Writeback-Aware Partitioning and Replacement for Last-Level Caches in Phase Change Main Memory Systems

MIAO ZHOU, YU DU, BRUCE CHILDERS, RAMI MELHEM, and DANIEL MOSSÉ,
University of Pittsburgh

Phase-Change Memory (PCM) has emerged as a promising low-power main memory candidate to replace DRAM. The main problems of PCM are that writes are much slower and more power hungry than reads, write bandwidth is much lower than read bandwidth, and limited write endurance. Adding an extra layer of cache, which is logically the *last-level cache* (LLC), can mitigate the drawbacks of PCM. However, writebacks from the LLC might (a) overwhelm the limited PCM write bandwidth and stall the application, (b) shorten lifetime, and (c) increase energy consumption. Cache partitioning and replacement schemes are important to achieve high throughput for multi-core systems. However, we noted that no existing partitioning and replacement policy takes into account the writeback information.

This paper proposes two writeback-aware schemes to manage the LLC for PCM main memory systems. *Writeback-aware Cache Partitioning* (WCP) is a runtime mechanism that partitions a shared LLC among multiple applications. Unlike past partitioning schemes, our scheme considers the reduction in cache misses as well as writebacks. *Write Queue Balancing* (WQB) replacement policy manages the cache partition of each application intelligently so that the writebacks are distributed evenly among PCM write queues. In this way, applications rarely stall due to unbalanced PCM write traffic among write queues. Our evaluation shows that WCP and WQB result in, on average, 21% improvement in throughput, 49% reduction in PCM writes, and 14% reduction in energy over a state-of-the-art cache partitioning scheme.

Categories and Subject Descriptors: B.3.2 [**Design Styles**]: Cache memories; C.1.4 [**Parallel Architectures**]

General Terms: Design, Performance

Additional Key Words and Phrases: Shared cache, cache partitioning, replacement, phase change memory

ACM Reference Format:
Zhou, M., Du, Y., Childers, B., Melhem, R., and Mossé, D. 2012. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. ACM Trans. Architec. Code Optim. 8, 4, Article 53 (January 2012), 21 pages.
DOI = 10.1145/2086696.2086732 http://doi.acm.org/10.1145/2086696.2086732

1. INTRODUCTION

Energy efficiency is one of the most important considerations in designing computer systems. The expected annual energy cost for data centers is $7.4 billion in 2011 [U.S. Environmental Protection Agency 2007]. Reducing energy consumption for processors and memory systems are two ways to improve energy efficiency. Modern processors have lowered energy/power demand by implementing either dynamic voltage and frequency scaling (DVFS) for sets of cores or by allowing systems to turn cores on and off independently [Kim et al. 2008]. With increasingly energy efficient processors,

This work was supported in part by NSF awards CNS-1012070, CCF-0811295, and CCF-0811352.
This work was done in the PCM@Pitt group (www.cs.pitt.edu/PCM).
Author's address: M. Zhou, University of Pittsburgh, Pittsburgh, PA; email: miaozhou@cs.pitt.edu.

DOI 10.1145/2086696.2086732 http://doi.acm.org/10.1145/2086696.2086732

reducing the energy consumption of memory is critical to the design of energy efficient computers.

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. Phase-Change Memory (PCM) [Lee et al. 2008; Kang et al 2006; Pellizzer et al. 2006] is proposed as a replacement for DRAM or in addition to DRAM (using a smaller DRAM as a cache for the larger PCM). PCM is a desirable technology because it is non-volatile, scales better than DRAM, has low power for reads and very low idle power.

PCM, nevertheless, faces important challenges. It is wearable such that only about 10^7 or 10^8 writes can be performed. A PCM system would fail in days or even hours without wear-leveling or write minimization [Zhou et al. 2009; Qureshi et al. 2009; Cho and Lee 2009; Ferreira et al. 2010b]. PCM is slower and consumes more energy on writes than DRAM.

To mitigate these shortcomings, researchers organize PCM main memory with a small cache [Lee et al. 2009; Ferreira et al. 2010a; Qureshi et al. 2009a]. The cache could be an on-chip cache or an off-chip DRAM cache. Logically, it is the *last-level cache* (LLC) of the PCM main memory. The LLC improves performance by caching highly accessed data, extends PCM lifetime, and reduces energy consumption by filtering a large portion of writes. The system we study has a shared on-chip LLC and a PCM main memory. Efficiently using the LLC is crucial for PCM main memory.

In this paper, we investigate writeback-aware cache partitioning and replacement for the LLC that are beneficial to PCM main memory. Many existing policies focus on minimizing total misses and do not account for writebacks. These policies are suitable for traditional main memory because DRAM reads and writes have the same latency and power consumption, and writebacks are not on the critical path.

Taking into account writebacks is essential due to PCM's unique asymmetric characteristics. PCM writes are much slower and more power hungry than reads. PCM write bandwidth is much lower than read bandwidth. Lastly, PCM has limited write endurance. As a result, too many writebacks from the LLC are likely to overwhelm the limited PCM write bandwidth and stall the application. In addition, each writeback causes a write to the PCM device, which wears the device and consumes a large amount of energy. These negative impacts on throughput, lifetime and energy are shown to be mitigated by adopting writeback-aware cache partitioning and replacement policies.

We study the effects of two techniques on writebacks: reducing writebacks and distributing writebacks evenly among write queues of the PCM device. We propose two writeback-aware schemes to manage the LLC. *Writeback-aware Cache Partitioning* (WCP) partitions a shared LLC between applications to reduce cache misses and writebacks. *Write Queue Balancing* (WQB) replacement ensures that the writebacks are distributed evenly among write queues. By reducing writebacks and avoiding congested write queues, WCP and WQB improve application performance and overall throughput. Our evaluation shows that WCP and WQB result in an average 21% improvement in performance, 49% reduction in PCM writes and 14% reduction in energy over the state-of-the-art Utility-based Cache Partitioning (UCP) [Qureshi and Patt 2006].

In this paper, we make the following contributions.

—We introduce the concept of writeback-aware cache partitioning and cache replacement for the shared last-level cache in PCM main memory.

—We propose WCP to partition the LLC of PCM main memory. Based on writeback information, WCP outperforms a seminal technique, UCP, in terms of throughput, fairness, PCM lifetime and energy efficiency. We further propose WCP with dynamic partitioning weight adjustment (WCP_DYN) that can learn the best partitioning weights at runtime. WCP_DYN outperforms WCP with the best static partitioning weight.
—We propose WQB to manage the cache partition of each application. WQB ensures that the writeback traffic to write queues is balanced.
—We provide a thorough evaluation of WCP and WQB, and analyze their impact on throughput, fairness, lifetime and energy consumption for a variety of workloads.

