Dynamic QoS Management for Chip Multiprocessors

BIN LI, Princeton University
LI-SHIUAN PEH, Massachusetts Institute of Technology
LI ZHAO and RAVI IYER, Intel Labs

With the continuing scaling of semiconductor technologies, chip multiprocessor (CMP) has become the de facto design for modern high performance computer architectures. It is expected that more and more applications with diverse requirements will run simultaneously on the CMP platform. However, this will exert contention on shared resources such as the last level cache, network-on-chip bandwidth and off-chip memory bandwidth, thus affecting the performance and quality-of-service (QoS) significantly. In this environment, efficient resource sharing and a guarantee of a certain level of performance is highly desirable. Researchers have proposed different frameworks for providing QoS. Most of these frameworks focus on individual resource for QoS management. Coordinated management of multiple QoS-aware shared resources at runtime remains an open problem. Recently, there has been work that proposed a class-of-service based framework to jointly managing cache, NoC and memory resources simultaneously. However, the work allocates shared resources *statically* at the beginning of application runtime, and do not dynamically track, manage and share shared resources across applications. In this article, we address this limitation by proposing dynamic resource management policies that monitor the resource usage of applications at runtime, then steals resources from the high-priority applications for lower-priority ones. The goal is to maintain the targeted level of performance for high-priority applications while improving the performance of lower-priority applications. We use a PI (Proportional-Integral gain) feedback controller based technique to maintain stability in our framework. Our evaluation results show that our policy can improve performance for lower-priority applications significantly while maintaining the performance for high-priority application, thus demonstrating the effectiveness of our dynamic QoS resource management policy.

Categories and Subject Descriptors: C.4 [Performance of Systems]: design studies; performance attributes

General Terms: Design, Management, Performance

Additional Key Words and Phrases: Cache, joint resource management, network-on-chip(NoC), quality-of-service(QoS)

ACM Reference Format:

Li, B., Peh, L.-S., Zhao, L., and Iyer, R. 2012. Dynamic QoS management for chip multiprocessors. ACM Trans. Architect. Code Optim. 9, 3, Article 17 (September 2012), 29 pages.
DOI = 10.1145/2355585.2355590 <http://doi.acm.org/10.1145/2355585.2355590>

1. INTRODUCTION

With the continuing advancements in semiconductor technologies, chip-multiprocessor (CMP) architectures have become the mainstream design for modern high performance computers. In order to reduce cost and improve system throughput, resources such as

B. Li is currently affiliated with Intel Lab.

Authors' addresses: B. Li, L. Zhao, and R. Iyer, Intel Labs, Hillsboro, OR, 97124; email: {bin.li, li.zhao, ravishankar.iyer}@intel.com; L.-S. Peh, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139; email: peh@csail.mit.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.

© 2012 ACM 1544-3566/2012/09-ART17 \$15.00

DOI 10.1145/2355585.2355590 <http://doi.acm.org/10.1145/2355585.2355590>

on-chip cache, interconnection networks and off-chip memory bandwidth are often shared by all the processor cores on the chip. With CMP architectures, it is expected that more and more applications will run simultaneously on the platform and need to share the platform resources. However, if the shared resources are not managed efficiently, this can cause severe resource contention problem and affect performance for the applications running on the platform in unpredictable ways [Guo et al. 2007; Iyer et al. 2007; Bitirgen et al. 2008]. This performance unpredictability is not suited for future uses of CMPs, such as service-oriented computing, server consolidation and virtualizations [Guo et al. 2007]. Oftentimes, many applications will require a guarantee of a certain level of performance. It becomes important to manage the allocation of these shared resources effectively to provide assured performance levels for certain applications regardless of the other applications running concurrently [Guo et al. 2007; Iyer et al. 2007; Nesbit et al. 2007].

