i], \delta)$, with different number of additional allocated cache ways, δ, is computed (Line 10). Among all the possible allocation decisions, the allocation decision with the highest marginal gain is chosen (Lines 8-17). The cache is allocated according to the allocation decision (Line 18). After the necessary iterations to allocate all the cache ways, it arrives at a cache partitioning decision $\{\pi[0], ..., \pi[N-1]\}$.

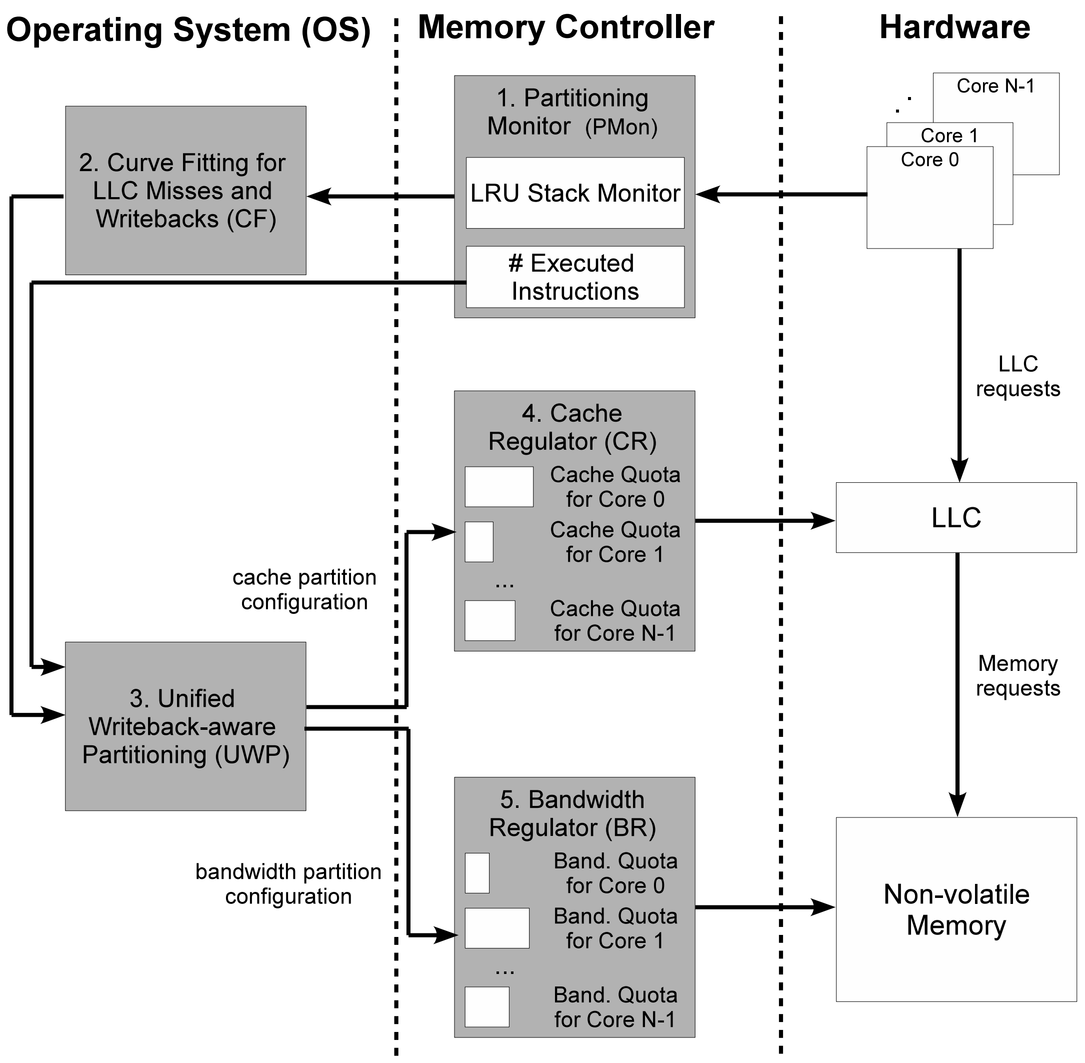


Figure 33: Architectural components to support UWP

After deciding on the cache partitions, the corresponding bandwidth partitioning decision can be computed by solving the expressions in Equations (5.7) and (5.8). Clearly the complexity of Algorithm 5.2 in UWP is smaller than that of Algorithm 5.1 in the search driven technique. The overhead of both techniques is evaluated experimentally in Section 5.2.2.

5.2.4 Architecture of UWP

Due to the bursty nature of memory requests and dynamic nature of memory traffic, it is natural to partition the cache and bandwidth periodically in epochs. Figure 33 depicts the necessary architectural components to support UWP. The architecture adds two components to the operating system (OS), namely CF (the logic of *curve fitting for LLC misses and writebacks*) and the UWP (*unified writeback-aware partitioning*) logic. Three components are added in the memory controller, namely PMon (the *partitioning monitor*), the *cache regulator* and the *bandwidth regulator*.

UWP operates in four steps. First, PMon collects information that is necessary to make partitioning decisions. More specifically, the *LRU stack monitor* tracks the cache miss and writeback information in the past epoch (epoch k-1) using shadow tag arrays and estimates the rates of cache misses and writebacks for epoch k. The number of executed instructions of each application in epoch k-1 is also monitored. Second, the CF logic approximates the miss and writeback rates in epoch k-1 as exponential functions of the allocated cache ways and produces the coefficients (s_i, t_i, u_i and v_i in Equations (5.9) and (5.10)). Third, the UWP determines the allocation of the cache capacity and the memory bandwidth for epoch k based on information from CF and PMon according to the unified analytic model. Lastly, the cache and bandwidth partition decisions for the epoch k are enforced by the cache regulator and bandwidth regulator, respectively.