The remainder of this paper is organized as follows. Section 2 provides background on PCM and motivates this work. Sections 3 and 4 describe WCP and WQB. Section 5 explains the experimental methodology. Section 6 presents evaluation results and analysis. Section 7 discusses related work and Section 8 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1. Phase Change Memory

PCM is a type of non-volatile memory that stores information by changing the state of a chalcogenide material. By applying electrical currents of different intensities and durations (which melt the material and let it cool at different rates), the physical state or phase of the material can be changed from amorphous to crystalline and vice versa. This property implies that the state persists for longer periods of time without needing power hungry refresh. PCM consumes very little power when idle, which makes it an ideal choice for energy efficient memory. Since the configuration of atoms in the material is different for each state (as opposed to the amount of charge stored, as in DRAM), the energy required to change the stored value is much higher.

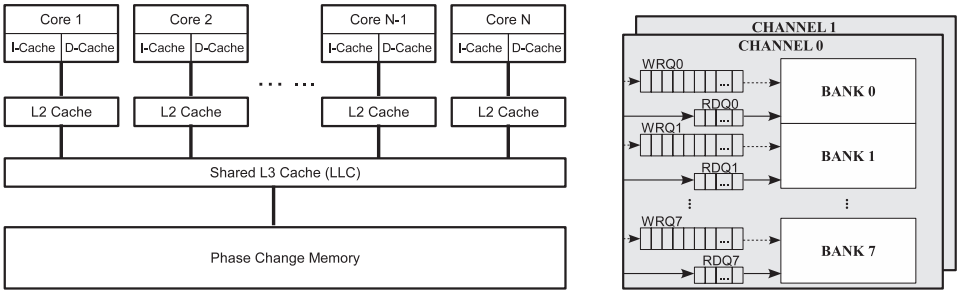
Although these properties make PCM a good replacement for DRAM, it has undesirable write properties. PCM has limited endurance, which is caused by a reduction in a PCM cell's ability to change its physical state after it has been written a large number of times. In addition, PCM writes consume more energy than reads, writes to PCM are about 20-50 times slower than to DRAM (reads are only 2-3 times slower) [Chen et al. 2011], and PCM write bandwidth is smaller than read bandwidth.

2.2. Phase Change Main Memory Architecture

To mitigate the shortcomings of PCM, many researchers propose adding an extra cache layer in front of the PCM main memory [Lee et al. 2009; Ferreira et al. 2010a; Qureshi et al. 2009a]. This cache layer can be used in two ways. It can cache clean and dirty blocks for PCM. 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 1(a) shows the main memory architecture that we consider throughout the paper. 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 on-chip L3 cache, which is the last-level cache. The LLC works as a traditional write-allocate cache with a write-back policy (to PCM). That is, when a modified cache line is evicted from the LLC, it must be written to the PCM. 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 PCM writes.

Figure 1(b) presents the organization of PCM main memory that we consider, with 2 channels of 8 banks each. Each bank has a 8-entry read queue (RDQ) and a 32-entry write queue (WRQ) for pending requests. When a writeback of a cache line from the



(a) Architecture (b) Organization of PCM

Fig. 1. PCM-based main memory architecture.

LLC occurs, a write request is sent to the PCM, which is queued at a write queue. The application progresses without delay if the write queue has available entries since the writebacks are not on the critical path. Otherwise, the application stalls.

2.3. Motivation

Efficiently using the LLC is crucial to improve throughput. For DRAM systems, Qureshi and Patt [2006] show that a reduction in LLC miss rate correlates with an improvement in IPC. 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 phase change main memory, however, reducing miss rate is insufficient to achieve high throughput. Reducing the number of LLC writebacks is also important due to the asymmetric property of PCM. PCM writes are much slower than reads, and PCM write bandwidth is much lower than read bandwidth. These characteristics make it likely that the write queues will become congested. Once the write queues 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 PCM write is much slower than a read.

Writebacks can also negatively affect PCM lifetime and energy consumption. PCM has limited write endurance, and PCM writes are more power hungry than reads. As a result, too many writebacks from the LLC shorten lifetime and consume more energy.

To the best of our knowledge, no existing cache partitioning and replacement scheme accounts for writeback information. Figure 2 shows the architectural overview of our approach to support writeback-aware cache partitioning and replacement. We present four major architectural components (gray colored in Figure 2), 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 queues of the PCM device. 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 queue balancing replacement logic* uses information from MMON to pick a victim entry when a cache miss occurs. Section 3.2 describes E-UMON. Section 3.3 explains the partition selection scheme. Section 3.4 describes MMON and the weight computation. WQB replacement is described in Section 4.

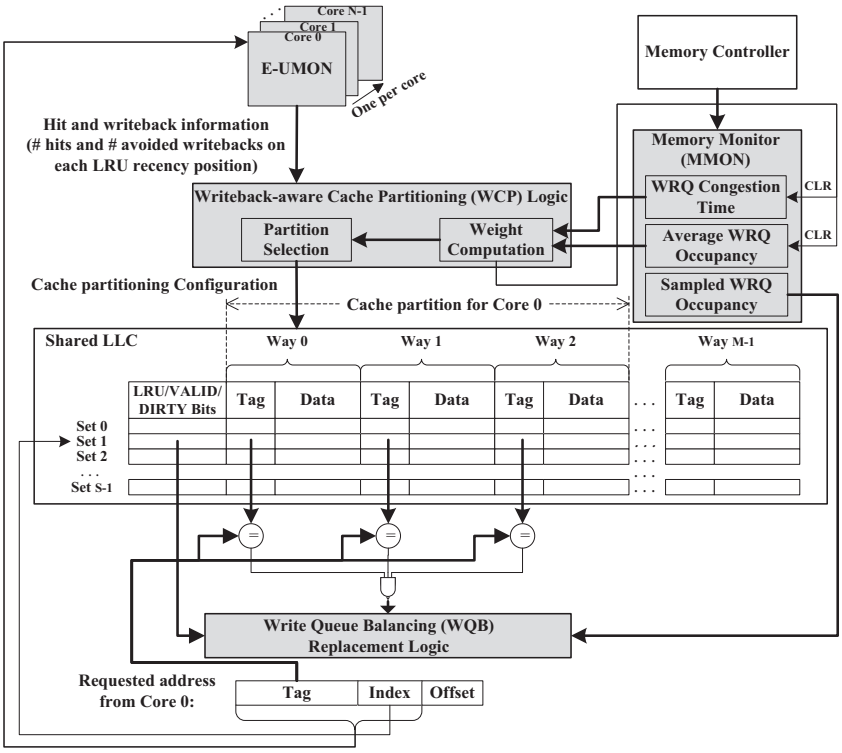


Fig. 2. Architectural overview.

3. WRITEBACK-AWARE CACHE PARTITIONING

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

3.1. Monitoring of Writebacks

The writeback information that we monitor 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. [1970]. 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 3 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. We make the following observations.