Researchers have proposed different quality-of-service (QoS) frameworks to provide certain levels of performance for different applications. One key objective of QoS is to provide applications a soft upper bound on the execution time by allocating a certain amount of resources to each application. This objective can be achieved by allocating the shared resources either statically or dynamically. For static resource allocation, the amount of resources allocated are kept constant throughout the execution time. This approach is easier to implement and has less hardware overhead. However, many applications exhibit a series of phases, and each phase may have very different requirement on different resources. Static resource allocation to provide a certain level of QoS may not utilize the shared resources efficiently. An alternative is to allocate the shared resources dynamically based on program demands. Dynamic resource allocation continuously reallocates the shared resources based on the resultant performance or target of the application. Dynamic resource allocation can achieve higher resource utilization and enforce system-level performance objectives on CMP.

Among the current dynamic QoS approaches, most focus on only one individual resource. For example, there are works that focus on cache QoS management, but ignores contention in the shared communication fabric [Nesbit et al. 2007; Suh et al. 2001; Kim et al. 2004; Yeh and Reinman 2005; Iyer et al. 2007; Guo et al. 2007; Rafique et al. 2006; Srikantaiah et al. 2009; Wu and Martonosi 2011]. While individual QoS schemes such as cache QoS work well on the CMP platform with 2 or 4 cores, they face problems when scaling to tens of cores where contention for all shared on-chip resources become severe. An application's guaranteed service level is determined by the weakest guarantee for any of its shared resources [Lee et al. 2008]. Guarantee on one individual resource is not sufficient to support overall performance any more in larger systems. There is a critical need to jointly manage all the key shared resources simultaneously and provide QoS at system level. Recently, there has been work that proposed a class-of-service based architecture which can support system level QoS by jointly managing all the critical shared resources on chip simultaneously [Li et al. 2011]. This is the first work to support QoS with consideration of all the key shared resources cache, NoC, and memory all together. The proposed hardware can support both static QoS and dynamic QoS management. However, the article only discussed the static resource allocation policy to support QoS, and did not provide policies for dynamic QoS resource allocation. While static resource allocation for QoS support can guarantee performance for higher-priority applications, it may not utilize the shared resources efficiently as the program may exhibit a series of phases and the demand on shared resources may vary throughout the execution time. As a result, there is a critical need for *dynamic* resource management that supports QoS by *jointly* managing the three critical on-chip shared resources together at runtime. This is the focus of our article.

In this article, we propose dynamic resource management policies that dynamically allocate multiple shared resources (cache, NoC, and memory) for each priority class based on their runtime characteristics. The goal is to further improve the performance of other lower-priority applications running concurrently on the platform while ensuring a certain target level of performance for high-priority applications. This is achieved by a runtime resource management framework that monitors the execution of each application and tries to steal resources from high-priority applications when they are not needed, then allocating them to the lower-priority applications. The runtime resource management framework consists of a hardware performance monitor that tracks the behavior of each application periodically, then feeds the information to a global resource manager that decides how to allocate the shared resources for the next time interval. We use a PI-controller (Proportion-Integral controller) to fine tune cache allocations so as to ensure stability.

We evaluated our proposed policy on a detailed trace-driven simulator with 16 cores interconnected by a mesh network, and show that the proposed dynamic QoS policy can improve performance for medium-priority applications by 12.5% on average, for low-priority applications by 4.3% on average, while having negligible impact on the high-priority applications.

The rest of the article is organized as follows. In Section 2, we present background information and related work. From Sections 3 to 5, we describe our dynamic resource management policies in detail. In Section 6, we discuss the implementation to support our dynamic QoS management framework. In Section 7, we present the evaluation setup and discuss the evaluation results. Finally we conclude the article in Section 8.