Many of the architecture components of UWP have similar functionalities as those of WCP (Chapter 3) and WBP (Chapter 4). PMon monitors the system and collects information which is used by UWP to make partitioning decisions, while E-UMON and MMON (Chapter 3) gathers information for WCP and BUMon (Chapter 4) provides information for WBP. The UWP logic makes decisions to partition the last-level cache and the memory bandwidth, while the WCP logic (Chapter 3) computes the cache partition and the WBP logic (Chapter 4) allocates the memory bandwidth.

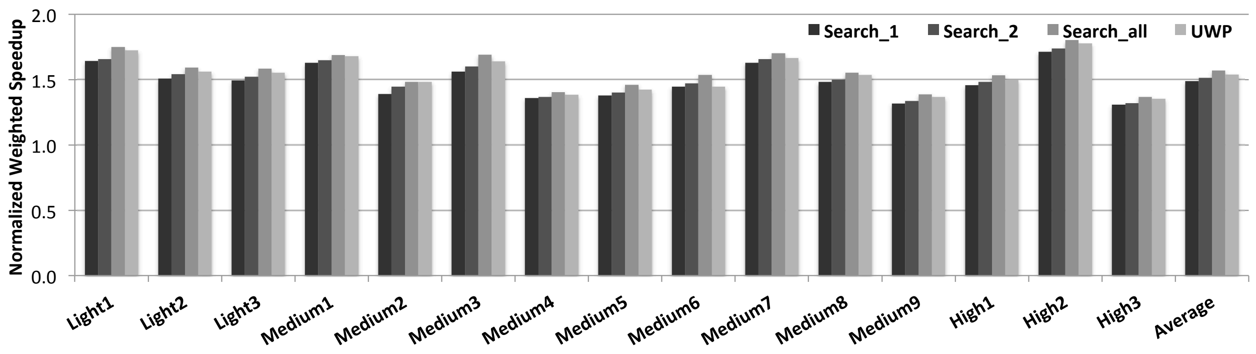


Figure 34: Weighted speedup of search driven partitioning and UWP normalized to SHARE

5.3 EVALUATION RESULTS

To validate the effectiveness of the proposed UWP scheme, UWP is compared against a non-partitioned baseline scheme (SHARE) that does not partition the cache nor the bandwidth, the cache and bandwidth partitioning schemes, namely WBP only (Chapter 4), WCP only (Chapter 3), and the combination of WBP and WCP together (WBP+WCP). The partitioning schemes allocate the cache and/or bandwidth at the beginning of each epoch, which lasts 5 million cycles. The partitioning decisions of cache capacity and/or memory bandwidth are enforced in the same manner as described in Chapters 3 and 4. Results are shown normalized to SHARE, unless otherwise noted. The experimental methodology and system parameters are the same as described in Section 3.4. Sensitivity studies with various memory read and write latencies, number of memory channels (between 2 and 16 channels) and number of cores are also performed in Section 5.3.5.

5.3.1 Search Driven Partitioning

Figure 34 shows the normalized weighted speedup of search driven partitioning with search radius of $r = 1$ (Search_1) and $r = 2$ (Search_2), an exhaustive search method (Search_all) and UWP. It is obvious that a larger search radius is favorable to improve the performance. On average, Search_1, Search_2 and Search_all improve the weighted speedup by 48.7%, 51.2% and 56.8% compared to SHARE. The intuition is that a larger search radius increases the probability of finding the best

Partitioning Scheme	# of Iterations	Time (epochs)
Search_1	$6,561$	2.6
Search_2	$390,625$	156.2
Search_all	> 4 billion	> 1 million
UWP	1	< 0.1

Table 8: Latency overhead of search driven schemes and UWP

cache and bandwidth partition configuration, and thus produces a better performance.

The proposed scheme, UWP, outperforms Search_1 and Search_2, but not Search_all. Clearly, the exhaustive search method performs the best, but it is not scalable. In a system with N cores, Search_r needs to iterate over $(2r+1)^N$ different cache partition configurations, because the cache partition for each core has $2r + 1$ possibilities. An exhaustive search method needs to test all possible cache partitions. For a 8-core system with 64-way cache, Search_all takes more than 4 billion iterations to compute the partition configuration.

The computational latencies of the search driven partitioning schemes and UWP are evaluated on a machine with a 4GHz processor. Table 8 lists the computational latencies of the search driven partitioning schemes and UWP for the baseline 8-core system. Search_1 and Search_2 take about 2.6 and 156.2 epochs, respectively, to calculate the partition configuration. It takes more than 1 million epochs for Search_all to make the partitioning decision. The long computational latencies of the search driven schemes are unaffordable because the partitioning decisions based on stale information are obsolete. Unlike the search driven schemes, UWP does not need to iterate over different cache partition configurations and can finish the computation fast enough (within 10% of the epoch). For the remaining evaluation, the thesis only shows the results for the feasible scheme, UWP.

