Symmetry-Agnostic Coordinated Management of the Memory Hierarchy in Multicore Systems

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

In a multicore system, many applications share the last-level cache (LLC) and memory bandwidth. These resources need to be carefully managed in a coordinated way to maximize performance. DRAM is still the technology of choice in most systems. However, as traditional DRAM technology faces energy, reliability, and scalability challenges, nonvolatile memory (NVM) technologies are gaining traction. While DRAM is read/write symmetric (a read operation has comparable latency and energy consumption as a write operation), many NVM technologies (such as Phase-Change Memory, PCM) experience read/write asymmetry: write operations are typically much slower and more power hungry than read operations. Whether the memory's characteristics are symmetric or asymmetric influences the way shared resources are managed.

We propose two symmetry-agnostic schemes to manage a shared LLC through way partitioning and memory through bandwidth allocation. The proposals work well for both symmetric and asymmetric memory. First, an exhaustive search is proposed to find the best combination of a cache way partition and bandwidth allocation. Second, an approximate scheme, derived from a theoretical model, is proposed without the overhead of exhaustive search. Simulation results show that the approximate scheme improves weighted speedup by at least 14% on average (regardless of the memory symmetry) over a state-of-the-art way partitioning and memory bandwidth allocation. Simulation results also show that the approximate scheme achieves comparable weighted speedup as a state-of-the-art multiple resource management scheme, XChange, for symmetric memory, and outperforms it by an average of 10% for asymmetric memory.

CCS Concepts: • **Computer systems organization** → *Multicore architectures*; • **Hardware** → *Non-volatile memory*

Additional Key Words and Phrases: Shared resource, cache partitioning, memory bandwidth partitioning, nonvolatile memory, phase change memory

ACM Reference Format:

Miao Zhou, Yu Du, Bruce Childers, Daniel Mosse, and Rami Melhem. 2015. Symmetry-agnostic coordinated management of the memory hierarchy in multicore systems. *ACM Trans. Archit. Code Optim.* 12, 4, Article 61 (December 2015), 26 pages.

DOI: <http://dx.doi.org/10.1145/2847254>

1. INTRODUCTION

Modern computer processors have an increasing number of cores that share hardware resources, causing contention and requiring coordination. Unmanaged sharing of resources results in high interference and poor performance [Bitirgen et al. 2008]. The last-level cache (LLC) and memory bandwidth are two important shared resources [Qureshi and Patt 2006; Liu et al. 2010]. More specifically, memory bandwidth

This work was done in the PCM@Pitt group, supported by NSF awards CNS-1012070.

Authors' addresses: M. Zhou, Y. Du, B. Childers, D. Mosse, and R. Melhem, Department of Computer Science, University of Pittsburgh, 6135 Sennott Square, 210 South Bouquet Street, Pittsburgh, PA 15260; emails: {miao Zhou, fisherdu, childers, mosse, melhem}@cs.pitt.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2015 ACM 1544-3566/2015/12-ART61 \$15.00

DOI: <http://dx.doi.org/10.1145/2847254>

can refer to two different resources: (1) the *interconnection bandwidth* between the chip and the off-chip main memory and (2) the *device bandwidth* of the memory. While both resources are important, the bottleneck depends on the memory architecture.

DRAM has been the de facto main memory technology for decades; however, it faces energy, reliability, and scalability issues in the future [Lee et al. 2009; Qureshi et al. 2009b; Zhou et al. 2009]. Nonvolatile memory (NVM) technologies, such as Phase-Change Memory (PCM) [Kang et al. 2006] and ReRAM [Fackenthal et al. 2014], are promising candidates to replace (or in addition to) DRAM. Unlike DRAM, NVM has a unique read/write asymmetry property. Reading information from NVM is fast and energy efficient. In contrast, NVM writes require longer time and/or higher power. The asymmetric property affects the performance of multicore NVM systems, given the long time for write operations and the contention for shared resources.

Partitioning shared resources is effective to improve performance. For LLCs, *way partitioning* reserves cache ways for applications/cores to improve cache utility and performance. Qureshi and Patt proposed *Utility-based Cache Partitioning* (UCP) to reduce cache misses, based on the observation that a reduction in cache misses correlates with a performance improvement [Qureshi and Patt 2006]. Liu et al. found interconnection bandwidth to be the system bottleneck and proposed *Read-only Bandwidth Partitioning* (RBP) to allocate it [Liu et al. 2010]. Both UCP and RBP improve performance for DRAM systems by considering only cache miss information because cache writebacks have a very limited, if any, impact on symmetric memory systems.

However, cache writeback information has been shown to be important for asymmetric memory. Zhou et al. [2012] proposed *Writeback-aware Cache Partitioning* (WCP) to take into consideration the LLC misses and writebacks. *Writeback-aware Bandwidth Partitioning* (WBP) is proposed to partition the memory device bandwidth, which is shown to be the bottleneck for asymmetric memory [Zhou et al. 2013]. WCP and WBP consider cache writeback information because cache writebacks consume a substantial portion of the device bandwidth.

While partitioning each resource separately is effective, managing multiple resources cooperatively further improves performance because resources tend to interact with each other. Managing multiple resources in a coordinated manner has been shown to be superior than managing them separately for symmetric memory systems [Bitirgen et al. 2008; Wang and Martinez 2015].

Asymmetric and symmetric memories have different characteristics and require different ways to manage the shared resources. Existing coordinated resource management schemes are designed for symmetric memory by ignoring the performance impact of LLC writebacks. Therefore, a new scheme is necessary for asymmetric memory or for both asymmetric and symmetric memory. This article presents the first techniques to **cooperatively** manage and allocate the last-level cache and memory bandwidth for **both** asymmetric and symmetric memory. In order to design a **symmetry-agnostic cooperative** technique, we first examine the quantitative impact of way partitioning and bandwidth allocation in isolation and derive a formulation of the problem that considers the impact of both resources on performance. Based on this formulation, we propose two techniques to manage LLC and bandwidth with the goal of maximizing weighted speedup. Our schemes use an OS/hardware cooperative approach to achieve a balance between hardware cost and effectiveness of the proposals.

Specifically, this article makes the following contributions:

- It quantifies the inefficiency of managing resources independently and motivates new coordinated resource management for both asymmetric and symmetric memory.
- It proposes two schemes: *Search-driven Coordinated Resource Management* (SCRM) and *Model-driven Coordinated Resource Management* (MCRM).

—It presents a thorough evaluation and shows that MCRM is as effective as a state-of-the-art multiresource management scheme, XChange [Wang and Martinez 2015], for a symmetric (DRAM) memory system and outperforms XChange by an average of 10% for an asymmetric (PCM) memory system.

Our proposals apply to different memory technologies (asymmetric and symmetric) because our techniques target the system bottleneck. For instance, bandwidth allocation addresses *device bandwidth* in an asymmetric PCM system and *interconnection bandwidth* for a symmetric DRAM system.

The article is organized as follows. Section 2 explains how way partitioning and bandwidth allocation influence each other to motivate our work. Section 3 describes two proposed techniques, SCRM and MCRM. Section 4 explains the OS/hardware cooperative implementation for coordinated management. Sections 5 and 6 present the experimental methodology, results, and analysis. Section 7 discusses related work, and Section 8 concludes the article.

2. THE PROBLEM OF MANAGING SHARED RESOURCES IN THE MEMORY HIERARCHY

2.1. Existing Shared Resource Management

Shared resource management is crucial for the performance of multicore systems. In particular, applications compete for shared memory resources, and unmanaged resources may cause significant interference between competing applications. Techniques to partition caches [Qureshi and Patt 2006; Sanchez and Kozyrakis 2011; Zhou et al. 2012] and allocate memory bandwidth [Liu et al. 2010; Zhou et al. 2013] have been proposed to reduce interference and improve performance.

Managing multiple shared resources is a more complex problem than managing a single shared resource due to interactions of different resources. Uncoordinated management of multiple resources may result in poor performance due to an inability to consider the interactions of resources. There are two important requirements for coordinated management schemes: scalability and generality. A scheme should apply to large-scale CMPs with affordable overhead and should be general enough to work seamlessly with asymmetric and symmetric memory architectures.

In order to manage shared resources in a coordinated way, it is necessary to understand the performance impact of an allocation decision. Several schemes have been proposed for symmetric memory without considering the performance impact of LLC writebacks [Choi and Yeung 2006; Bitirgen et al. 2008; Chen and John 2011; Wang and Martinez 2015]. Choi and Yeung estimate the performance from simple trials; Bitirgen et al. learn performance functions through artificial neural networks; Chen and John model performance using analytic models; and Wang and Martinez approximate the performance impact of individual resource as utility function. Among these schemes, XChange [Wang and Martinez 2015] is the most scalable. The other techniques are centralized, which limits their applicability for CMPs with large core counts because their overhead increases dramatically with core count. In addition, XChange is designed for DRAM systems by considering only cache miss information. A new scalable scheme is needed to manage the shared resources for asymmetric memory.