2. BACKGROUND AND RELATED WORK

In future CMP architectures, a likely organization is a tiled architecture where each tile contains one or more processor cores, one or two level private caches and a shared distributed last level cache. The tiles are interconnected through a network-on-chip fabric for communications between various cache banks as well as communications between cache banks and off-chip memory through on-chip memory controllers. As a result, the cores contend for cache space, NoC bandwidth and memory bandwidth resources. As the number of tiles increases, the contention for these critical shared resources will increase significantly and might cause destructive interference with each other. Thus, there is a need for quality-of-service (QoS) techniques to ensure a certain level of performance for individual applications that are running concurrently with other applications on the platform. In this section, we discuss popular QoS metrics and commonly used hardware support microarchitecture that researchers have proposed to provide QoS in cache, NoC and memory, while introducing the metrics and hardware support we adopt in our proposal.

2.1. QoS Target Specification

In order to design appropriate QoS policies, researchers have proposed various QoS metrics as target for providing differentiated services for individual applications. Iyer et al. [2007] proposed three types of QoS targets. The first is Resource Usage Metrics (RUM), which specify the amount of resources required by the application, such as cache space. The second is Resource Performance Metrics (RPM), which specify the performance of the resource, such as cache miss rate. The third is Overall Performance Metrics (OPM), which specify the overall performance of the program, for example, IPC. Most prior work in architectural QoS support uses OPM (e.g., IPC) as QoS target [Iyer et al. 2007; Yeh and Reinman 2005; Rafique et al. 2006; Bitirgen et al. 2008; Sharifi et al. 2011]. However, it is difficult to specify a single performance as the global target metric because a lot of programs experience different phases during the execution time

and their performance also varies significantly. Similarly, it is difficult to specify a single RPM such as cache miss rate as the global target because the cache miss rate may change significantly during the execution time as well. In contrast to RPM and OPM, RUM is easier to specify and achieve in hardware and are commonly used in prior work as well [Guo et al. 2007; Nesbit et al. 2007]. However, RUM itself cannot be used for dynamic resource management.

In this article, we use both RUM and OPM as metrics for runtime QoS support to ensure a certain level of performance. We first allocate a certain amount of resources for each application to provide basic QoS support (RUM). We then dynamically adjust the shared resources at runtime so as to maintain the performance of higher-priority applications while pushing up the performance for other lower-priority applications (OPM).

2.2. Related Work

Researchers have proposed different hardware architectures that support a guaranteed allocation for the shared resources. In this subsection, we discuss some of the previous work.

Capacity-based cache QoS support. The most commonly used techniques for cache QoS leverage cache partitioning [Suh et al. 2001; Nesbit et al. 2007; Kim et al. 2004; Yeh and Reinman 2005; Rafique et al. 2006; Iyer et al. 2007; Qureshi and Patt 2006; Srikantaiah et al. 2009; Wu and Martonosi 2011]. For example, Nesbit et al. [2007] proposed a hardware mechanism that can guarantee the allocation of shared cache to different threads. Iyer et al. [2007] proposed a QoS-enabled memory architecture for CMP platforms that control the allocation of cache space to individual workloads for providing QoS. Suh et al. [2001] proposed a dynamic cache partitioning policy that improves system throughput. Kim et al. [2004] proposed a cache partitioning policy that attempts to equalize the amount of performance degradation of equal-priority applications. Yeh and Reinman [2005] proposed a distributed NUCA cache design which can dynamically allocates its distributed cache resources. The goal of this work is to guarantee a minimum performance bound for each core while improving the system throughput. The focus of this policy is to meet the target, not for further improving the system performance. Guo et al. [2007] proposed an admission control policy that accepts jobs only when their QoS targets can be satisfied. In their article, the authors proposed a resource stealing scheme which dynamically steal cache capacity from an elastic job to improve system throughput. Rafique et al. [2006] proposed a hardware-based cache management technique that allows OS-level cache partitioning to ensure performance differentiation. One of the policies they proposed was reactive performance differentiation, which tweaks cache quota allocation until the specified metric is optimized. Srikantaiah et al. [2009] proposed a formal feedback control based method to dynamically partitioning the shared last level cache. To achieve a high fair speedup. Wu and Martonosi [2011] proposed a fine-grained cache capacity management mechanism for shared caches based on timekeeping techniques.