5.3.2 Performance

Figure 35 shows weighted speedup of WBP, WCP, the combined application of WBP and WCP (WBP+WCP) and UWP normalized to the SHARE scheme for the workloads from Table 5. On average, WBP and WCP outperform the baseline by 27.3% and 32.7%, respectively. WCP outper-

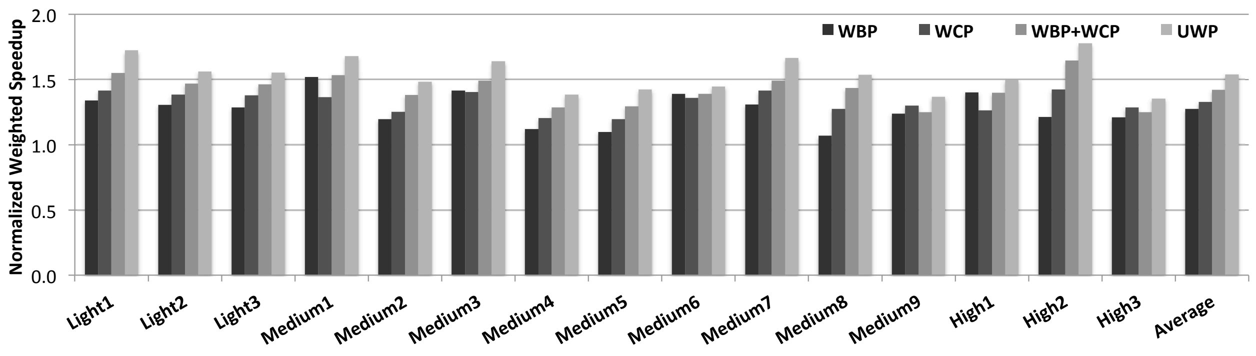


Figure 35: Weighted speedup of WBP, WCP, WBP+WCP and UWP normalized to SHARE

forms WBP for the *Light* workloads. For the *Light* workloads, WBP and WCP outperforms the non-partitioned scheme by an average of 30.9% and 39.3%, respectively. For the *Medium* and *High* workloads, WBP and WCP improve the weighted speedup by an average of 26.4% and 31.1%, respectively, over SHARE. Bandwidth partitioning by itself (WBP) can dramatically improves the system performance when the bandwidth is extremely limited, while it has insignificant effect when the bandwidth is not the system bottleneck. By partitioning both the cache and bandwidth, the combination of WBP and WCP can further improve the performance for most workloads. On average, WBP+WCP outperforms SHARE by 42.1%. However, note that WBP+WCP performs worse than WBP or WCP for *Medium9* and *High3*. The reason is that WBP and WCP may make decisions that negatively impact the effectiveness of each other. For instance, allocating very limited bandwidth to an application that gets a large cache partition results in inefficient utilization of the cache and bad overall system performance.

Unlike WBP+WCP, UWP considers the interaction of cache and bandwidth partitioning, relying on an unified analytic model. UWP performs better than WBP+WCP, WBP, and WCP for all workloads. On average, UWP outperforms WBP+WCP by 8.3%, WBP by 20.9%, WCP by 15.9% and SHARE by 53.9%, in terms of weighted speedup.

Figure 36 shows the throughput for different policies. By reducing the interference among applications through bandwidth partitioning and cache partitioning, WBP and WCP improve the throughput by an average of 29.4% and 35.3%, respectively, compared to SHARE. Comparing

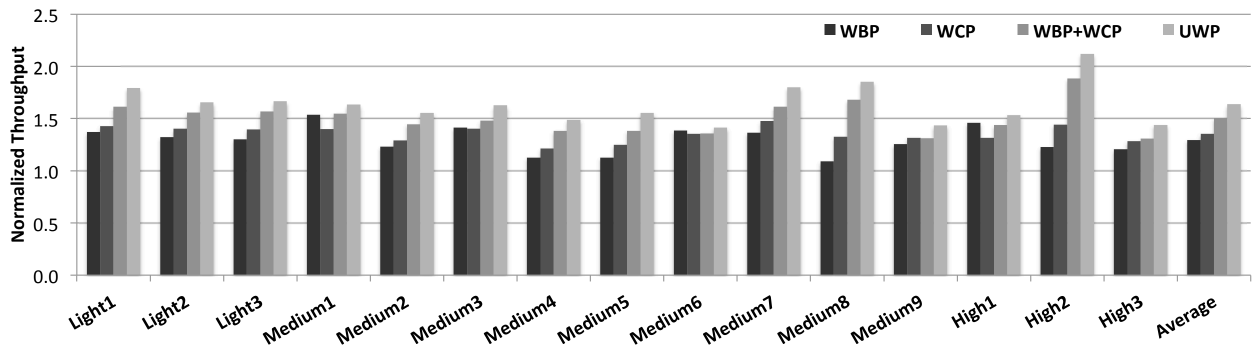


Figure 36: Throughput of WBP, WCP, WBP+WCP and UWP normalized to SHARE

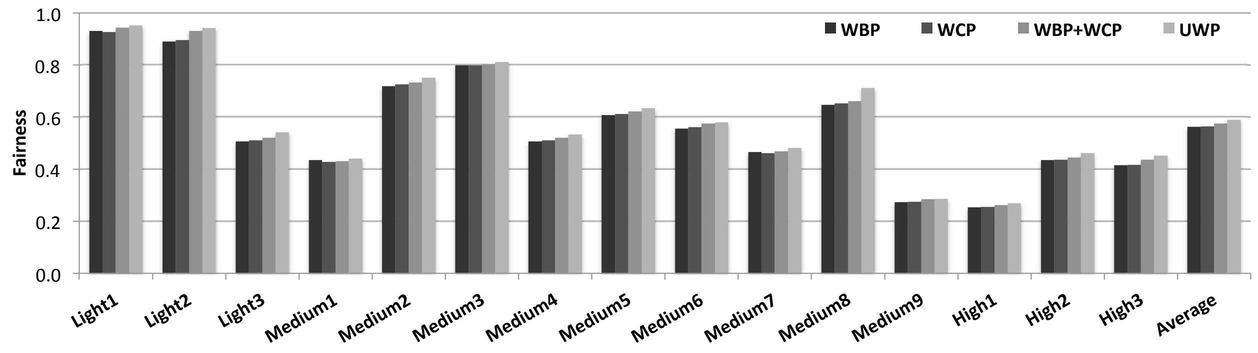


Figure 37: Fairness of WBP, WCP, WBP+WCP and UWP normalized to SHARE

to WBP and WCP, WBP+WCP benefits the system throughput for most workloads, similarly to the weighted speedup results (and due to the same reason). UWP manages cache and memory bandwidth cooperatively so that cache partitioning and bandwidth partitioning influence each other positively, resulting in better system performance. UWP improves the throughput by 8.8% on average over WBP+WCP.