2.2. Problem Definition of Coordinated Resource Management

This article proposes a *scalable* coordinated technique to partition two shared resources—the last-level cache capacity and the memory bandwidth—that *applies to both asymmetric and symmetric* memory systems. In particular, this article studies how to cooperatively manage these shared resources to achieve better performance. For ease of presentation, we assume that each application runs on a dedicated core. However, our proposal applies as well to the case where multiple applications run on

the same core (see Section 4.3). In an N -core system with W ways of shared LLC and memory bandwidth of B , the problem of *coordinated management* is to find a cache way partition, $\bar{\pi} = \{\pi_0, \dots, \pi_{N-1}\}$, and a bandwidth allocation, $\bar{\alpha} = \{\alpha_0, \dots, \alpha_{N-1}\}$ and $\bar{\beta} = \{\beta_0, \dots, \beta_{N-1}\}$, to maximize the system weighted speedup under two constraints:

$$\sum_{i=0}^{N-1} \pi_i = W, \text{ where } 1 \leq \pi_i \leq W - N + 1; \quad (1)$$

$$\sum_{i=0}^{N-1} (\alpha_i + \beta_i) = B, \quad (2)$$

where π_i is the number of cache ways allocated to application i , and α_i and β_i are the bandwidth allocated to memory reads and writes of application i . The vectors $\bar{\pi}$, $\bar{\alpha}$, and $\bar{\beta}$ specify the cache way partition and the bandwidth allocation for the whole system. Our techniques specify the cache partitioning policy; they are orthogonal to cache partitioning schemes. Although we assume way partitioning as the cache partitioning scheme, the proposed techniques can also work well with Vantage [Sanchez and Kozyrakis 2011] to allow finer-granularity cache management. Recall that the bandwidth allocation can be enforced at either the interconnection level or the memory device level depending on the system bottleneck. In other words, the allocation remains the same while the system bottleneck may differ for different memory.

2.3. Accurately Modeling of Memory Bandwidth

The memory bandwidth constraint in Equation (2) makes two assumptions about memory bandwidth: (1) the available memory bandwidth is 100% consumed and (2) the allocated bandwidth of each application is fully utilized.

The first assumption is needed to enable the analytical formulation but is not always satisfied in practical systems. Specifically, in a DRAM system, the memory interconnection bus can only transfer memory traffic in one direction at a time (read or write requests). Write-to-read turnaround noticeably hurts the interconnection bandwidth utilization [Chatterjee et al. 2012; Stuecheli et al. 2010]. To model such an effect, we can introduce a parameter r_{ub} to reflect the actual ratio of utilized memory bandwidth to the available memory bandwidth.

Similarly, the second assumption does not consider the effect of prefetching techniques, which are widely applied in modern computing systems. Prefetching techniques fetch the data before it is requested. While an accurate prefetch reduces the access latency and improves performance, a mispredicted prefetch wastes memory bandwidth and pollutes the cache [Lee et al. 2012]. One way to account for the impact of prefetching is to introduce a parameter r_i to reflect the ratio of effectively utilized bandwidth to the allocated memory bandwidth of application i . It is possible to estimate r_i by periodically tracking the ratio of the number of accessed cache blocks to the number of cache blocks brought into the cache for application i .

Hence, Equation (2) can be modified as follows to model the memory bandwidth constraint more accurately:

$$\sum_{i=0}^{N-1} \frac{(\alpha_i + \beta_i)}{r_i} = r_{ub} B. \quad (3)$$

To simplify presentation, we assume r_{ub} and r_i to be 1. Existing bandwidth allocation techniques are based on the same simplification [Liu et al. 2010; Zhou et al. 2013]. It has been shown that an analytic model based on such a simplification can effectively improve system performance.

Fig. 1. Performance of *mcf-bzip2* on an asymmetric memory system.

2.4. Motivation

In the past, the performance impact of bandwidth allocation was believed to be *secondary* to that of way partitioning [Suh et al. 2004]. The intuition behind this belief is that way partitioning can significantly reduce the number of cache misses and writebacks, thus reducing the memory traffic, while bandwidth allocation cannot reduce the memory traffic. It seems natural to only consider the performance impact of way partitioning while ignoring that of bandwidth allocation. A closer examination of the interaction between resources, however, calls into question this belief. Although bandwidth allocation does not directly impact the number of cache misses and writebacks, it decides the speed of processing 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 way 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 allocation affects cache utility.

To quantitatively illustrate the interaction between way partitioning and bandwidth allocation, Figure 1 shows the throughput (sum of IPCs) of a workload, *mcf-bzip2*, under different static way partitioning and bandwidth allocation configurations on an asymmetric memory system.¹ A two-core system is used where one core runs *mcf* and the other core runs *bzip2*. Unsurprisingly, system performance varies for different partitioning configurations. The best performance is obtained when *mcf* and *bzip2* are allocated a 90% and 10% share of the bandwidth (denoted 90%-10%) and two ways and 14 ways of the cache (denoted 2w-14w), respectively. To our point, note that the best bandwidth allocation varies under different way partitions. This result is due to the change in off-chip memory traffic and bandwidth demand for different cache allocations, which impacts the best bandwidth allocation decision. This example validates that the choice of way partitioning influences the bandwidth allocation decision, and vice versa.

Given that the best static configuration for *mcf-bzip2* is 90%-10% and 2w-14w, a natural question is whether other workloads favor the same partition configuration. Toward that end, we measured the performance of 150 randomly selected workloads under different static way partitioning and bandwidth allocation configurations on a two-core system. Figure 2 shows the number of workloads that perform the best for different configurations.² An important point from this graph is that different workloads favor different configurations. No single configuration fits all workloads. A dynamic

¹More experimental details can be found in Section 5.

²Figure 2 is not mirrored because the workloads are randomly selected. The figure would be mirrored if the workloads were picked in a mirrored way.

Fig. 2. Number of workloads favoring each static partitioning configuration.

scheme is necessary to better manage the cache and bandwidth. The graph also shows that most workloads benefit the most when the cache ways are partitioned in an unbalanced fashion. This allocation is reasonable when one application dominates the cache accesses, or when one application is cache friendly while the other is streaming.

Unfortunately, simply adopting both way partitioning and bandwidth allocation independently does not guarantee good performance because doing so ignores the interaction between way partitioning and bandwidth allocation, and each of these schemes makes decisions based on different information. Note that bandwidth allocation assumes that the memory traffic does not change much between consecutive epochs; this assumption may be appropriate when the cache way partition does not change. However, the assumption is not valid when both the cache and bandwidth are managed dynamically, as proposed in this article. Therefore, coordinated ways of managing the cache and memory bandwidth are needed. We describe two such methods next.

3. SYMMETRY-AGNOSTIC COORDINATED MANAGEMENT OF CACHE AND BANDWIDTH

The coordinated management of the LLC and bandwidth determines an allocation of the resources, considering their interaction. We first explain existing analytic models for bandwidth partitioning, which serve as the starting point of coordinated management. We then propose two ways to allocate cache and bandwidth. First, a search-driven technique examines candidate allocations of cache partitions and memory bandwidth to select the one that is expected to maximize performance. Second, a model-driven method uses an approximation to determine the allocation without search.

3.1. Existing Bandwidth Partitioning Techniques

Liu et al. propose an analytic model to understand the performance impact of DRAM memory bandwidth [Liu et al. 2010]. They only consider the bus bandwidth consumed by cache misses because the cache writebacks consume little bandwidth and have a small impact on performance. The model is based on an extension of the additive CPI (cycles per instruction) formula [Emma 1997; Luo et al. 1998]. In a CMP system, the CPI expression of an application (e.g., application i) is decoupled as the sum of the CPI assuming an infinite LLC, the CPI due to LLC misses, and the extra CPI caused by contention on memory bandwidth:

$$CPI_i = CPI_{LLC\infty,i} + h_{m,i}(t_{m,i} + \Delta t_{m,i}), \quad (4)$$

where $h_{m,i}$ is the average number of LLC misses per instruction, $t_{m,i}$ is the average LLC miss penalty, and $\Delta t_{m,i}$ is the average expected delay due to contention on memory bandwidth. If $\lambda_{m,i}$ and f_{clk} denote the rate of LLC misses and the processor clock

frequency, respectively, then $h_{m,i}$ can be expressed as $h_{m,i} = \frac{\lambda_{m,i} CPI_i}{f_{clk}}$. By treating the memory system as a queuing system and applying Little's law, the average queuing delay can be expressed as $\Delta t_{m,i} = \frac{\lambda_{m,i} d_r^2}{\alpha_i^2}$, where d_r is the memory read latency and α_i is the allocated memory bandwidth. By substituting $h_{m,i}$ and $\Delta t_{m,i}$ into Equation (4), the CPI expression is

$$CPI_i = \frac{CPI_{LLC\infty,i}}{1 - \frac{\lambda_{m,i} t_{m,i}}{f_{clk}} - \frac{\lambda_{m,i}^2 d_r^2}{\alpha_i^2}}. \quad (5)$$