NoC QoS support. A typical NoC router consists of several input ports (each with one or more buffers) contending for switch allocation to one of several output ports. NoC QoS support can be achieved by reserving bandwidth or by reserving buffer space. Examples of those that reserve bandwidth are *Æthereal* [Goossens et al. 2005] and *Nostrum* [Millberg et al. 2004] which implement guaranteed performance services by scheduling the communication channels through time division multiplexing. Works that reserve buffer space are *Mango* [Bjerregaard and Sparso 2005], which utilizes virtual channels to provide connection oriented guaranteed services and connection-less best effort routing, and *QNoC's* router design that uses separate buffers for each service level [Bolotin et al. 2004]. There are also works that partition time into epochs to provide

bandwidth and latency guarantees. For example, Weber et al. [2005] proposed an arbitration scheme based on epochs to provide different QoS services. Lee et al. [2008] proposed globally synchronized frames that allocates frames of buffers at the network interface in CMPs to provide bandwidth and latency guarantees to applications. Das et al. [2009] proposed a batch-based application-aware prioritization policies in NoC to improve system throughput. This work also allows requests from higher-priority applications to be scheduled faster within a batch in the network. Grot et al. [2009] proposed a preemptive virtual clock NoC QoS scheme that provides bandwidth guarantees for applications. It does not require per-flow buffering in the router. However, packets may subject to drop, thus requiring an ACK network and source buffering for retransmission. Grot et al. [2011] recently proposed a lightweight topology-aware NoC QoS architecture to provide fairness to the communicating nodes. However, the main focus of this work is to provide fairness inside NoC. It does not offer any guarantees to applications.

Memory QoS support. For memory QoS support, Nesbit et al. [2006] proposed a fair queuing memory scheduler which can ensure each thread receive its allocated fraction of the memory bandwidth regardless of the load placed on the memory system from other threads. Akesson et al. [2007] proposed a memory controller design that provides a guaranteed minimum bandwidth and maximum latency bound. Neither of these works explored policies for allocating the off-chip bandwidth and left the decisions to the OS. Mutlu and Moscibroda [2007] proposed a stall-time fair memory scheduling algorithm which provides fairness to different threads sharing the memory system. Mutlu and Moscibroda [2008] later on proposed a parallelism-aware batch scheduling that can provide fairness while avoid starvation. However, both works cannot provide any bandwidth guarantee to threads. Ipek et al. [2008] use machine learning techniques to implement memory scheduling policies which aim at maximize memory throughput. Kim et al. [2010a] designed a memory scheduling algorithms that improves system throughput while is scalable to large number of memory controllers. Another work by Kim et al. [2010b] proposed a thread-cluster memory scheduling algorithm that considers both system throughput and fairness together. All these works focus on the memory controller for QoS support.

Joint QoS support. Most of the existing work for run time QoS support focuses on only one individual shared resource, such as cache or memory. These techniques work well for CMP system with several cores. However, future CMP architectures are expected to have tens of cores that share cache, NoC and memory simultaneously. QoS support and management for one single resource is no longer sufficient for providing system level guarantees. It becomes important to jointly managing cache, NoC and memory allocation simultaneously to provide QoS and improve system level performance. To the best of the authors' knowledge, there have only been several works that jointly explored chip resources: Bitirgen et al. [2008] proposed a machine learning based approach to manage multiple shared CMP resources in a coordinated fashion. The shared resources considered in this work were cache, memory and power budget, but ignoring the interconnect effects. The goal of this work was to maximize total system performance. However, it cannot provide differentiated performance to individual applications. Furthermore, the applications studied are all sequential applications. Nesbit et al. [2008] introduced a virtual private machine framework for joint resource management, which allow software policies to explicitly manage microarchitecture resources. However, the work did not provide any mechanism or policy to achieve this. Hansson et al. [2009] proposed a composable and predictable multi-processor SoC platform to fully remove the interference between applications and provide hard real-time guarantees by time division multiplexing. However, the capacity unused by one application cannot be given to another application. Besides, the shared resources considered in this work are NoC