Partitioning of the cache and bandwidth might improve the performance of some applications by severely penalizing others. The fairness metric is introduced to reflect whether the partitioning schemes benefit all applications uniformly. Figure 37 shows the performance of WBP, WCP, WBP+WCP and UWP in terms of fairness. On average, UWP improves the fairness metric by 2.4% compared to WBP+WCP.

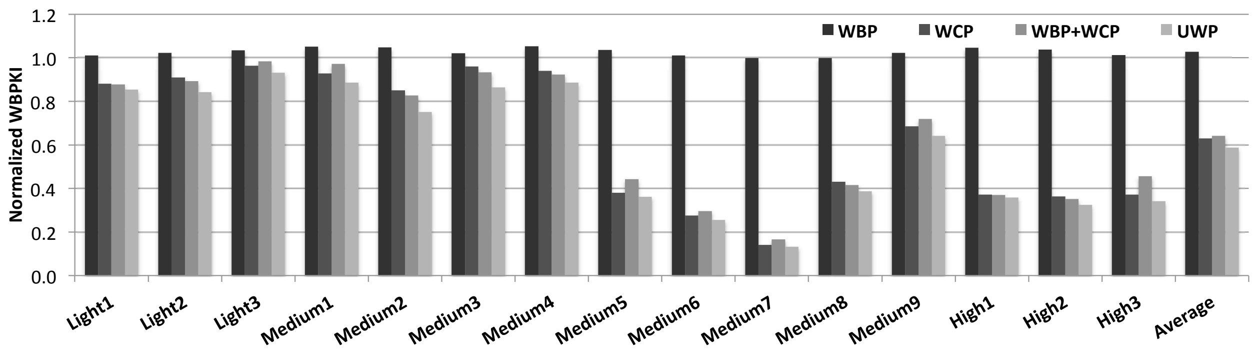


Figure 38: WBPKI of WBP, WCP, WBP+WCP and UWP normalized to SHARE

5.3.3 Memory Lifetime

Given that non-volatile memory has limited endurance, the number of memory writes (due to cache writebacks) is an important metric to evaluate. Assuming that a perfect wear-leveling scheme [65, 18, 11] is applied, the reduction in writebacks directly translates into memory lifetime extension. Figure 38 shows the writebacks per 1K instructions (WBPKI) of different policies. WBP has little impact on WBPKI. This is intuitive because bandwidth partitioning only influences the processing speed of memory requests, and has insignificant impact on the number of cache writebacks. Unlike WBP, WCP influences WBPKI by allocating the cache among applications. By minimizing a weighted sum of the number of cache misses and the number of writebacks, WCP reduces WBPKI by 37% over the non-partitioned scheme. For the workloads with high memory traffic, the *High* workloads, WCP achieves a reduction of 63% on WBPKI.

On average, UWP reduces the WBPKI by 8.5%, resulting in an memory lifetime extension of 9.2%, compared to WBP+WCP. While bandwidth partitioning cannot impact the memory traffic directly, it influences the effectiveness of cache partitioning and, in turn, indirectly affects the number of memory writes. By positively influencing cache partitioning through better bandwidth partitioning decisions, UWP reduces the WBPKI over WBP+WCP.

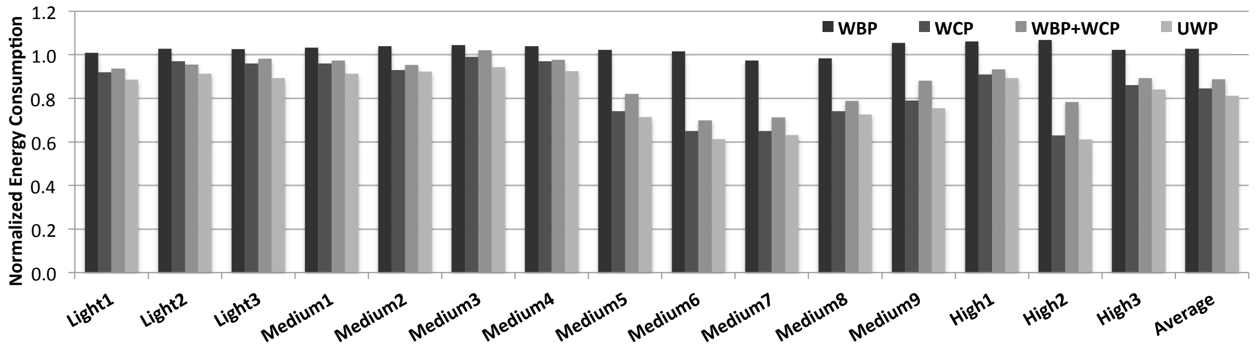


Figure 39: Energy consumption of WBP, WCP, WBP+WCP and UWP normalized to SHARE

5.3.4 Energy Consumption

Figure 39 shows the memory energy consumption of WBP, WCP, WBP+WCP and UWP normalized to SHARE. The memory energy consumption is defined as the sum of the energy consumed by memory reads and the energy consumed by memory writes. The memory energy consumption results follow the same pattern as that of WBPKI because memory writes are more energy hungry than reads and dominate the energy consumption. Overall, SHARE and WBP have similar energy consumption. WCP saves energy by reducing cache writebacks (i.e., memory writes) over SHARE and WBP. On average, WCP, WBP+WCP and UWP reduce energy by 15.5%, 11.3% and 18.9% over SHARE, respectively. Note that the results are conservative as UWP can also improve the energy consumption of other system components (e.g., the cores and the DRAM LLC) by reducing the processing time of applications.