Based on this CPI expression, the weighted speedup can be expressed as $Weighted\ Speedup = \sum_{i=0}^{N-1} \frac{CPI_{Single,i}}{CPI_{LLC\infty,i}} (1 - \frac{\lambda_{m,i} t_{m,i}}{f_{clk}} - \frac{\lambda_{m,i}^2 d_r^2}{\alpha_i^2})$. To simplify derivation, it is assumed that $\frac{CPI_{Single,0}}{CPI_{LLC\infty,0}} \approx \dots \approx \frac{CPI_{Single,N-1}}{CPI_{LLC\infty,N-1}} \approx 1$. By adopting the simplification, the weighted speedup expression consists of three components, where two components (1 and $\frac{\lambda_{m,i} t_{m,i}}{f_{clk}}$) are unaffected by bandwidth allocation. Therefore, the problem of maximizing weighted speedup is equivalent to minimizing the third component:

$$F(\bar{\alpha}) = \sum_{i=0}^{N-1} \frac{\lambda_{m,i}^2 d_r^2}{\alpha_i^2}. \quad (6)$$

Zhou et al. [2013] show that cache writebacks consume a large portion of memory bandwidth (i.e., device service cycles) and influence the system performance for asymmetric memory. To consider the impact of cache writebacks, an additional CPI component due to LLC writebacks is introduced to the CPI formula:

$$CPI_i = CPI_{LLC\infty,i} + h_{m,i}(t_{m,i} + \Delta t_{m,i}) + p h_{w,i} \Delta t_{w,i}, \quad (7)$$

where $h_{w,i}$ is the average number of LLC writebacks per instruction, $\Delta t_{w,i}$ is the average expected queuing delay of memory writes, and p is the probability that memory writes are on the critical path. Accordingly, following derivation steps similar to the ones described previously, it has been shown that the weighted speedup is maximized when a function $F(\bar{\pi}, \bar{\alpha}, \bar{\beta})$ is minimized for asymmetric memory:

$$F(\bar{\pi}, \bar{\alpha}, \bar{\beta}) = \sum_{i=0}^{N-1} \left(\frac{(\lambda_{m,i}(\pi_i))^2 d_r^2}{\alpha_i^2} + p \frac{(\lambda_{w,i}(\pi_i))^2 d_w^2}{\beta_i^2} \right), \quad (8)$$

where d_r and d_w are memory read and write latencies, $\lambda_{m,i}(\pi_i)$ and $\lambda_{w,i}(\pi_i)$ are the rates of LLC misses and writebacks with an allocation of π_i cache ways, and α_i and β_i are the bandwidth allocation for memory reads and writes of application i , respectively.

3.2. The Basic Principle of Coordinated Resource Management

As explained in Section 2.2, the objective of the coordinated resource management is to maximize weighted speedup. Weighted speedup maximization can be transformed to a problem of minimizing function $F(\bar{\pi}, \bar{\alpha}, \bar{\beta})$ in Equation (8). Parameter p determines the influence of writebacks on performance. Selecting a suitable value for parameter p is the key to making our proposal applicable to different memory architectures (asymmetric and symmetric). In a DRAM system, LLC writebacks have very limited impact on performance, and thus, a parameter $p = 0$ is adequate. In contrast, in a PCM system, LLC writebacks influence the performance due to expensive and slow memory writes. Therefore, a parameter $p > 0$ is suitable.

In Equation (8), d_r and d_w are known at design time, while p can be determined dynamically at runtime [Zhou et al. 2013]. 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 cache 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, this relies on an implicit assumption that the cache way partition does not change from epoch to epoch.

Given a cache management policy, bandwidth allocation can be applied as long as $\lambda_{m,i}$ and $\lambda_{w,i}$ accurately reflect the rates of cache misses and writebacks for the upcoming epoch. One way of coordinating cache and bandwidth allocation is to estimate $\lambda_{m,i}$ and $\lambda_{w,i}$ for different cache way partitions, $\bar{\pi} = \{\pi_0, \dots, \pi_{N-1}\}$; use the estimated values of $\lambda_{m,i}$ and $\lambda_{w,i}$ to derive the corresponding bandwidth allocations; and then choose the way partition and bandwidth allocation that maximize Equation (8). We propose a search method that relies on runtime measurements to estimate the relation between $\lambda_{m,i}$, $\lambda_{w,i}$, and $\bar{\pi}$ and a model-driven method that relies on an analytic approximation of $\lambda_{m,i}$ and $\lambda_{w,i}$ as functions of $\bar{\pi}$. These methods are described next.

3.3. Search-Driven Coordinated Resource Management

Equation (8) can be optimized by searching many different cache way partitions. To estimate $\lambda_{m,i}$ and $\lambda_{w,i}$ for different cache way partitions, we monitor the number of hits and avoidable writebacks on each cache way using E-UMON (Extended Utility Monitor) [Zhou et al. 2012]. Specifically, E-UMON records the number of additional hits (HIT_c^i) and avoidable writebacks (AWB_c^i) on way c for $0 \leq c \leq W - 1$. By allocating 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 . In other words, when cache way c is not allocated to application i , the number of cache misses and writebacks will be increased by HIT_c^i and AWB_c^i , respectively. Note that the number of additional hits and avoidable writebacks may still be nonzero 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 way partition $\bar{\pi} = \{\pi_0, \dots, \pi_{N-1}\}$ and the epoch length, $EpochLength$:

$$\lambda_{m,i} = \sum_{c=\pi_i}^W HIT_c^i / EpochLength; \quad \lambda_{w,i} = \sum_{c=\pi_i}^W AWB_c^i / EpochLength. \quad (9)$$

Algorithm 1 shows the search approach, *Search-driven Coordinated Resource Management*. SCRMM examines all possible cache partitions (Line 2). For each cache partition, the algorithm computes $\lambda_{m,i}$ and $\lambda_{w,i}$ according to Equation (9) (Line 3). It then calculates the bandwidth allocation by applying Lagrange multipliers to solve the minimization problem of function F in Equation (8) (Line 4) and computes the value of function F (Line 5). The algorithm finds the way partition and bandwidth allocation that minimizes $F(\bar{\pi}, \bar{\alpha}, \bar{\beta})$, yielding the maximum weighted speedup (Lines 6–9).