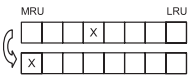
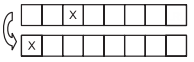
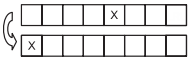
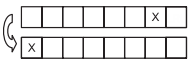
Time	Access	LRU Stack	Stack Distance	Writeback Avoidance Distance
...				
t_1	Write X		4	0
...				
t_2	Read X		3	3
...				
t_3	Read X		5	5
...				
t_4	Read X		2	5
...				
t_5	Write X		7	7

Fig. 3. An example of tracking writebacks from a cache.

—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.
—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.

We draw two conclusions 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, $\{W_0, R_1, R_2, \ldots, R_n, W_{n+1}\}$, including two write accesses (W_0 and W_{n+1}) and n read references ($R_1, R_2, \ldots, R_n$). The stack distances for these accesses are $SD_0, SD_1, SD_2, \ldots, SD_n, SD_{n+1}$. 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 $R_1, R_2, \ldots, R_n$ and W_{n+1} are hits on the cache. In other words, we can save a writeback of the cache line X if and only if $M \geq max(SD_1, SD_2, \ldots, SD_n, SD_{n+1})$.

We define the *writeback avoidance distance* of write access W_0 to cache line X 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. We observe that 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.

ALGORITHM 1: Monitoring Hits

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

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

3.2. 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 [Moreto et al. 2007; Qureshi and Patt 2006]. Similar to UMON [Qureshi and Patt 2006], 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 [Qureshi et al. 2006] 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 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 1). For an M-way cache, M counters $HIT^c_0, HIT^c_1, \ldots, HIT^c_{M-1}$ are used to get hit information. HIT^c_i is the number of additional hits when the i^{th} way is available for core c, where $0 \leq i \leq M-1$. Counter HIT^c_{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 $AWB^c_0, AWB^c_1, \ldots, AWB^c_{M-1}$, where AWB^c_i is the number of avoidable writebacks if the i^{th} way is available for $0 \leq i \leq M-1$ (see Algorithm 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^c_{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 number of hits ($hits_{M'}$) and avoidable writebacks ($avoidablewritebacks_{M'}$) for an M'-way cache, where $M' < M$, as follows.

$$hits_{M'} = \sum_{i=0}^{M'-1} HIT^c_i \tag{1}$$

Table I. Partitioning Weights for Different Optimization Objectives

Optimization Objective	Partitioning Weights
Cache Miss Rate [Qureshi and Patt 2006]	$w_{Hit} = 1$, $w_{Awb} = 0$
Memory Dynamic Energy	$w_{Hit} : w_{Awb} = E_{PR} : E_{PW}$, where E_{PR} and E_{PW} are energy consumption of a PCM read and write.
PCM Writes	$w_{Hit} = 0$, $w_{Awb} = 1$

ALGORITHM 2: Monitoring Writebacks

parameters:
c: 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**
    $AWB^c_{st.WAD} \leftarrow AWB^c_{st.WAD} + 1$
    $st.WAD \leftarrow 0$
  end if
else
  $st.WAD \leftarrow 0$
end if

$$avoidablewritebacks_{M'} = \sum_{i=0}^{M'-1} AWB_i^c \tag{2}$$

3.3. Partition Selection

Let π_i be the number of the ways used by core i. For cache way partitioning, $\{\pi_0, \ldots, \pi_{N-1}\}$ is a valid partition for a system with N cores and an M-way shared LLC, when $\sum_{i=0}^{N-1} \pi_i = M$, and $0 < \pi_i < M$ for $0 \leq i \leq N-1$. 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:

$$\sum_{c=0}^{N-1} \sum_{i=0}^{\pi_c - 1} \left(w_{Hit} \cdot HIT_i^c + w_{Awb} \cdot AWB_i^c \right). \tag{3}$$

Two parameters w_{Hit} and w_{Awb} are *partitioning weights* assigned to hit counters and writeback 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 I gives the partitioning weights for three optimization objectives. UCP was proposed to improve the throughput of a DRAM system by reducing the overall LLC miss rate [Qureshi and Patt 2006]. 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 $w_{Hit} : w_{Awb} = E_{PR} : E_{PW}$, where E_{PR} and E_{PW} are energy consumption of a PCM read and write. The reason is that a PCM read is avoided

for a hit, while a PCM write is saved for each avoided writeback. If reduction of PCM writes is the objective, then $w_{Hit} = 0$ and $w_{Awb} = 1$ should be used.

3.4. Weight Computation for Throughput Optimization

The appropriate partitioning weights for throughput 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 PCM write bandwidth. In this case, the system throughput 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 PCM bandwidth. The penalty of incurring a writeback is much higher than a cache miss as a PCM write 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, we define a new parameter $w = w_{Awb}/w_{Hit}$, which is used in the following maximization problem:

$$\sum_{c=0}^{N-1} \sum_{i=0}^{\pi_c - 1} \left(HIT_i^c + w \cdot AWB_i^c \right). \tag{4}$$

We experimentally show in Section 6.1 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 dynamically for different workloads, or even for different phases of the same workload. We make two observations on picking the partitioning weight.

First, w 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 writebacks. For example, if the write queues are rarely congested, then it is unimportant to reduce the number of writebacks for throughput 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 queues 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 PCM bandwidth.

We propose to adjust w 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 queue congestion time, sampled write queue occupancy, and average write queue occupancy. *Write queue congestion time* (WQCT) is the number of cycles when a write queue is full. *Sampled write queue occupancy* (SWQO) records the occupancy of write queues. The occupancy of a write queue is defined as the ratio of the number of pending requests to its capacity. *Average write queue occupancy* (AWQO) is the average occupancy of write queues during an epoch. WQCT and AWQO are reset at the beginning of each epoch, while SWQO 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 partitioning weight adjustment policy (Algorithm 3). We use the parameters $ThreshOccuHigh = 80\%$, $ThreshOccuLow = 50\%$, $wBoostValue = 5$ and $ThreshWQCongest = 20\%$.

—If there is a write queue with high average occupancy ($AWQO \geq ThreshOccuHigh$) and the write queues are congested for a long time ($WQCT/EPOCH\_CYCLES \geq ThreshWQCongest$), then the weight is either boosted to $wBoostValue$ or incremented.

ALGORITHM 3: Adjusting Partitioning Weight (w) Dynamically

parameters:
WQ: Number of write queues
$AWQO_i$: The average occupancy of write queue i
$WQCT$: Number of cycles when there exists a congested write queue
$EPOCH\_CYCLES$: Number of cycles per epoch

$MaxAWQO = \max_{i=0}^{WQ-1} AWQO_i$

if $MaxAWQO \geq ThreshOccuHigh$ **then**
  if $(WQCT/EPOCH\_CYCLES \geq ThreshWQCongest)$ *and* $(w < wBoostValue)$ **then**
    $w \leftarrow wBoostValue$
  else
    $w \leftarrow w + 1$
  end if
else if $MaxAWQO \leq ThreshOccuLow$ **then**
  $w \leftarrow w - 1$
end if

If the weight is already larger than $wBoostValue$, then it is incremented. Otherwise, the weight is set to $wBoostValue$.