5.3.5 Sensitivity Study

To analyze the benefit of UWP under different system configurations, this section studies the sensitivity of the proposed schemes to memory read and write latencies, the number of memory channels and the number of cores. Only the results for weighted speedup are shown for brevity. The trends for throughput, fairness, lifetime and energy are similar.

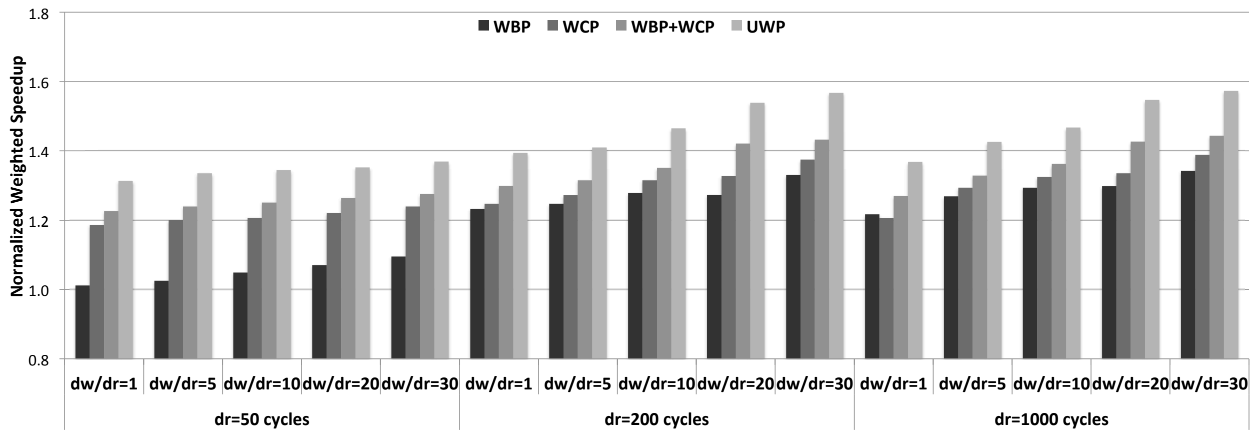


Figure 40: Sensitivity of UWP to memory write latency

Different NVM technologies finish read and write requests at different speeds. Typically, STT-RAM processes reads in 30ns and writes in 40ns [40], a typical PCM device has a read latency of 50ns and a write latency of 1μs [93, 94], while a ReRAM device services read request in 2μs and write request in 10μs [67]. To understand whether UWP benefits different NVM technologies, we vary the memory read and write latencies. The read latency (d_r) is varied between 50 cycles and 1000 cycles, and the ratio of write latency to read latency (d_w/d_r) is varied between 1 and 30. Figure 40 shows the normalized weighted speedup (average over all 15 workloads) of WBP, WCP, WBP+WCP and UWP under different memory read and write latencies. WBP failed to achieve comparable performance to WCP when the read latency is fast ($d_r = 50$ cycles). The reason is that, for memory systems with relative fast access speed, the bandwidth bottleneck is the bus, not the bank bandwidth.

The benefit of WCP, compared to SHARE, increases as memory writes become slower. This result is expected as the importance of WCP is proportional to the write latency, which is the penalty of incurring a writeback when the memory is saturated. The benefit of UWP over all schemes, including WBP+WCP, is stable across different memory access latencies. This shows that coordinated management is beneficial for different NVM technologies.

The baseline assumes 4 channels with 8 banks per channel. Figure 41 shows the performance impact by varying the number of channels while keeping the same number of banks per channel,