ALGORITHM 1: Search-Driven Coordinated Resource Management

```

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

A problem for SCRM is the long search process. The algorithm needs to compute the bandwidth allocation under all possible cache partitions. Although the search overhead can be dramatically reduced by using dynamic programming techniques [Moreto et al. 2009], the latency overhead is still too expensive. 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 way partition configuration used in the last epoch (epoch $k - 1$), $\{\pi_{0,k-1}, \dots, \pi_{N-1,k-1}\}$, with a limited search radius of r . A radius of r means SCRM searches only partition configurations $\bar{\pi}' = \{\pi'_0, \dots, \pi'_{N-1}\}$ when $|\pi'_i - \pi_{i,k-1}| \leq r$ for $0 \leq i \leq N - 1$. Section 6.2 shows the sensitivity analysis of the latency overhead for different values of r .

3.4. Model-Driven Coordinated Resource Management

To avoid the long search process, we devise an analytic model that considers the performance impact of way partitioning and bandwidth allocation together. Using the analytic model, we propose a novel coordinated scheme, *Model-driven Coordinated Resource Management*. In comparison to SCRM, the tradeoff is that MCRM relies on approximation and may not always find the best allocation, but it has much less overhead. We examine this tradeoff in Section 6.2.

Hartstein et al. showed 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 [Hartstein et al. 2006]. 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:

$$\lambda_{m,i} = A_i u_i \pi_i^{s_i}; \lambda_{w,i} = A_i v_i \pi_i^{t_i}, \quad (10)$$

where s_i , t_i , u_i , and v_i are application-specific coefficients that are obtained through an online curve fitting process. Substituting $\lambda_{m,i}$ and $\lambda_{w,i}$ from Equation (10) into the expression in Equation (8), function F can be expressed as

$$F(\bar{\pi}, \bar{\alpha}, \bar{\beta}) = \sum_{i=0}^{N-1} f_i(\pi_i, \alpha_i, \beta_i), \quad (11)$$

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

$$f_i(\pi_i, \alpha_i, \beta_i) = \frac{A_i^2 u_i^2 \pi_i^{2s_i} d_r^2}{\alpha_i^2} + p \frac{A_i^2 v_i^2 \pi_i^{2t_i} d_w^2}{\beta_i^2}. \quad (12)$$

As explained previously, maximizing the weighted speedup can be 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 (11) and the constraints on π_i , α_i , and β_i given by Equations (1) and (2). To solve this problem, we apply Lagrange multipliers by introducing two new variables (γ_1 , γ_2) and a Lagrange function:

$$L(\bar{\pi}, \bar{\alpha}, \bar{\beta}, \gamma_1, \gamma_2) = F(\bar{\pi}, \bar{\alpha}, \bar{\beta}) + \gamma_1 \left(\sum_{i=0}^{N-1} (\alpha_i + \beta_i) - B \right) + \gamma_2 \left(\sum_{i=0}^{N-1} \pi_i - W \right). \quad (13)$$

Coordinated management is a discrete problem (involves way partitioning), which cannot be directly solved by Lagrange multipliers (used for continuous functions).

Therefore, we use Lagrange multipliers to derive the relationship between way partitioning and bandwidth allocation and then find a near-optimal solution. By differentiating $L(\bar{\pi}, \bar{\alpha}, \bar{\beta}, \gamma_1, \gamma_2)$ with respect to $\pi_i, \alpha_i, \beta_i, \gamma_1$, and γ_2 and solving the differential equations, the bandwidth allocation can be expressed as

$$\alpha_i(\pi_i) = B \cdot g_i(\pi_i) / \sum_{j=0}^{N-1} ((1 + h_j(\pi_j)) \cdot g_j(\pi_j)); \quad (14)$$

$$\beta_i(\pi_i) = h_i(\pi_i) \cdot \alpha_i(\pi_i), \quad (15)$$

where

$$g_i(\pi_i) = \sqrt{s_i A_i^2 \pi_i^{\frac{2}{3}s_i + \frac{4}{3}t_i - 1} u_i^{\frac{2}{3}} d_r^{\frac{2}{3}} v_i^{\frac{4}{3}} d_w^{\frac{4}{3}} p^{\frac{2}{3}} + t_i A_i^2 \pi_i^{2t_i - 1} v_i^2 d_w^2 p}, \quad (16)$$

$$h_i(\pi_i) = \pi_i^{\frac{2}{3}s_i - \frac{2}{3}t_i} u_i^{\frac{2}{3}} d_r^{\frac{2}{3}} v_i^{\frac{2}{3}} d_w^{-\frac{2}{3}} p^{-\frac{1}{3}}. \quad (17)$$

In these equations, d_r and d_w are system parameters, A_i can be monitored and estimated, p can be determined dynamically at runtime [Zhou et al. 2013], and 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. However, the value of π_i remains unknown. Given Equations (14) through (17), way partitioning remains an unsolved problem. A lookahead algorithm [Qureshi and Patt 2006] is introduced to solve the problem (as described next).

In Equations (14) and (15), the bandwidth allocated to an application can be expressed as a function of π_i , that is, $\alpha(\pi_i)$ and $\beta(\pi_i)$. By substituting α_i and β_i from Equations (14) and (15) into Equation (11), function F can be expressed as a function of only the cache way partition $\bar{\pi} = \{\pi_0, \dots, \pi_{N-1}\}$:

$$F(\bar{\pi}) = \sum_{i=0}^{N-1} f_i(\pi_i), \quad (18)$$

where $f_i(\pi_i)$ is derived by substituting α_i and β_i from Equations (14) and (15) into the expression in Equation (12).

The problem of minimizing $F(\bar{\pi})$ in Equation (18) can be translated into a problem of allocating a total of W ways to N applications to maximize the overall utility, where the utility of application i , U_i , is defined as the negative of the function $f_i(\pi_i)$:

$$U_i(\pi_i) = -f_i(\pi_i). \quad (19)$$

Marginal Utility is defined as the utility per unit of 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 way allocation from π_i to $\pi_i + \delta$ for application i is defined as

$$\Delta U_i(\pi_i, \delta) = \frac{f_i(\pi_i) - f_i(\pi_i + \delta)}{\delta}. \quad (20)$$

While the greedy algorithm allocates one unit of resource at each step, a lookahead algorithm [Qureshi and Patt 2006] is able to find a better solution by allocating more than one unit of resource if it achieves a higher marginal gain. Algorithm 2 shows the process of cache way partitioning. First, the way partition is initialized such that each application gets 1 allocated cache way (Lines 1–3). The remaining cache ways are allocated iteratively (Lines 4–18). In each iteration, the marginal utility of each application, $\Delta U_i(\pi[i], \delta)$, with different number of additional allocated cache ways, δ ,

ALGORITHM 2: Lookahead [Qureshi and Patt 2006] Way Partitioning Algorithm