—If there is a write queue with high occupancy ($AWQO \geq ThreshOccuHigh$) but the write queues are not congested for a long time ($WQCT/EPOCH\_CYCLES < ThreshWQCongest$), then the weight is incremented.

—If all write queues are underutilized ($AWQO < ThreshOccuHigh$), then the weight is decremented.

WCP partitions the LLC after the selection of the partitioning weight. WCP selects the partition that produces the maximum weighted sum of total hits and saved writebacks (equation (4)). 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, we add a ($\log N$)-bit core identifier field 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.

3.5. Multiprogrammed and Multithreaded Workloads

WCP works for both multiprogrammed and multithreaded workloads. For multiprogrammed workloads, E-UMON monitors the hit and writeback information of each thread (application), and WCP chooses the cache partition configuration for different threads (applications) that maximizes equation (4).

Kim et al. [2010] classify multithreaded applications into two categories: those whose threads execute mostly independent of each other and those whose threads require frequent synchronization. The first type of multithreaded applications behave in a way similar to multiprogrammed workloads. As a result, WCP and E-UMON deal with them in the same way as multiprogrammed workloads. In contrast, for the second type

ALGORITHM 4: Write Queue Balancing Replacement

```
output:
VictimEntry: the victim entry

Populate S with candidate entries
Set S ⇐ SortByLastTimeUsed(S) {S[1] ← LRU}
if there exists invalid entries then
  VictimEntry ← an invalid entry
else
  VictimEntry ← S[1]
  for i ← 1 to LengthOf(S) do
    WriteQueueId ← GetMappedWriteQueueId(S[i])
    if (IsEntryClean(S[i]) == TRUE) or
    (GetSampledOccupancy(WriteQueueId) < ThreshOccuHigh) then
      VictimEntry ← S[i]
      break
    end if
  end for
end if
```

multithreaded applications, WCP should partition the cache among applications, not threads. E-UMON and WCP view all threads that belong to an application as a group because these threads share, not compete for, the cache resources.

4. WRITE QUEUE BALANCING REPLACEMENT

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

Algorithm 4 shows the pseudocode for WQB 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, WQB searches the candidate entries, and checks whether there exists one that is (a) clean or (b) mapped to a write queue 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. WQB behaves the same as LRU when the applications generate few writes. In this case, each write queue has few pending requests.

5. EXPERIMENTAL METHODOLOGY AND METRICS

We use Simics 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 L3 cache and PCM. Our baseline system is a quad-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 L3 cache for a quad-core system is a 32-way 16MB shared last-level cache (4MB per core). Each core issues two instructions per cycle, and has a 168-entry instruction

Table II. Baseline Configuration

Processor	4-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
L3 Cache	shared, 16MB, 256B line, 32-way, writeback policy 50 cycles access latency
Main Memory	32GB PCM, 2 channels of 8-banks each 32 entry write queue per bank 8 entry read queue per bank read priority scheduling via write pausing [Qureshi et al. 2010]
PCM	read: 200 cycles (50ns) latency, 1 J/GB energy consumption write: 4000 cycles (1μs) latency, 6 J/GB energy consumption

Table III. Benchmark Classification Based on Writebacks per 1000 Instructions

Type	Benchmark
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
ML	401.bzip2, 410.bwaves, 433.milc, 437.leslie3d, 445.gobmk, 473.astar
MH	450.soplex, 459.GemsFDTD, 463.libquantum
H	429.mcf, 470.lbm

Table IV. Multiprogrammed Workloads

Workload	WBPKI	Benchmarks
Light1	1.22	cactusADM, hmmer, perlbench, sjeng
Light2	1.95	bzip2, bwaves, zeusmp, xalancbmk
Medium1	6.00	astar, cactusADM, GemsFDTD, libquantum
Medium2	5.45	astar, leslie3d, soplex, GemsFDTD
Medium3	5.19	astar, bzip2, GemsFDTD, libquantum
Medium4	5.33	bzip2, leslie3d, libquantum, soplex
Medium5	6.64	bzip2, bwaves, GemsFDTD, libquantum
Medium6	8.67	mcf, soplex, xalancbmk, zeusmp
Medium7	9.62	leslie3d, mcf, soplex, xalancbmk
Medium8	7.09	GemsFDTD, lbm, soplex, xalancbmk
Medium9	7.86	bwaves, mcf, xalancbmk, zeusmp
High1	19.86	cactusADM, mcf(2), omnetpp
High2	22.38	gobmk, mcf(2), xalancbmk
High3	19.36	mcf(2), omnetpp, xalancbmk
High4	13.09	lbm, mcf, omnetpp, xalancbmk

window. We also evaluate our schemes on 2-, 8-, and 16-core systems. The L3 cache size (associativity) is proportional to the number of cores. For example, an 8-core system has a 64-way 32MB L3 cache. Table II shows the major parameters in the default configuration. We use the permutation-based page interleaving scheme [Zhang et al. 2000] as the address mapping scheme for phase change main memory. Assuming there are 2^q write queues, the lower order q bits of the L3 tag and the lower order q bits of the L3 cache set index are used as the input to a q-bit bitwise XOR logic to generate the write queue index. This interleaving scheme requires negligible area overhead compared to E-UMON and WCP, which dominate in total overhead (see Section 6.8).

Benchmarks. We use the SPEC CPU2006 benchmarks for evaluation. In Table III, we characterize the benchmarks in terms of writebacks. Based on writebacks per 1K instructions (WBPKI), we classified the benchmarks into four types: Heavy (H), Medium-Heavy (MH), Medium-Light (ML), and Light (L). From this classification, we create 15 workloads as described in Table IV. We also list the WBPKI for each workload for the baseline scheme (UCP). Light, Medium, and Heavy workloads represent the amount of