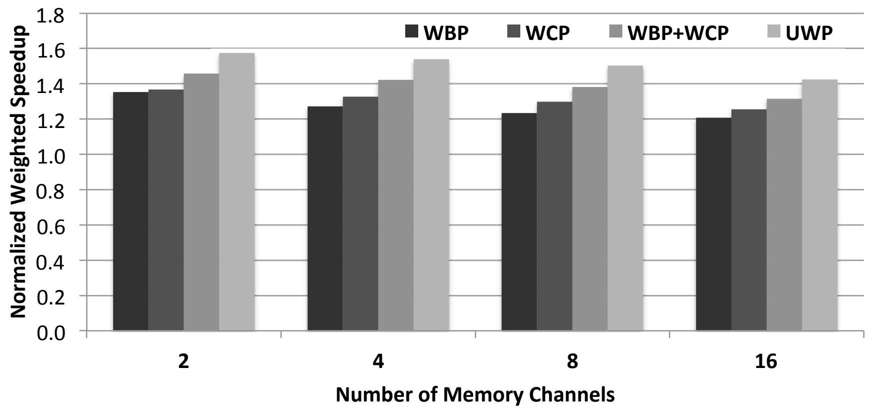


Figure 41: Sensitivity of UWP to number of memory channels

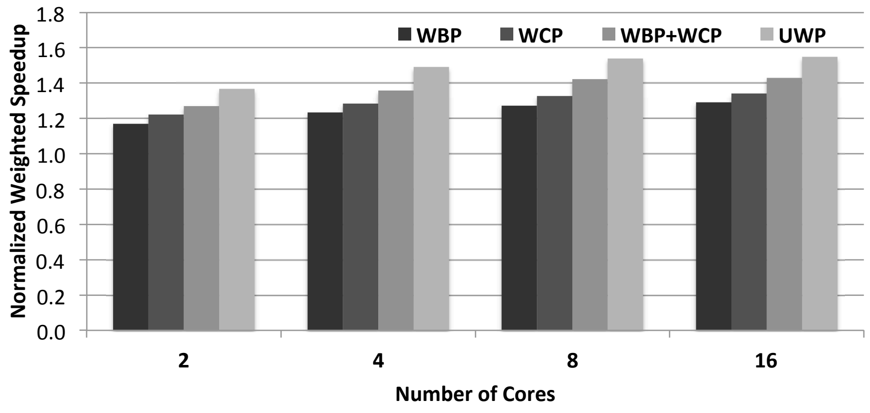


Figure 42: Scalability of UWP

that is, varying the total memory bandwidth. The benefit of WBP, compared to SHARE, decreases as the number of memory channels increases. This is because the importance of bandwidth partitioning is proportional to the scarceness of memory bandwidth, which naturally decreases as the system has more memory channels (banks). However, the effectiveness of UWP over WBP+WCP is unchanged under different number of memory channels. On average, UWP improves weighted speedup by 8.1%, 8.3%, 8.9% and 8.2% for 2, 4, 8 and 16 channels over WBP+WCP, respectively.

To understand the scalability of the proposed scheme, UWP is evaluated on 2-, 4-, 8- and 16-core systems with a shared DRAM LLC of size 8MB, 16MB, 32MB and 64MB and a hybrid memory of 1, 2, 4 and 8 memory channels, respectively. For each CMP configuration, UWP is evaluated using 30 randomly generated workloads. Figure 42 shows the summarized weighted speedup among all workloads. On average, UWP improves weighted speedup for 2-, 4-, 8- and 16-core systems by 7.7%, 9.8%, 8.3% and 8.2% over WBP+WCP, respectively. These results show that UWP is scalable.

5.4 CONCLUSION

In this chapter, we experimentally show that cache partitioning and bandwidth partitioning are dependable on each other. The partitioning decision on one resource impacts the effectiveness of partitioning the other resource. Based on such observation, we propose a coordinated analytic model the consider the performance impact of both cache partitioning and bandwidth partitioning. UWP is proposed to partition the cache and memory bandwidth cooperatively, improving the system performance effectively.

6.0 CONCLUSION

Non-volatile memory technologies (NVM) are promising candidates for energy-efficient main memory due to their promising features such as non-volatility, good scalability and energy efficiency. Phase Change Memory (PCM) is one of the NVMs that has been proposed as a replacement for (or in addition to) DRAM by many researchers. Unlike DRAM, most NVMs have a unique read/write *asymmetry* such that the write operations take longer time and consume higher energy/power than memory reads. Existing resource partitioning techniques have been shown to be beneficial for the system performance by reducing the interferences between competing applications. However, these techniques are designed for symmetric memory (e.g., DRAM) and fail to take into consideration the read/write asymmetry property of NVM. In this dissertation, we study shared resource management schemes that are tailored to non-volatile asymmetric memory by considering NVM's asymmetric property.

The last-level cache (LLC) and memory bandwidth are two important *shared* resources in the memory hierarchy. First, we propose a *Writeback-aware Cache Partitioning* (WCP) scheme to better partition the LLC among applications. Under the observation that both memory writes (due to LLC writebacks) and memory reads (due to LLC misses) impact the system performance, WCP improves the overall system performance by reducing the LLC misses as well as LLC writebacks.

Second, we use an analytic model to study the management of memory bandwidth in NVM systems. Based on the analytic model, we propose a *Writeback-aware Bandwidth Partitioning* (WBP) scheme to allocate the memory service cycles among applications. Comparing to existing techniques such as Utility-based Cache Partitioning (UCP), WBP is more suitable and achieves better performance for NVM systems by (a) allocating the *bank bandwidth* rather than the bus bandwidth; and (b) considering the bandwidth consumption by the memory write requests.

Finally, we study the interactions between cache partitioning and bandwidth partitioning. We validate the bidirectional relationship between cache and bandwidth partitioning such that the management policy of one shared resource affects the effectiveness of the other shared resource management scheme. A *coordinated* analytic model is proposed to study the performance impacts of cache partitioning coordinated with bandwidth partitioning. We propose a *Unified Writeback-aware Partitioning* (UWP) to partition the last-level cache and the memory bandwidth cooperatively. UWP can further improve the system performance by considering the interaction of cache partitioning and bandwidth partitioning.

NVM technologies provide an alternative solution for main memory to achieve higher energy efficiency. The undesirable characteristics, such as slow and energy hungry writes and low write bandwidth, might hinder the application of NVMs as main memory. The three techniques proposed in this thesis are shown to improve the system performance through better management of the shared resources (i.e., LLC and memory bandwidth). The proposed techniques partially alleviate the negative impacts of NVM's undesirable characteristics. More architectural techniques are still needed to make NVM a viable solution for energy-efficient main memory.

BIBLIOGRAPHY

[1] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1), 2009.

[2] Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO '08*, 2008.

[3] Mainak Chaudhuri. Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. In *MICRO '09*, 2009.

[4] Feng Chen, Song Jiang, and Xiaodong Zhang. Smartsaver: turning flash drive into a disk energy saver for mobile computers. In *ISLPED '06*, 2006.

[5] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *CIDR '11*, 2011.

[6] YC Chen, CT Rettner, S Raoux, GW Burr, SH Chen, RM Shelby, M Salinga, WP Risk, TD Happ, GM McClelland, et al. Ultra-thin phase-change bridge memory device using gesb. In *International Electron Devices Meeting*, 2006.