```

1: for  $i = 0$  to  $N - 1$  do
2:    $\pi[i] \leftarrow 1$ 
3: end for
4: while  $W > \sum_{i=0}^{N-1} \pi[i]$  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  $W - \sum_{i=0}^{N-1} \pi[i]$  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

```

is computed (Line 10). The allocation decision with the highest marginal gain is chosen (Lines 8–17). The cache ways are allocated according to the allocation decision (Line 18). After all the necessary iterations to allocate all the cache ways, we arrive at a way partitioning decision $\{\pi[0], \dots, \pi[N - 1]\}$. After deciding on the way partitions, the bandwidth allocation decision can be computed by solving the expressions in Equations (14) and (15).

4. IMPLEMENTATION

We choose to implement MCRM in an OS/hardware cooperative manner. Figure 3 depicts the implementation overview to support coordinated management. Two software components (*CF* and *CRM*) are responsible for computing the resource allocation decision. Three hardware components added in the memory controller (*RMon*, *cache regulator*, and *bandwidth regulator*) are responsible for performance monitoring and resource allocation decision enforcement. Due to the time-varying nature of memory traffic, it is natural to manage the cache and bandwidth periodically.

4.1. Software

There are two software components: CF and CRM. CF (curve fitting module) approximates the miss and writeback rates as exponential functions of the allocated cache ways. It produces the coefficients (s_i , t_i , u_i , and v_i in Equations (16) and (17)) through an online curve fitting for each application. The coefficients are selected from a list of candidate values to minimize the sum of the squares of the errors. CRM (coordinated resource manager) determines the allocation of cache ways and memory bandwidth according to an analytic model based on information from CF and RMon. The way partition and bandwidth allocation decisions are enforced by the cache regulator and bandwidth regulator. RMon and the two regulators are described in Section 4.2.

We propose to implement CRM as part of an OS kernel module in a similar way as XChange [Wang and Martinez 2015]. In many SMP systems with Linux OS, all cores are simultaneously interrupted by an APIC timer every 1ms to periodically update kernel statistics. In order to reduce latency overhead, we implement CRM in a distributed manner by piggybacking on the APIC timer interrupt. A master core collects information from other cores and computes the resource allocation decision. A shared-memory

Fig. 3. Components added to OS and memory controller.

model is assumed so that intercore communication requires no extra hardware. The procedure of CRM is described as follows:

- (1) Every 1ms, after each core has finished the kernel update routine, the master core posts the total number of unallocated cache ways.
- (2) Each core computes its maximum marginal utility and the corresponding number of allocated ways (Line 9–16 of Algorithm 2 in Section 3.4).
- (3) After a global barrier to ensure that every core finishes the computation, the master core collects the maximum marginal utility of each application. It picks the application with the highest maximum marginal utility and allocates cache ways to this application. If all the cache ways are allocated, a cache way allocation decision is reached and the corresponding memory bandwidth allocation decision is computed according to Equations (14) through (17). Otherwise, repeat Step (2).

The latency overhead of MCRM is discussed in Section 6.2.

4.2. Hardware

The resource monitor hardware component, RMon, collects information needed to make management decisions. RMon monitors the number of executed instructions and tracks the cache miss and writeback information using E-UMON.

To enforce way partitioning, a $(\log N)$ -bit core identifier is added to each tag entry for an N -core processor. The core identifier indicates which core owns the cache line. The second hardware component, the cache regulator, ensures that each core does not use more cache resources than its quota. On a cache miss, the cache regulator counts

Table I. Per-Core Hardware Overhead

Component	Quantity	Width	Bits
E-UMON shadow tag arrays	32×128	33	135,168
E-UMON HIT counters	32	32	1,024
E-UMON AWB counters	32	32	1,024
Executed instruction counter	1	32	32
Cache quota	1	5	5
Bandwidth quota	1	32	32
Total			137,285

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 cache regulator replaces the LRU block of that application. Otherwise, the LRU block of all the blocks that do not belong to the application is evicted.

The bandwidth allocation decisions are enforced by the third hardware component, the bandwidth regulator, which includes one *token bucket* per core. While the proposed techniques work well for different memory technologies, it is necessary that the bottleneck bandwidth resource of the memory system is managed and partitioned. CRM distributes the tokens (i.e., device service cycles for asymmetric PCM memory, or interconnection bandwidth for symmetric DRAM memory) among applications, based on decisions from MCRM. A memory request is ready for scheduling when the corresponding bucket has enough tokens to fulfill the request.

Table I details the per-core hardware overhead of MCRM (in bits). MCRM relies on the E-UMON shadow tag array to monitor and predict the cache utility information of each application. Dynamic Set Sampling (DSS) [Qureshi and Patt 2006] is applied to reduce the hardware overhead. We sample 128 out of 4,096 sets. Two counters per cache way are required to record the number of cache hits and avoidable writebacks. In addition, each core needs a counter to track the number of executed instructions. Further, we need two additional counters to record the cache way quota and memory bandwidth quota for each application. In sum, the per-core hardware overhead of MCRM is about 17,161 bytes (137,285 bits), which is 0.43% of a 4MB per-core last-level cache size.

4.3. Implementation Issues

To simplify discussion, we assumed that each application runs on a dedicated core and that applications are independent. We explain next how to apply our proposals to (1) multiple applications run on the same core and (2) multithreaded applications.

For the case where multiple applications run on the same core, CRM and CF allocate last-level cache and memory bandwidth among applications, not cores. RMon monitors the information of the application. By making the monitored information in RMon (the shadow tag arrays, HIT and AWB counters of E-UMON, and the executed instruction counter) part of the context information, applications run on the same core can share the RMon hardware circuit. This incurs an increase of 17,156 bytes to the context information of an application.

For multithreaded workloads, we can either treat each thread independently in the resource allocation process or combine all threads of one application into one unit to compete for the shared resources. The former approach is not desirable due to the fact that it is difficult to determine the owner of a cache block. More specifically, a cache block brought in by one thread might be accessed by another thread of the same application. We choose to treat all threads from the same application as one unit for

Table II. Baseline Configuration

Processor	8-core CMP, 4GHz, Out-of-Order 168-entry instruction window 2 instructions per cycle in each core
L1 I-, D-cache	private, 64KB, 64B line, 4-way, LRU
L2 Cache	private, 2MB, 256B line, 8-way, LRU 15 cycles access latency
L3 Cache (LLC)	shared, 32MB, 256B line, 32-way, writeback 30-cycle access latency
Asymmetric Memory (PCM)	64GB, 4 channels of 2 ranks each 12.8GB/s peak interconnection bandwidth write pausing prioritizes read [Qureshi et al. 2010] read: 200 cycles (50ns) [Zhou et al. 2012] write: 4,000 cycles (1 μ s)
Symmetric Memory (DRAM)	64GB, 4 channels of 2 ranks each 12.8GB/s peak interconnection bandwidth read/write: 50 cycles

Table III. Benchmark Classification

Type	Benchmark
LRLW	perlbench, gcc, gamess, zeusmp, gromacs, cactusADM, dealIII, calculix, wrf
LRHW	hmmer, sjeng
HRLW	bwaves, milc, GemsFDTD, libquantum, omnetpp, xalancbmk
HRHW	bzip2, mcf, leslie3d, gobmk, soplex, lbm, astar, sphinx3

resource allocation. The last-level cache and memory bandwidth are partitioned among applications, not threads.

5. EXPERIMENTAL METHODOLOGY

We use Simics [Magnusson et al. 2002] to model a multicore CMP with three levels of on-chip caches, generating memory traces that are input to an in-house memory simulator. The baseline configuration shown in Table II assumes an eight-core system with either an asymmetric (PCM) or a symmetric (DRAM) memory. In Section 6.6, we show a sensitivity study for different memory read and write latencies.

All experiments, except the sensitivity study on the number of cores, assume a system with eight cores. We carry out the evaluation on two-, four-, four-, and 16-core systems with one application per core for a sensitivity study on different numbers of cores (Section 6.6). The sizes of the private L1 and L2 caches are unchanged throughout the study. The L3 size (associativity) is proportional to the number of cores. That is, the per-core associativity (four-way) and size (4MB) of the L3 cache are unchanged. The main memory uses permutation-based page interleaving [Zhang et al. 2000] for address mapping. The contention on memory interconnection and bank/rank level is modeled.

Benchmarks. We use SPEC CPU2006 for most of the evaluation. In Table III, we characterize the benchmarks according to the amount of memory read and write traffic (after the last-level cache). We classified the benchmarks into four types: low read low write (LRLW), low read high write (LRHW), high read low write (HRLW), and high read high write (HRHW). From this classification, we created 12 multiprogrammed workloads classified as *Light*, *Medium*, and *Heavy* according to the amount of memory read and write traffic. Table IV lists the details about the multiprogrammed workloads.

Table IV. Multiprogrammed Workloads

Workload	Benchmarks
Light1	gameess(2), gcc, milc(2), GemsFDTD(3)
Light2	bwaves(2), hmmer(2), sjeng, libquantum(2), wrf
Light3	gameess, sjeng(2), milc(2), omnetpp(2), wrf
Light4	hmmer(2), perlbench(2), omnetpp, xal(2), zeusmp
Medium1	astar, bwaves(3), milc, omnetpp(2), xal
Medium2	bzip2(2), soplex, libquantum, milc(2), xal(2)
Medium3	gcc(2), lbm, mcf, milc(2), omnetpp(2)
Medium4	hmmer, GemsFDTD(3), leslie3d(2), sjeng(2)
Heavy1	astar(2), hmmer, mcf, milc, soplex(2), sjeng
Heavy2	bwaves, bzip2, lbm(2), gcc, leslie3d, omnetpp(2)
Heavy3	bzip2(2), gcc, milc, libquantum, soplex, sphinx(2)
Heavy4	astar, bwaves(2), mcf(2), sphinx3, xalancbmk(2)

Table V. Multithreaded Workloads (Each Benchmark has 2 Threads)

Workload	Benchmarks