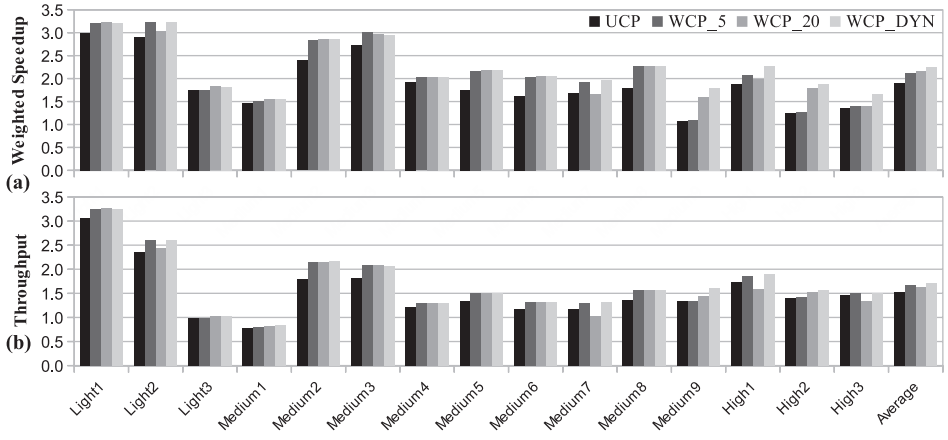


Fig. 4. Weighted speedup (a) and throughput (b) of UCP (WCP with $w = 0$), WCP with weights 5 and 20 and WCP with dynamic weight adjustment (WCP_DYN).

writebacks generated by the workloads. Although we only evaluate multiprogrammed workloads, we expect multithreaded workloads to have similar results as the multiprogrammed workloads (see Section 3.5).

Each benchmark was run in Simics. Simulation for a quad-core system is continued until each benchmark executes at least 2 billion instructions. If one benchmark finishes its 2 billion instructions before the other benchmarks finish, it is restarted so that the four benchmarks continue competing for the shared L3 cache.

Evaluation Metrics. We use three performance metrics for multi-core systems: weighted speedup, throughput and fairness, which are defined as follows.

$$\text{Weighted Speedup} = \sum \left(IPC_i / IPC_i^{Single} \right) \tag{5}$$

$$\text{Throughput} = \sum IPC_i \tag{6}$$

$$\text{Fairness} = N \Big/ \sum \left(IPC_i^{Single} / IPC_i \right), \tag{7}$$

where IPC_i is the IPC of the i^{th} application when it executes with other applications, and IPC_i^{Single} is the IPC 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. The fairness metric is the harmonic mean of normalized IPCs [Kun Luo, Gummaraju, and Franklin 2001]. We also evaluate the impact of the proposed schemes on PCM lifetime and energy consumption.

6. EVALUATION RESULTS AND ANALYSIS

To understand the effectiveness of our schemes, we compare to the state-of-the-art LLC partitioning policy, UCP [Qureshi and Patt 2006]. Sensitivity studies on PCM write latency, number of PCM banks, write queue length and epoch length are included.

6.1. Dynamic Weight Adjustment for WCP

Figure 4 shows the weighted speedup and throughput of WCP with different partitioning weights. 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

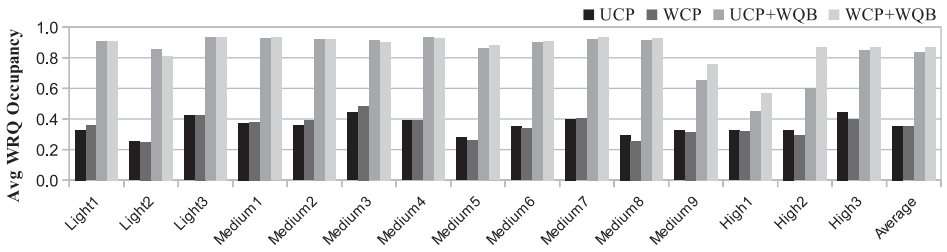


Fig. 5. Average occupancy of write queues when there are congested write queue(s).

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 two workloads. On average, WCP_5 improves throughput by 9%, WCP_20 improves throughput by 6%, and WCP_DYN improves throughput by 11%, compared to UCP. For Medium9, 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 paper, WCP by default means WCP_DYN.

6.2. Unbalanced Write Queues

Figure 5 shows the average occupancy of all write queues when one write queue becomes congested. The instantaneous occupancy of a write queue is defined as the ratio of the number of pending requests to its capacity. Each time a write queue becomes full, the instantaneous occupancy of the write queues are averaged. The average instantaneous occupancy of write queues are then averaged over time. The result shows that the distribution of writebacks among write queues is highly skewed without write queue balancing. On average, when one write queue is full, the other write queues are underutilized (with occupancy less than 0.4) for UCP and WCP. When applying WQB, all write queues are evenly utilized. WQB helps both UCP and WCP to reduce the chances of filling up write queues.

6.3. PCM Lifetime

Figure 6(a) shows the writebacks per 1K instructions (WBPKI) of LRU, UCP, WCP, UCP+WQB and WCP+WQB. We include the evaluation of LRU to estimate the advantages of the partitioned caches (e.g., UCP and WCP) over non-partitioned caches (e.g., LRU). UCP increases the WBPKI by an average of 7% over LRU. As an extreme example, for High1, UCP incurs 110% more writebacks than LRU. 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 workloads, WCP reduces WBPKI by at least 25% over UCP. For Medium9, WCP achieves a reduction of 78% 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 PCM lifetime. As a result, on average, WCP improves lifetime by almost 100% compared to UCP.

6.4. Energy Consumption

Figure 6(b) shows the energy consumption of UCP, WCP, UCP+WQB, and WCP+WQB normalized to LRU. Overall, LRU and UCP have similar energy consumption. WCP

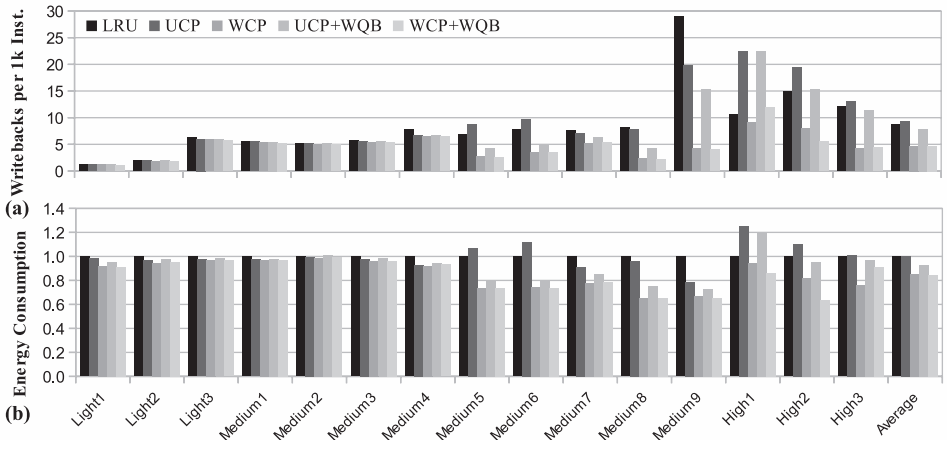


Fig. 6. Impact on (a) PCM lifetime, and (b) energy consumption.

saves energy by minimizing writebacks (i.e., PCM writes) over both LRU and UCP. WCP reduces the energy for eight of the fifteen workloads by reducing PCM writes which are energy hungry. For High1, WCP reduces the energy consumption by 36%. WQB saves energy for eight workloads, and increases energy consumption marginally for three workloads. It is noticable that UCP+WQB can also reduce energy consumption. The reason is that applications with many writebacks stall less frequently, and get a relatively larger cache quota when WQB is applied. The writebacks generated by these applications are reduced as a result of a larger cache partition. On average, WCP, UCP+WQB, and WCP+WQB reduce energy by 14%, 7% and 14% over UCP, respectively.