[7] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO '09*, 2009.

[8] CJ Lin et al. 45nm low power cmos logic compatible embedded stt mram utilizing a reverse-connection 1t/1mtj cell. In *IEDM '09*, 2009.

[9] DH Kang et al. Two-bit cell operation in diode-switch phase change memory cells with 90nm technology. In *VLSIT '08*, 2008.

[10] Dmytro Apalkov et al. Spin-transfer torque magnetic random access memory (stt-mram). *JETC*, 9(2), 2013.

[11] Jianbo Dong, Lei Zhang, Yinhe Han, and Xiaowei Li. Wear rate leveling: Lifetime enhancement of PRAM with endurance variation. In *DAC '11*, 2011.

[12] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3d stacking magnetic ram (mram) as a universal memory replacement. In *DAC '08*, 2008.

[13] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS '10*, 2010.

[14] P. G. Emma. Understanding some simple processor-performance limits. *IBM J. Res. Dev.*, 41(3), 1997.

[15] F. Pellizzer et al. A 90nm phase change memory technology for stand-alone non-volatile memory applications. In *Symp. on VLSI Tech.*, 2006.

[16] Fai Yeung et al. Ge2sb2te5 confined structures and integration of 64 mb phase-change random access memory. *Japanese Journal of Applied Physics*, 44(4S), 2005.

[17] Alexandre P. Ferreira, Bruce Childers, Rami Melhem, Daniel Mosse, and Mazin Yousif. Using PCM in next-generation embedded space applications. In *RTAS '10*, 2010.

[18] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mosse. Increasing PCM main memory lifetime. In *DATE '10*, 2010.

[19] International Technology Roadmap for Semiconductors. Process integration, devices, and structures. http://www.itrs.net/links/2007itrs/2007_chapters/2007_PIDS.pdf, 2007.

[20] David Halupka, Safeen Huda, William Song, Ali Sheikholeslami, Koji Tsunoda, Chikako Yoshida, and Masaki Aoki. Negative-resistance read and write schemes for stt-mram in 0.13μm cmos. In *ISSCC '10*, 2010.

[21] Allan Hartstein, Viji Srinivasan, Thomas R. Puzak, and Philip G. Emma. Cache miss behavior: is it sqrt(2)? In *Conf. Computing Frontiers*, 2006.

[22] Yenpo Ho, Garng M Huang, and Peng Li. Nonvolatile memristor memory: device characteristics and design implications. In *ICCAD '09*, 2009.

[23] D Ielmini, S Lavizzari, D Sharma, and AL Lacaita. Physical interpretation, modeling and impact on phase change memory (pcm) reliability of resistance drift due to chalcogenide structural relaxation. In *IEDM '07*, 2007.

[24] Intel. Intel, stmicroelectronics deliver industry's first phase change memory prototypes. http://www.intel.com/pressroom/archive/releases/20080206corp.htm, 2008.

[25] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA '08*, 2008.

[26] J. Liang, R.G.D. Jeyasingh, H.-Y. Chen, and H.-S.P. Wong. A 1.4ua reset current phase change memory cell with integrated carbon nanotube electrodes for cross-point memory application. In *VLSIT '11*, 2011.

[27] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr, and Joel Emer. Adaptive insertion policies for managing shared caches. In *PACT '08*, 2008.

[28] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ISCA '10*, 2010.

[29] JH Oh et al. Full integration of highly manufacturable 512mb pram based on 90nm technology. In *IEDM'06*, 2006.

[30] Norman P Jouppi, Sarita Adve, and Parthasarathy Ranganathan. Reconfigurable caches and their application to media processing. In *ISCA '00*, 2000.

[31] J. Gummaraju K. Luo and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS '01*, 2001.

[32] Michael Kanellos. Ibm changes directions in magnetic memory. `http://news.cnet.com/IBM-changes-directions-in-magnetic-memory/2100-1004_3-6203198.html`, 2007.

[33] Kang et al. A 0.1 μm 1.8V 256Mb 66MHz Synchronous Burst PRAM. In *ISSCC '06*, 2006.

[34] Dimitris Kaseridis, Jeffrey Stuecheli, Jian Chen, and Lizy Kurian John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems. In *HPCA '10*, 2010.

[35] Taeho Kgil, David Roberts, and Trevor Mudge. Improving nand flash based disk caches. In *ISCA '08*, 2008.

[36] Ki Chul Chun et al. A scaling roadmap and performance evaluation of in-plane and perpendicular mtj based stt-mrams for high-density cache memory. *IEEE Journal of Solid-State Circuits*, 48(2), 2013.

[37] Kinam Kim and Su Jin Ahn. Reliability investigations for manufacturable high density pram. In *Reliability Physics Symposium '05*, 2005.

[38] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04*, 2004.

[39] Joonho Kong, Jinhang Choi, Lynn Choi, and Sung Woo Chung. Low-cost application-aware DVFS for multi-core architecture. In *ICCIT '08*, 2008.

[40] Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. Evaluating stt-ram as an energy-efficient main memory alternative. In *ISPASS '13*, 2013.

[41] Stephan Lai and Tyler Lowrey. Oum-a 180 nm nonvolatile memory cell element technology for stand alone and embedded applications. In *IEDM'01*, 2001.

[42] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09*, 2009.

[43] Lee, Kwang-Jin et al. A 90nm 1.8v 512mb diode-switch pram with 266mb/s read throughput. In *ISSCC '07*, 2007.

[44] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. Energy management for commercial servers. *Computer*, 36(12), 2003.

[45] Dean L Lewis and H-HS Lee. Architectural evaluation of 3d stacked rram caches. In *3DIC '09*, 2009.

[46] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA '08*, 2008.