MT1	canneal, streamcluster, x264, specjbb
MT2	canneal, canneal, x264, specjbb
MT3	streamcluster, x264, x264, specjbb
MT4	canneal, streamcluster, streamcluster, specjbb
MT5	canneal, canneal, streamcluster, x264
MT6	streamcluster, x264, specjbb, specjbb

The number in the parentheses indicates the number of instances executed in each run (one is the default and, thus, omitted).

In order to validate the effectiveness of our proposal on multithreaded applications, we also evaluate MCRM for multithreaded workloads listed in Table V. Note that each multithreaded application has two threads.

Each benchmark/thread executes on a dedicated core. Simulation continues until every benchmark/thread executes at least 1 billion instructions. If a benchmark/thread finishes before others have executed 1 billion instructions, it is restarted. This ensures contention for shared resources throughout the experiment.

Evaluation Metrics. We use two performance metrics:

$$\text{Weighted Speedup} = \sum (CPI_{Single,i} / CPI_i); \quad (21)$$

$$\text{Fairness} = N / \sum (CPI_i / CPI_{Single,i}), \quad (22)$$

where CPI_i and $CPI_{Single,i}$ are the CPIs of application i when it competes with other applications and when it executes by itself, respectively. Weighted speedup indicates reduction in execution time [Snavey et al. 2002]; fairness is the harmonic mean of normalized IPCs [Luo et al. 2001]. We do not report the results for throughput because they have the same trend as weighted speedup.

6. EVALUATION RESULTS

To validate its effectiveness, MCRM is compared against a state-of-the-art multire-source allocation scheme, XChange [Wang and Martinez 2015] (Section 6.1). Section 6.2 discusses the performance impact and latency overhead of SCRM and MCRM. We compare MCRM against (1) a nonpartitioned baseline scheme (SHARE assumes LRU as the cache replacement policy and uses the default bandwidth management); (2) *way*

partitioning (WP)-only scheme, where UCP [Qureshi and Patt 2006] is used for symmetric memory and WCP [Zhou et al. 2012] for asymmetric memory; (3) *bandwidth allocation* (BA)-only scheme, where RBP [Liu et al. 2010] is used for symmetric memory and WBP [Zhou et al. 2013] for asymmetric memory; and (4) WP+BA, which applies the way partitioning and bandwidth allocation independently. We evaluate asymmetric (Section 6.3) and symmetric (Section 6.5) memory systems separately for clarity. An evaluation on multithreaded workloads is included in Section 6.4.

The partitioning (allocation) schemes allocate the cache ways and/or bandwidth at the beginning of each epoch, which lasts 5 million cycles. Results are shown normalized to SHARE, unless otherwise noted. The default configuration for the experiments is listed in Table II. However, in Section 6.6, we carry out several sensitivity studies with (1) various read and write latencies, (2) different number of memory channels (between two and 16 channels); (3) different L3 cache sizes (between 16MB and 128MB); and (4) different number of cores (between two and 16).

6.1. Comparing MCRM and XChange

XChange [Wang and Martinez 2015] is a recent state-of-the-art shared resource allocation scheme. Because the pricing-bidding mechanism used in XChange is generic to the problem of multiresource allocation, XChange can be applied to allocate any resource as long as the marginal utility function can be formulated. In the original work, XChange partitions power and cache. We extended XChange to partition cache and memory bandwidth, to compare it against MCRM.

We describe the utility model for memory bandwidth allocation. Miftakhutdinov et al. propose a model to estimate the delay due to memory accesses for a program [Miftakhutdinov et al. 2012]. We estimate the delay due to the memory system in a similar way. A per-core memory critical path counter MCP_{global} is maintained. On a cache read miss, a request is issued to access the main memory. The issued request copies MCP_{global} to its own counter MCP_{local} . After Δt cycles, the request is served, and the critical path counter is set as $MCP_{global} = \max(MCP_{global}, MCP_{local} + \Delta t)$. The value of counter MCP_{global} reflects the delay due to main memory accesses.

The delay due to memory accesses is proportional to the number of memory accesses and inversely proportional to the allocated memory bandwidth. We can collect the statistics from the last interval: the number of cache misses N_m and the allocated memory bandwidth α_0 . Based on information from UMON, we can estimate the number of cache misses $N_m(j)$ for j allocated cache ways. The delay due to memory accesses under new bandwidth allocation α and j allocated cache ways is

$$MCP_{global}(\alpha, j) = MCP_{global} \cdot \frac{N_m(j) \alpha_0}{N_m \alpha}. \quad (23)$$

The marginal utility for memory bandwidth is defined as follows:

$$MU_{memory\ bandwidth}(\alpha + \Delta\alpha, j) = \frac{MCP_{global}(\alpha + \Delta\alpha, j) - MCP_{global}(\alpha, j)}{\Delta\alpha}, \quad (24)$$

where $\Delta\alpha$ is the increment of memory bandwidth.

Figure 4 compares the weighted speedup of MCRM against that of XChange for an asymmetric memory (PCM) system (left) and a symmetric memory (DRAM) system (right). In a symmetric memory system, XChange achieves comparable performance as MCRM in general (XChange is 1.4% worse than MCRM on average). It means that both XChange and MCRM make good decisions on allocating cache and memory bandwidth.

In an asymmetric memory system, however, XChange performs worse than MCRM (XChange is 10.1% worse than MCRM on average). The degradation happens because

Fig. 4. Weighted speedup of MCRM and XChange on (a) an asymmetric and (b) a symmetric memory system.

Fig. 5. Weighted speedup of SCRM and MCRM normalized to SHARE on an asymmetric memory system.

XChange makes decisions relying on cache hit information while ignoring the writeback information, which is important and critical for asymmetric memory.

To evaluate the effectiveness and worthiness of an architectural scheme, we have to consider its cost, in addition to its performance impact. It is shown in Table I that MCRM requires a per-core hardware overhead of 17,161 bytes. To implement XChange on the same system, it requires a per-core cost of 14,532 bytes without limiting the stack distance. MCRM requires 18% more hardware overhead than XChange because MCRM monitors hit as well as writeback information, while XChange relies solely on cache hit information. Similar to XChange, MCRM can make decisions solely based on hit information for symmetric memory. In this way, MCRM achieves comparable performance at a similar hardware cost as XChange for symmetric memory.

6.2. Comparing SCRM and MCRM

Figure 5 shows the weighted speedup of (1) SCRM with a search radius $r = 1$ (SCRM_1), (2) SCRM with a search radius $r = 2$ (SCRM_2), (3) an exhaustive search method (SCRM_all), and (4) MCRM on an asymmetric (PCM) memory system. A larger search radius for SCRM is favorable to improve performance. On average, SCRM_1, SCRM_2, and SCRM_all improve weighted speedup by 65%, 68%, and 75% compared to SHARE. The intuition is that a larger search radius increases the probability of finding the best way partition and bandwidth allocation configuration, but also increases the search

Table VI. Latency Overhead of SCRM and MCRM

Partitioning Scheme	# of Cache Way Partition Iterations	Time (Epochs)
SCRM_1	3,425	1.4
SCRM_2	29,625	11.9
SCRM_all	>10 million	>4,000 ^b
MCRM	1 ^a	<0.038

^aSCRM needs to iterate over multiple cache way partition configurations. In contrast, MCRM uses a lookahead algorithm to directly arrive at a cache way partition decision, avoiding iterating over different cache way partition configurations.

^bThe latency overhead of the exhaustive search method can be dramatically reduced to 3.35 epochs ($N \cdot W^2 = 8,192$ iterations) by using dynamic programming techniques [Moreto et al. 2009].

overhead. In a system with N cores, SCRM_ r needs to iterate over $(2r + 1)^N$ different cache way partition configurations, because the cache way partition for each core has $2r + 1$ possibilities. An exhaustive search method needs to test all possible cache way partitions. For an eight-core system with 32-way cache, SCRM_all needs to iterate over more than 10 million different cache way partition configurations to compute the management decision. In contrast, MCRM arrives at a cache way partition decision using a lookahead algorithm without iterating over different cache way partition candidates. MCRM outperforms SCRM_1 and SCRM_2, but not SCRM_all; MCRM achieves this performance at a fraction of the cost, as described next.

The computation of the resource allocation decision can be overlapped with workload execution [Zhou et al. 2013]. Even though the computational latency overhead is not on the critical path, it determines whether the decision is based on the latest or stale information. The latency overhead is evaluated on a machine with a 4GHz processor (see Table II for the baseline system). To evaluate the latency overhead, we account for the overhead of curve fitting and coordinated resource management. As listed in Table VI, SCRM_1 and SCRM_2 take 1.4 and 11.9 epochs, respectively, to compute the allocation decision. It takes more than 4,000 epochs for SCRM_all to make the decision. The long computational latency of SCRM is undesirable because decisions based on stale information will be obsolete. Unlike SCRM, MCRM makes decisions quickly enough (in less than 4% of the epoch) by avoiding search at the cost of acceptable performance degradation. For the remaining evaluation, we only show results for MCRM.

6.3. Analysis for Asymmetric Memory

To understand the effectiveness of MCRM on asymmetric memory (PCM) systems, this section presents the impact of MCRM on performance (weighted speedup) and fairness. Recall that BA and MCRM allocate **device bandwidth** for asymmetric memory.

Weighted Speedup. Figure 6 shows weighted speedup results for multiprogrammed workloads from Table IV on a system with PCM. On average, BA and WP outperform the baseline by 47% and 49%, respectively. WP outperforms BA for the *Light* workloads. For the *Light* workloads, BA and WP outperform the nonpartitioned scheme by an average of 18% and 26%, respectively. WP provides comparable performance as BA on average for the *Medium* and *Heavy* workloads. For the *Medium* and *Heavy* workloads, both BA and WP improve the weighted speedup by an average of 61% over SHARE. These trends happen because bandwidth allocation by itself can dramatically improve performance when the bandwidth is extremely limited, while it has an insignificant effect when the bandwidth is not the system bottleneck.

BA+WP can further improve performance for most workloads. However, BA+WP performs worse than BA and WP for *Heavy2* and *Heavy4* because BA and WP