6.5. Performance

Figure 7(a) shows the weighted speedup for different policies. On average, UCP outperforms LRU by 18% in terms of weighted speedup. For Light1 and Medium1, however, UCP reduces performance marginally compared to LRU. This result happens because UCP changes the cache partition configurations once every epoch (5 million cycles [Qureshi and Patt 2006]) and, therefore, it cannot respond to the phase changes that occur at a finer granularity. LRU, on the other hand, naturally responds to such a fine-grained phase change. WCP improves weighted speedup by more than 20% for seven out of the fifteen workloads over UCP. For Medium9, WCP outperforms UCP by 66%. The benefits of using WQB are as follows. By distributing writebacks evenly among write queues, WQB benefits the weighted speedup in spite of the cache partitioning policy. When UCP is used to partition the cache, WQB improves the weighted speedup by 12% on average. When WCP is applied as the cache partitioning policy, WQB result in an average improvement of 10% in weighted speedup.

Figure 7(b) compares the throughput of the proposed schemes to LRU and UCP. UCP achieves an average of 21% throughput improvement over LRU. WCP further improves the throughput over UCP by 11%. For nine out of the fifteen workloads, WCP improves the throughput by more than 10% over UCP. WCP can improve the IPC of some applications without hurting others. Examples of such workloads are Light1, Light2, Medium2, and Medium6. WCP can also improve the throughput by increasing the IPCs of some applications significantly while marginally reducing the IPCs of others. Examples include Medium6, Medium7, Medium9, High3, and High4. WCP+WQB improves the throughput by 21% on average over UCP.

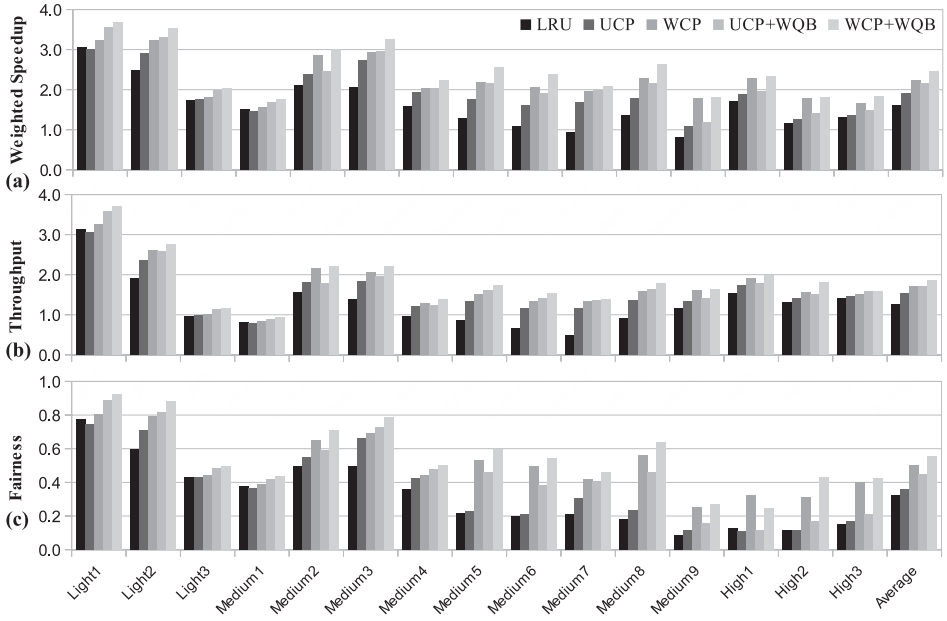


Fig. 7. Impact on performance: (a) weighted speedup, (b) throughput, and (c) fairness.

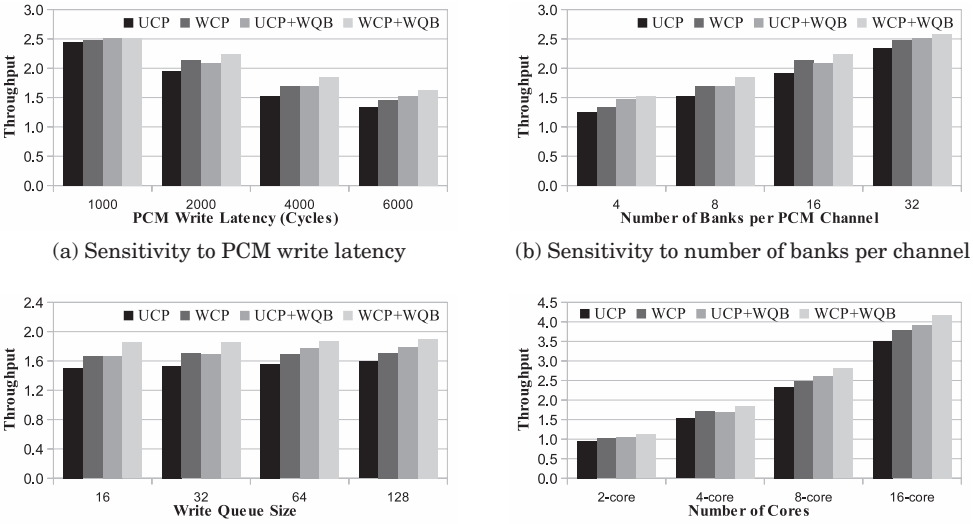
Cache partitioning and replacement policies might improve the performance of some applications by severely penalizing others. The harmonic mean of the normalized IPCs considers both fairness and performance [Kun Luo, Gummaraju, and Franklin 2001]. Figure 7(c) shows the performance of LRU, UCP, WCP, UCP+WQB, and WCP+WQB in terms of fairness. LRU has an average fairness metric value of 0.32, UCP of 0.36, WCP of 0.5, UCP+WQB of 0.45, and WCP+WQB of 0.56. On average, WCP+WQB improves the fairness metric by 14% compared to UCP.

6.6. Sensitivity Study

To analyze the benefit of our schemes under different system configurations, this section describes a study of sensitivity to PCM write latency, number of banks per channel, write queue size, and the epoch length. We show only the results for throughput for brevity. The trends for weighted speedup, fairness, lifetime, and energy are similar.