[47] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA '10*, 2010.

[48] Fang Liu and Yan Solihin. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *SIGMETRICS '11*, 2011.

[49] Yong Luo, Olaf M. Lubeck, Harvey Wasserman, Federico Bassetti, and Kirk W. Cameron. Development and validation of a hierarchical memory model incorporating CPU- and memory-operation overlap model. In *WOSP '98*, 1998.

[50] M Hosomi et al. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram. In *IEDM '05*, 2005.

[51] Magnusson, Peter S. et al. Simics: A full system simulation platform. *Computer*, 35, 2002.

[52] Manu Awasthi et al. Handling pcm resistance drift with device, circuit, architecture, and system solutions. In *NVMW '11*, 2011.

[53] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[54] M. Moreto, F.J. Cazorla, A. Ramirez, and M. Valero. Online prediction of applications cache utility. In *IC-SAMOS '07*, july 2007.

[55] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, and Mateo Valero. Mlp-aware dynamic cache partitioning. In *HiPEAC '08*. 2008.

[56] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO '07*, 2007.

[57] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *MICRO '06*, 2006.

[58] Stanford R. Ovshinsky. Reversible electrical switching phenomena in disordered structures. *Phys. Rev. Lett.*, 21, 1968.

[59] John D. Owens, Peter Mattson, Ujval J. Kapasi, William J. Dally, and Scott Rixner. Memory access scheduling. *ISCA '10*, 2000.

[60] A. Pirovano, A.L. Lacaita, A. Benvenuti, F. Pellizzer, S. Hudgens, and R. Bez. Scaling analysis of phase-change memory technology. In *IEDM '03*, 2003.

[61] M. K. Qureshi, M.M. Franceschini, and L.A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA '10*, 2010.

[62] M. K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for MLP-aware cache replacement. In *ISCA '06*, 2006.

[63] M. K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39*, 2006.

[64] M. K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA '09*, 2009.

[65] Qureshi, M. K. et al. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO '09*, 2009.

[66] R. G. Neale, D. L. Nelson, and G. E. Moore. Nonvolatile and reprogramable, the read-mostly memory is here. *Electronics*, 1970.

[67] Richard Fackenthal et al. 19.7 a 16gb reram with 200mb/s write and 1gb/s read in 27nm technology. In *ISSCC '14*, 2014.

[68] SangBum Kim et al. Generalized phase change memory scaling rule analysis. In *Non-Volatile Semiconductor Memory Workshop*, 2006.

[69] Stuart Schechter, Gabriel H. Loh, Karin Straus, and Doug Burger. Use ecp, not ecc, for hard failures in resistive memories. In *ISCA '10*, 2010.

[70] Simone Raoux et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5), 2008.

[71] SL Cho et al. Highly scalable on-axis confined cell structure for high density pram beyond 256mb. In *VLSIT '05*, 2005.

[72] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS '02*, 2002.

[73] Shekhar Srikantaiah, Mahmut Kandemir, and Qian Wang. Sharp control: controlled shared cache management in chip multiprocessors. In *MICRO '09*, 2009.

[74] G Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02*, 2002.

[75] G Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1), 2004.

[76] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28, 2004.

[77] Guang Suo, Xuejun Yang, Guanghui Liu, Junjie Wu, Kun Zeng, Baida Zhang, and Yisong Lin. Ipc-based cache partitioning: An ipc-oriented dynamic shared cache partitioning mechanism. In *ICHIT'08*, 2008.

[78] T Kishi et al. Lower-current and fast switching of a perpendicular tmr for high speed and high density spin-transfer-torque mram. In *IEDM '08*, 2008.

[79] Farhad Tabrizi. The future of scalable stt-ram as a universal embedded memory. *Embedded.com, February*, 2007.

[80] Andrew S. Tanenbaum. *Computer Networks, 3rd Edition*. Prentice Hall, 1996.

[81] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman P Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA '08*, 2008.

[82] W Mueller et al. Challenges for the dram cell scaling to 40nm. In *IEDM '05*, 2005.

[83] Ruisheng Wang, Lizhong Chen, and Timothy Pinkston. An analytical performance model for partitioning off-chip memory bandwidth. In *IPDPS '13*, 2013.

[84] Woo Yeong Cho et al. A 0.18-um 3.0-v 64-mb nonvolatile phase-transition random access memory (pram). *IEEE JSSC*, 40, 2005.

[85] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. Pacman: prefetch-aware cache management for high performance caching. In *MICRO '11*, 2011.

[86] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *ACM SigPlan Notices*, volume 29, 1994.

[87] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA '09*, 2009.

[88] Yi-Hsuan Chiu et al. Impact of resistance drift on multilevel pcm design. In *ICICDT '10*, 2010.

[89] Chenjie Yu and Peter Petrov. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *DAC '10*, 2010.

[90] Wangyuan Zhang and Tao Li. Helmet: A resistance drift resilient architecture for multi-level cell phase change memory system. In *DSN '11*, 2011.

[91] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO '00*, 2000.

[92] Miao Zhou, Santiago Bock, Alexandre P Ferreira, Bruce Childers, Rami Melhem, and Daniel Mossé. Real-time scheduling for phase change main memory systems. In *TrustCom '11*, 2011.

[93] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *TACO*, 8(4), 2012.

[94] Miao Zhou, Yu Du, Bruce R. Childers, Rami Melhem, and Daniel Mosse. Writeback-aware bandwidth partitioning for multi-core systems with pcm. In *PACT '13*, 2013.

[95] Ping Zhou, Yu Du, Youtao Zhang, and Jun Yang. Fine-grained qos scheduling for pcm-based main memory systems. In *IPDPS '10*, 2010.

[96] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09*, 2009.