Fig. 6. Weighted speedup of BA, WP, BA+WP, and MCRM normalized to SHARE on an asymmetric memory system.

Fig. 7. Fairness of BA, WP, BA+WP, and MCRM normalized to SHARE on an asymmetric memory system.

independently make decisions that negatively affect 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 performance.

Unlike BA+WP, MCRM considers the interaction of way partitioning and bandwidth allocation, relying on a coordinated analytic model. MCRM outperforms other schemes for all workloads. On average, MCRM outperforms BA+WP by 8%, BA by 16.2%, WP by 14.5%, and SHARE by 71%, in terms of weighted speedup.

Fairness. Partitioning the cache and bandwidth might improve the performance of some applications by severely penalizing others. The fairness metric is used to reflect how close to uniformly the management schemes benefit all applications (see Figure 7). On average, MCRM improves the fairness metric by 5.8% compared to BA+WP. MCRM improves fairness by avoiding the scenario of streaming applications starving other applications; streaming applications require large bandwidth but only small cache partition, which is provided by MCRM.

6.4. Multithreaded Workload

MCRM can be applied to multithreaded workloads as explained in Section 4.3. Figure 8 shows weighted speedup of different policies for multithreaded workloads from Table V on a system with PCM. By reducing the interference among applications through partitioning (allocation), BA and WP outperform SHARE by an average of 49.4% and

Fig. 8. Weighted speedup of BA, WP, BA+WP, and MCRM normalized to SHARE for multithreaded workloads on an asymmetric memory system.

Fig. 9. Weighted speedup of BA, WP, BA+WP, and MCRM normalized to SHARE on a symmetric memory system.

51%, respectively. Compared to BA and WP, BA+WP benefits weighted speedup for all workloads, except *MT5*. By treating all threads from the same application as one unit and allocating shared resources among applications, MCRM achieves an average weighted speedup improvement of 9.1% over BA+WP.

6.5. Analysis for Symmetric Memory

To validate the claim that MCRM also works well for symmetric memory, this section shows the impact of MCRM on weighted speedup and fairness in a symmetric memory (DRAM) system. BA and MCRM manage **interconnection bandwidth** for symmetric memory.

Weighted Speedup. Figure 9 shows weighted speedup of different shared resource management schemes on a DRAM memory system. BA and WP outperform SHARE by an average of 20% and 21%, respectively. Comparing to BA and WP, BA+WP benefits the system weighted speedup for most workloads, while it hurts performance for one workload, *Heavy4*, for the same reason as asymmetric memory. MCRM manages cache and memory bandwidth cooperatively so that way partitioning and bandwidth allocation influence each other positively, resulting in an average weighted speedup improvement of 8.5% over BA+WP for a DRAM system.

Fig. 10. Fairness of BA, WP, BA+WP, and MCRM normalized to SHARE on a symmetric memory system.

Fairness. Figure 10 shows the impact on fairness for a DRAM system. On average, MCRM improves the fairness metric by 4% compared to BA+WP. The same explanation of the asymmetric case is also valid for the symmetric case.

6.6. Sensitivity Studies

To analyze the benefit of MCRM under different system configurations, we study the sensitivity of our scheme to the memory read and write latencies, number of memory channels, LLC (L3) size, and number of cores. We show only the results for weighted speedup for brevity. The trend for fairness is similar.

Different memory technologies finish read and write requests at different speeds. STT-RAM processes reads in 30ns and writes in 40ns [Kultursay et al. 2013], PCM in 50ns and 1 μ s [Zhou et al. 2013], and ReRAM in 2 μ s and 10 μ s [Fackenthal et al. 2014]. To understand whether MCRM benefits from different memory technologies, we vary the memory read and write latencies. The read latency (d_r) is varied from 50 to 200 to 1,000 cycles, and the ratio of write latency to read latency (d_w/d_r) takes the values of 1, 5, 10, 20, and 30. Note that BA manages the device bandwidth for the sensitivity study on different memory access latencies. Figure 11(a) shows the average weighted speedup of BA, WP, BA+WP, and MCRM under different memory read and write latencies. It is shown that, for symmetric DRAM memory, allocating the interconnection bandwidth (Figure 9) is more suitable and achieves better weighted speedup than managing the device bandwidth (Figure 11(a)). This result validates the claim that managing the system bottleneck resource is important and beneficial.

The benefit of WP increases as memory writes become slower (i.e., as d_w/d_r increases under a fixed d_r). This result is expected as the importance of way partitioning is proportional to the write latency, which is the penalty of incurring a writeback when the memory is saturated. The benefit of MCRM over all schemes, including BA+WP, is stable across different memory access latencies. This shows that symmetry-agnostic coordinated management is beneficial for different memory technologies.

The remaining sensitivity studies are evaluated on an asymmetric (PCM) memory system. Figure 11(b) shows the performance impact by varying the number of channels while keeping the same number of ranks per channel, that is, varying the total memory bandwidth. The baseline assumes four channels with two ranks per channel. The benefits of BA decrease as the number of memory channels increases, because the importance of bandwidth allocation is proportional to the scarceness of memory

Fig. 11. Sensitivity of MCRM to (a) memory latencies, (b) number of memory channels, (c) LLC size, and (d) number of cores.

Table VII. Latency Overhead of MCRM Under Different Number of Cores

Number of cores	8	16	32	64
Latency overhead (% epoch)	3.8%	4.2%	5.1%	6.3%

bandwidth. However, the effectiveness of MCRM over BA+WP (and the other schemes) is stable under different number of memory channels. On average, MCRM improves weighted speedup by approximately 8% over BA+WP.

It is known that the effectiveness of bandwidth allocation, way partitioning, and coordinated management depends on the memory traffic and the scarceness of the memory bandwidth resource. We vary the size of the LLC to change the traffic to the memory system; see Figure 11(c). The baseline system has a 32MB LLC. WP benefits performance at a greater extent for systems with a smaller LLC capacity. BA is also more effective when the LLC size is smaller. This is expected because a small LLC size filters a small portion of the LLC accesses, resulting in a high pressure to the memory system. MCRM outperforms all schemes under different LLC sizes (e.g., about 8% over BA+WP).

To understand the effectiveness of our schemes on systems with a different number of cores, we evaluated them on two-, four-, eight-, and 16-core systems with a shared LLC of size 8MB, 16MB, 32MB, and 64MB and a system of one, two, four, and eight memory channels, respectively. For each CMP configuration, we use 30 randomly generated workloads. Figure 11(d) shows the average weighted speedup of all workloads. On average, MCRM improves weighted speedup for two-, four-, eight-, and 16-core systems by 7.4%, 9.5%, 8%, and 7.9% over BA+WP, respectively. It shows that MCRM is stable. To validate the scalability of MCRM, we show the latency overhead in Table VII. The latency overhead grows sublinearly with the number of cores. The reason is that the computation of marginal gains for each application can be done locally and concurrently on each core. The latency overhead is quite small when compared with the epoch length. The overhead is even more insignificant if one considers the overhead *per core*: with a

64-core system, the overhead is 6.3% of the epoch length, or less than 0.1% of the total CPU utilization is lost to MCRM. The result shows that MCRM is scalable.

MCRM manages the shared resources at epochs of 5 million cycles. We varied the epoch length from 2 million to 10 million cycles and found that it has limited impact on performance. For an asymmetric (PCM) system, MCRM outperforms BA+WP in terms of average weighted speedup by 7.8%, 8%, and 8.1% for epoch lengths of 2 million, 5 million, and 10 million cycles, respectively. The graph is not included for brevity.

7. RELATED WORK

PCM is a promising energy-efficient NVM technology. While PCM is favored due to its nonvolatility, high scalability, and low-power consumption, it suffers from undesirable properties such as its slow and power-hungry write operations. Lee et al. [2009] mitigate the negative impact of PCM writes through partial write and area-neutral buffer organizations. Redundant Bit-write Removal programs a bit only when the new data bit differs from the old one, avoiding unnecessary write operations [Zhou et al. 2009]. Flip-N-Write further reduces the PCM write traffic [Cho and Lee 2009]. Row-Shifting, Segment Swapping, and Start-Gap balance writes at different levels [Zhou et al. 2009; Qureshi et al. 2009a]. In Ferreira et al. [2010b], a swap-based algorithm is proposed for wear leveling. Dong et al. [2011] take into account the endurance variation effect and balance the wear rates, instead of wear traffic, of cells across the memory. [Qureshi et al. [2009b] propose a hybrid architecture to reduce the number of PCM accesses by introducing a DRAM cache into the hybrid architecture. Ferreira et al. [2010a] study the effect of page partitioning of the DRAM cache to further reduce the PCM traffic. Several architectural designs are adopted to mitigate the undesirable properties of PCM writes [Zhou et al. 2009; Zhang and Li 2009]. Write cancellation and write pausing improve the performance of PCM reads by delaying the slow write operations [Qureshi et al. 2010]. Du et al. [2013a, 2013b] propose compression techniques and bit mapping functions to mitigate undesirable impact of writes. The architecture proposed in Qureshi et al. [2009b] is the closest to the architecture that we study in this article.

Cache partitioning has been shown to be beneficial for performance. Suh et al. [2004] propose to dynamically partition the shared cache. Qureshi and Patt [2006] propose to partition the cache based on cache utility information for each application. Moreto et al. [2008] differentiate cache misses based on their performance impact. Wang et al. [2011] optimize the energy consumption for real-time systems while meeting the task deadline. Ye et al. [2014] propose COLORIS to support both static and dynamic cache partitioning using page coloring. Lin et al. [2008] propose an efficient software approach to support cache partitioning in OS through memory address mapping. Cook et al. [2013] evaluate cache partitioning in hardware. While the cache writeback information is ignored by most researchers, it is important for asymmetric (PCM) memory architecture because of the undesirable characteristics of memory writes. Ferreira et al. [2010b] reduce the number of cache writebacks in exchange for more cache misses. Zhou et al. [2012] partition the last-level cache in a way that both the cache misses and writebacks are reduced.