We vary the write latency of PCM devices to analyze the impact of our schemes. The PCM read latency is unchanged (i.e., 200 cycles). The PCM write latency is varied from 1000 cycles (5x read latency) to 6000 cycles (30x read latency). Figure 8(a) shows the throughput of UCP, WCP, UCP+WQB, and WCP+WQB as the write latency varies. It only shows the average throughput among 15 workloads. The improvement in throughput decreases as PCM 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 PCM bandwith is saturated. The throughput improvement of WCP+WQB is 3% when the write latency is 1000 cycles, 14% for 2000 cycles, 21% for 4000 cycles and 21% for 6000 cycles.

Our baseline configuration assumes each PCM channel has 8 banks. Figure 8(b) shows the throughput impact as the number of banks per channel is varied from 4 to 32. All policies benefit from more banks per channel, as the number of write queues is increased. The effectiveness of our schemes reduces as the number of banks increases



(a) Sensitivity to PCM write latency

(b) Sensitivity to number of banks per channel

(c) Sensitivity to write queue size

(d) Sensitivity to number of cores (scalability)

Fig. 8. Sensitivity of WCP and WQB to (a) PCM write latency, (b) number of banks per channel, (c) write queue size, and (d) number of cores.

because more banks can naturally reduce the likelihood of congested write queues. On average, WCP+WQB improves throughput by 22%, 21%, 16%, and 10% for 4, 8, 16, and 32 banks per channel, respectively.

The baseline contains a 32-entry write queue for each PCM bank. Figure 8(c) shows the throughput for UCP, WCP, UCP+WQB, and WCP+WQB as the write queue length is varied from 16 to 128. All policies benefit very little when the write queue size increases, and the improvement of WCP+WQB over UCP varies slightly for various write queue sizes. The average throughput improvement of WCP+WQB compared to UCP is 23%, 21%, 20% and 19% for 16-, 32-, 64-, and 128-entry write queues.

We also have a sensitivity study on epoch length by varying it from 1 million to 10 million cycles. The results are similar for different epoch lengths. The average throughput improvement of WCP+WQB compared to UCP is 22%, 21%, 21%, and 22% for 1, 2, 5, and 10 million cycles epoch length, respectively (graphs are not included for brevity).

6.7. Scalability

To understand the scalability of our schemes, we evaluate WCP and WQB for a large number of workloads on 2-, 4-, 8-, and 16-core systems with a shared L3 cache of size 8MB, 16MB, 32MB, and 64MB respectively. For each CMP configuration, we use 35 randomly generated workloads. WCP uses the lookahead algorithm [Qureshi and Patt 2006] when scaled to systems with more than 4 cores. Figure 8(d) shows the summarized throughput among all workloads. On average, WCP+WQB improves throughput for 2-, 4-, 8-, and 16-core systems by 18%, 21%, 20%, and 19% over UCP, respectively.

The lifetime scalability results are similar to throughput scalability. On average, WCP+WQB reduces the WBPKI for 2-, 4-, 8-, and 16-core systems by 46%, 49%, 49%, and 47% over UCP, respectively (the graphs are not included for brevity). These results show that our techniques are scalable.

Table V. Storage Overhead for a Quad-Core System

Component	Description	Storage Overhead
Shadow tag array per core	32 sets, 32 entries per set, each entry (V 1b, D 1b, TAG 21b, LRU 5b, WAD 5b)	4224 B
HIT counters per core	32 counters, 4B for each counter	128 B
AWB counters per core	32 counters, 4B for each counter	128 B
E-UMON per core	shadow tag array, HIT counters, AWB counters	4480 B
Total E-UMON overhead	4 E-UMONs	17920 B
WRQ congestion time		4 B
Average WRQ occupancy	2 PCM channels, 8 WRQs per channel, 4B for each WRQ	64 B
Sampled WRQ occupancy	2 PCM channels, 8 WRQs per channel, 4B for each WRQ	64 B
Total MMON overhead		132 B
WCP	2048 sets, 32 tag entries per set, 2b core id in each tag entry	16384 B
Overall overhead		34436 B

6.8. Hardware Overhead

E-UMON and WCP are the two largest sources of hardware overhead in our scheme. Table V 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 quad-core configuration with 16MB shared LLC, our schemes require 0.2% storage overhead. None of the structures or operations required by WCP are on the critical path for cache hits. WQB introduces negligible latency overhead to cache replacement.

7. RELATED WORK

7.1. Phase Change Main Memory Systems

PCM has been proposed as a candidate for main memory systems by many researchers [Ferreira et al. 2010b; Lee et al. 2009; Qureshi et al. 2009a; Zhou et al. 2009; Zhang and Li 2009]. Lee et al. [2009] successfully mitigate the negative impacts of PCM's long latencies, high energy, and finite endurance by using area-neutral buffer organizations and applying partial writes technique. Qureshi et al. [2009a] propose a hybrid architecture that uses a DRAM cache to filter accesses to PCM. Such an anchitecture has the latency benefits of DRAM and the capacity and scalability benefits of PCM. Ferreira et al. [2010b] study the page partitioning in the DRAM cache to reduce the amount of data written back to PCM. Zhou et al. [2009] propose PCM as a direct replacement for DRAM in main memory without buffer organization. Zhang et al. [2000] present a hybrid PCM/DRAM memory architecture that uses a small DRAM as write buffer. OS-level paging scheme is applied to improve PCM write performance and lifetime. Among these architectures, the one proposed in Qureshi et al. [2009a] is the closest to the architecture that we study in the paper.

There is a lot of research that aims at mitigating the undesirable PCM characteristics. As mentioned above, buffer organizations [Ferreira et al. 2010b; Lee et al. 2009; Qureshi et al. 2009a; Zhang and Li 2009] are effective to hide the impacts of slow PCM writes (compared to DRAM). Write cancellation and write pausing [Qureshi et al. 2010] improves the performance of PCM reads by delaying the extremely slow write operations.

There are two ways to extend PCM lifetime: write minimization and wear-leveling. Write minimization techniques reduce the number of writes to PCM. Lazy Write Line Level Write-back, Page level Bypass [Qureshi et al. 2009] are applied in the buffer organization to reduce writes to PCM. Partial Writes [Lee et al. 2009] reduces PCM writes by tracking cache modifications and writing only dirty lines. Redundant Bit-write Removal [Zhou et al. 2009] programs a bit only when the new data bit differs

from the old one. N-Flip-Write [Cho and Lee 2009] further reduces the write traffic to PCM by introducing a bit that indicates whether the associated data is stored negated or not, thereby reducing the number of bit updates.

Wear-leveling techniques strive to distribute write traffic among different cells uniformly. Row-Shifting and Segment swapping [Zhou et al. 2009] balances writes across each row and among segments, and Start-Gap [Qureshi et al. 2009b] tries to balance writes across the entire memory. Ferreira et al. [2010a] propose a swap-based algorithm for wear-leveling. Dong et al. [2011] take into account the endurance variation effect, and balances the wear rates instead of wear traffic of cells across the memory.