Similar to the cache, memory bandwidth is also an important resource that influences performance. There are two different ways of managing the memory bandwidth: memory request scheduling and memory bandwidth allocation. FR-FCFS scheduling policy favors requests with row buffer hits over other requests, and older over younger requests [Owens et al. 2000]. The Fair Queuing Memory Scheduler prioritizes requests according to the QoS objectives [Nesbit et al. 2006]. Ipek et al. [2008] use a reinforcement learning algorithm to schedule memory requests. Subramanian et al. [2013] propose a simple Memory-Interference-induced Slowdown Estimation (MISE) model to

estimate application slowdowns due to interapplication interference. Zhou et al. [2011] study the scheduling schemes for the PCM system. While memory request scheduling prioritizes the processing of different memory requests, bandwidth allocation schemes allocate the memory bandwidth. Liu et al. [2010] propose an analytic model to understand the performance impact of off-chip bandwidth allocation. Wang et al. [2013] study the relationship between various bandwidth allocation schemes and different system objectives. Fairness via Source Throttling enables fair sharing of the memory by throttling the cores that cause unfairness [Ebrahimi et al. 2010]. Zhou et al. [2013] recognize the importance of writeback information in asymmetric memory systems.

The interaction of cache and memory bandwidth has been studied by many researchers. Yu and Petrov [2010] minimize the bandwidth requirement through cache partitioning. Zhao et al. [2011] propose a reconfigurable cache hierarchy to optimize the bandwidth provided by the overall hierarchy.

Managing multiple shared resources is a more complex problem than allocating one resource because of the interactions between different resources. To manage resources of a multicore system, Nesbit et al. [2008] propose a hardware/software interface based on the virtual private machine abstraction, allowing software policies to explicitly manage microarchitecture resources. Uncoordinated management of multiple resources is inefficient due to its inability to consider the interactions of resources. Many coordinated management schemes are centralized and unscalable [Choi and Yeung 2006; Bitirgen et al. 2008; Chen and John 2011]. XChange [Wang and Martinez 2015] provides a scalable solution by applying a dynamic, distributed market-based framework to solve the resource allocation problem. Vega et al. [2013] coordinate DVFS and per-core power gating for good power performance efficiency. The Tessellation many-core OS separates global decisions of resource allocation from application-specific scheduling of resources [Colmenares et al. 2013]. All existing techniques are designed for symmetric (DRAM) memory systems as they ignore the performance impact of LLC writebacks. To the best of our knowledge, our proposal is the first scalable solution to manage the last-level cache and the memory bandwidth cooperatively for both asymmetric and symmetric memories.

8. CONCLUSION

In this article, we propose a *coordinated* analytic model to consider the performance impact of partitioning cache ways and memory bandwidth, taking into account the bidirectional relationship between these resources. Based on an analytic model, *Model-driven Coordinated Resource Management* (MCRM) allocates the cache ways and memory bandwidth cooperatively to optimize weighted speedup. MCRM is scalable and effective for both asymmetric and symmetric memory. Our evaluation results show that MCRM achieves comparable performance as a state-of-the-art multiple resource management scheme, XChange, for symmetric memory, and outperforms it by 10% on average for asymmetric memory.

REFERENCES

- Ramazan Bitirgen, Engin Ipek, and Jose F Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO'08*.
- Niladrish Chatterjee, Naveen Muralimanohar, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. 2012. Staged reads: Mitigating the impact of DRAM writes on DRAM reads. In *HPCA'12*.
- Jian Chen and Lizy Kurian John. 2011. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *ICS'11*.
- Sangyeun Cho and Hyunjin Lee. 2009. Flip-N-write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *MICRO'09*.

- Seungryul Choi and Donald Yeung. 2006. Learning-based SMT processor resource distribution via hill-climbing. In *ISCA'06*.
- Jianbo Dong, Lei Zhang, Yinhe Han, and Xiaowei Li. 2011. Wear rate leveling: Lifetime enhancement of PRAM with endurance variation. In *DAC'11*.
- Yu Du, Miao Zhou, Bruce Childers, Daniel Mossé, and Rami Melhem. 2013. Bit mapping for balanced PCM cell programming. In *ISCA'13*.
- Yu Du, Miao Zhou, Bruce Childers, Daniel Mossé, and Rami Melhem. 2013. Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory. *TACO* 9, 4 (Jan. 2013).
- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2010. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS'10*.
- Philip G. Emma. 1997. Understanding some simple processor-performance limits. *IBM J. Res. Dev.* 41, 3 (May 1997).
- Richard Fackenthal, Makoto Kitagawa, Wataru Otsuka, Kirk Prall, Duane Mills, Keiichi Tsutsui, Jahanshir Javanifard, Kerry Tedrow, Tomohito Tsushima, Yoshiyuki Shibahara, and Glen Hush. 2014. 19.7 A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *ISSCC'14*.
- Alexandre P. Ferreira, Bruce Childers, Rami Melhem, Daniel Mossé, and Mazin Yousif. 2010a. Using PCM in next-generation embedded space applications. In *RTAS'10*.
- Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010b. Increasing PCM main memory lifetime. In *DATE'10*.
- Allan Hartstein, Viji Srinivasan, Thomas R. Puzak, and Philip G. Emma. 2006. In *Computing Frontiers'06*.
- Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ISCA'13*.
- Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA'08*.
- Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA'08*.
- Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moreto, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanovic, and John D. Kubiatowicz. 2013. Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation. In *DAC'13*.
- Emre Kultursay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *ISPASS'13*.
- Kyle J. Nesbit, Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Mateo Valero, and James E. Smith. 2008. Multicore resource management. *IEEE Micro* 28, 3 (May 2008).
- Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *ISCA'09*.
- Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.* 9, 1 (March 2012).
- Fang Liu, Xiaowei Jiang, and Yan Solihin. 2010. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA'10*.
- Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing throughput and fairness in SMT processors. In *ISPASS'01*.
- Yong Luo, Olaf M. Lubeck, Harvey Wasserman, Federico Basseti, and Kirk W. Cameron. 1998. Development and validation of a hierarchical memory model incorporating CPU- and memory-operation overlap model. In *Proc. of the 1st Intl. Workshop on Software and Performance*.
- Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. 2012. Predicting performance impact of DVFS for realistic memory systems. In *MICRO'12*.
- Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. 2009. FlexDCP: A QoS framework for CMP architectures. *SIGOPS Oper. Syst. Rev.* 43, 2 (April 2009).
- Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. 2008. MLP-aware dynamic cache partitioning. In *HiPEAC'08*.
- Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair queuing memory systems. In *MICRO'06*.
- John D. Owens, Peter Mattson, Ujval J. Kapasi, William J. Dally, and Scott Rixner. 2000. Memory access scheduling. In *ISCA'00*.

- Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (Feb. 2002).
- Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO'06*.
- Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009a. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *MICRO'09*.
- Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009b. Scalable high performance main memory system using phase-change memory technology. In *ISCA'09*.
- Moinuddin K. Qureshi, Michele Franceschini, and Luis Alfonso Lastras-Montaña. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *HPCA'10*.
- Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *ISCA'11*.
- Sangbeom Kang et al. 2006. A 0.1 μm 1.8V 256Mb 66MHz synchronous burst PRAM. In *ISSCC'06*.
- Allan Snaveley, Dean M. Tullsen, and Geoff Voelker. 2002. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS'02*.
- Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. In *ISCA'10*.
- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA'13*.
- Gookwon E. Suh, Larry S. Rudolph, and Srinivas Devadas. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput.* 28, 1 (April 2004).
- Augusto Vega, Alper Buyuktosunoglu, Heather Hanson, Pradip Bose, and Srinivasan Ramani. 2013. Crank it up or dial it down: Coordinated multiprocessor frequency and folding control. In *MICRO'13*.
- Ruisheng Wang, Lizhong Chen, and Timothy Pinkston. 2013. An analytical performance model for partitioning off-chip memory bandwidth. In *IPDPS'13*.
- Weixun Wang, P. Mishra, and S. Ranka. 2011. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *DAC'11*.
- Xiaodong Wang and J. F. Martinez. 2015. XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *HPCA'15*.
- Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. 2014. COLORIS: A dynamic cache partitioning system using page coloring. In *PACT'14*.
- Chenjie Yu and Peter Petrov. 2010. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *DAC'10*.
- Wangyuan Zhang and Tao Li. 2009. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *PACT'09*.
- Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. 2000. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *MICRO'00*.
- Jishen Zhao, Cong Xu, and Yuan Xie. 2011. Bandwidth-aware reconfigurable cache design with hybrid memory technologies. In *ICCAD'11*.
- Miao Zhou, Santiago Bock, Alexandre Ferreira, Bruce Childers, Rami Melhem, and Daniel Mossé. 2011. Real-time scheduling for phase change main memory systems. In *TrustCom'11*.
- Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. 2012. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *TACO* 8, 4 (Jan. 2012).
- Miao Zhou, Yu Du, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2013. Writeback-aware bandwidth partitioning for multi-core systems with PCM. In *PACT'13*.
- Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *ISCA'09*.

Received June 2015; revised September 2015; accepted November 2015