Our work is orthogonal to the above mentioned techniques as we investigate the effecient utilization of the cache for multi-core systems with PCM main memory, which has not been studied yet. Our techniques differ from past work as our approaches use writeback information, which has not been previously studied.

7.2. Cache Partition and Replacement

Improving cache performance is essential for CMPs with shared caches. Cache partitioning is a popular way to manage the shard cache among applications. Suh et al. [2004] propose dynamic partitioning to manage the shared cache among competing applications. The cache utility of each application is monitored by counting the number of cache hits to each LRU recency position. Based on the utility information, Suh et al. [2004] partition the cache to achive high overall utility. Qureshi and Patt [2006] extend this idea by separating the utility monitoring circuits for different cores so that the utility information for one application is not polluted by others. A circuit tracks the utility information for the cores and is used to make partitioning decisions to achieve low cache miss rate and high performance. All misses are treated equally. Moreto et al. [2011] differentiate misses based on their impacts on performance. Clustered misses share the penalty as they can be served in parallel, while isolated misses have a greater impact on performance. They propose a dynamic cache partitioning scheme that is aware of such information. Most of the studies of cache partitioning are evaluated by simulations. Lin et al. [2008] design and implement an OS-based cache partitioning mechanism on multicore processors which enables the evaluation of cache partitioning schemes on real machines. Recently, a power-aware cache partitioning scheme is proposed in Kotera et al. [2011] to achieve good performance and low power.

There is limited study on cache replacement for PCM main memory systems. Ferreira et al. [2010b] propose 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.

Qureshi and Patt [2006] and Ferreira et al. [2010a] are the most related studies to this paper. Our work differs from them by taking the writebacks into consideration, as well as making replacement decisions based on additional information about the congestion of the write queues.

8. CONCLUSION

Traditional cache partitioning and replacement do not consider writebacks, which we show to be important to achieve high throughput for phase change main memory systems. Towards that end, this paper describes an efficient mechanism to monitor and gather information about writebacks. We propose two writeback-aware cache management policies that require low hardware overhead. Writeback-Aware Cache Partitioning (WCP) divides the cache among competing applications such that the number of misses and writebacks are reduced. Write Queue Balancing (WQB) replacement spreads writeback traffic among write queues. Evaluation results show that, on average, WCP and WQB improve throughput by 21%, reduce PCM writes by 49%, and

save energy by 14% over a state-of-the-art cache partitioning scheme (UCP). We also show that our schemes are beneficial to a range of multi-core systems and improve the throughput of a 2-, 4-, 8-, and 16-core system by 18%, 21%, 20%, and 19%, respectively.

REFERENCES

CHEN, S., GIBBONS, P. B., AND NATH, S. 2011. Rethinking database algorithms for phase change memory. In *Proceedings of CIDR '11*.

CHO, S. AND LEE, H. 2009. Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance. In *Proceedings of MICRO 42*. 347–357.

DONG, J., ZHANG, L., HAN, Y., AND LI, X. 2011. Wear rate leveling: Lifetime enhancement of pram with endurance variation. In *Proceedings of DAC '11*.

FERREIRA, A. P., CHILDERS, B., MELHEM, R., MOSSE, D., AND YOUSIF, M. 2010a. Using pcm in next-generation embedded space applications. In *Proceedings of RTAS '10*.

FERREIRA, A. P., ZHOU, M., BOCK, S., CHILDERS, B., MELHEM, R., AND MOSSE, D. 2010b. Increasing pcm main memory lifetime. In *Proceedings of DATE '10*.

KANG, S., CHO, W.-Y., CHO, B.-H., LEE, K. J., ET AL. 2006. A 0.1 μm 1.8V 256Mb 66MHz Synchronous Burst PRAM. In *Proceedings of ISSCC '06*.

KIM, W., GUPTA, M. S., YEON WEI, G., AND BROOKS, D. 2008. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Proceedings of HPCA '08*.

KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of MICRO 43*. 65–76.

KOTERA, I., ABE, K., EGAWA, R., TAKIZAWA, H., AND KOBAYASHI, H. 2011. Power-aware dynamic cache partitioning for CMPs. Lecture Notes in Computer Science, vol. 6590, 135–153.

KUN LUO, GUMMARAJU, J., AND FRANKLIN, M. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of ISPASS '01*. 164–171.

LEE, K.-J., CHO, B.-H., CHO, W. Y., KANG, S. ET AL. 2008. A 90 nm 1.8 v 512 mb diode-switch pram with 266 mb/s read throughput. *IEEE J. Solid-State Circuits 43,* 1, 150–162.

LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of ISCA '09*. 2–13.

LIN, J., LU, Q., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of HPCA '08*.

MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9,* 2, 78–117.

MORETO, M., CAZORLA, F., RAMIREZ, A., AND VALERO, M. 2007. Online prediction of applications cache utility. In *Proceedings of IC-SAMOS '07*. 169–177.

MORETO, M., CAZORLA, F. J., RAMIREZ, A., AND VALERO, M. 2011. Dynamic cache partitioning based on the MLP of cache misses. Lecture Notes in Computer Science, vol. 6590, 3–23.

PELLIZZER, F., BENVENVTI, A., GLEIXNER, B., KIM, Y., JOHNSON, B., MAGISTRETTI, M., MARANGON, T., PIROVANO, A., BEZ, R., AND ATWOOD, G. 2006. A 90nm phase change memory technology for stand-alone non-volatile memory applications. In *Proceedings of the Symposium on VLSI Technology*.

QURESHI, M., FRANCESCHINI, M., AND LASTRAS-MONTANO, L. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of HPCA '10*. 1–11.

QURESHI, M. K., LYNCH, D. N., MUTLU, O., AND PATT, Y. N. 2006. A case for mlp-aware cache replacement. In *Proceedings of ISCA '06*. 167–178.

QURESHI, M. K. AND PATT, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of MICRO 39*. 423–432.

QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. 2009a. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of ISCA '09*. 24–33.

QURESHI, M. K., FRANCHESCINI, M., SRINIVASAN, V., LASTRAS, L., ABALI, B., AND KARIDIS, J. 2009. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of MICRO 42*. 14–23.

SUH, G. E., RUDOLPH, L., AND DEVADAS, S. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput. 28*, 7–26.

U.S. ENVIRONMENTAL PROTECTION AGENCY. 2007. Report to congress on server and data center energy efficiency. Public law 109–431.

ZHANG, W. AND LI, T. 2009. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of PACT '09*. 101–112.

ZHANG, Z., ZHU, Z., AND ZHANG, X. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of MICRO 33*. 32–41.

ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of ISCA '09*. 14–23.

Received July 2011; revised October 2011; accepted November 2011