and SRAM. There are no caches and memory controllers on the platform. Later, the Predator memory controller proposed by Akesson et al. [2007] has been integrated into this framework [Akesson 2010]. Ebrahimi et al. [2010] took an alternative approach to improve fairness in the entire shared memory system. Instead of partitioning shared resources to provide fairness, they proposed a source-based fairness control mechanism which constrains the rate at which an application's memory requests are injected to the shared resources. Sharifi et al. [2011] proposed a control theory based dynamic resource allocation scheme to achieve end-to-end specified performance targets. The shared resources considered are cores, caches and memory, but ignoring the interconnect effects. Besides, the framework targets sequential applications that have a single IPC throughout the runtime, but will be very inefficient with applications that have dynamic IPC at runtime. Recently, Li et al. [2011] proposed a class-of-service based architecture (CoQoS) to jointly manage all the three critical shared resources on chip simultaneously: namely cache, NoC, and memory. This work was the first to provide a unified approach to coordinately managing all the critical shared resources on chip for QoS support. The focuses of this work is on architectural support to enable such joint resource management. The proposed work CoQoS is very effective in providing QoS with low hardware overhead. However, the article only presented policies that allocate resources statically for QoS support, and does not present policies for dynamic resource management. Our work in this article aims at further improve performance for lower-priority applications while maintaining the performance for high-priority application compared to static resource allocation framework, thus further improving system performance.

Our work differs from previous works for dynamic resource management in that we are the first to consider all the three critical shared resources with differentiated end-to-end QoS support. We are also the first to investigate dynamic resource management for both sequential applications as well as parallel applications. The dynamic resource management framework we proposed is very effective in improving system performance while maintaining the performance target for higher-priority applications with low hardware overhead.

2.3. Architecture Overview

The focus of this work is on dynamic QoS management policy that jointly manage three critical shared resources simultaneously. We employ the class-of-service based QoS architecture (CoQoS) proposed in Li et al. [2011] as our baseline architecture. In this section, we briefly introduce the CoQoS architecture presented in Li et al. [2011].

Figure 1 gives an overview of the class-of-service based CoQoS architecture. The CoQoS architecture consists of three key components:

Class-of-Service Assignment. This step assigns each application a class-of-service ID (CoS-ID) for the shared resources. This CoS-ID is a thread identifier which indicates the service level the application will receive during its execution. Figure 1(b) illustrates this class-of-service assignment approach. The OS or runtime layer writes the CoS-ID into the CoQoS Register (CQR) whenever it schedules an application or software thread to run on the core.

Class-of-Service Mapping. The second step is to appropriately encode the class-of-service specification so that it can be mapped to shared resource management flexibly. This step allows a separate class-of-service ID for each resource in the CQR register as shown in Figure 1(c) (C-ID for cache, N-ID for NoC and M-ID for memory) for finer resource control.

Once the class-of-service encoding is available, the next step is to allow the class-of-service to be mapped onto cache, NoC and memory resource management. This requires

Fig. 1. Class-of-Service (CoS) based CoQoS architecture overview.

a configuration process to essentially map each class-of-service ID to the appropriate limits or weights that are to be used within the resource for resource management. The limits/weights for each class-of-service ID is maintained in a table of registers that can be either statically specified (on boot-time) or dynamically determined (by the OS or runtime hardware). In this article, we focus on dynamic modification through resource usage profiling to further improve system performance.

Class-of-Service Based Shared Resource Management. The classes-of-service are encoded in the CQR and mapped to a table of limits/weights that are required for use in shared resource management. For cache, NoC and memory resource management, every transaction or request arriving at the resource is tagged with the CoS-ID. Once the CoS-ID is available, the resource locally looks up the limits/weights associated with the CoS-ID and performs the QoS-aware resource management appropriately.

The cache QoS is limit-based (where a maximum space is specified for each class-of-service). The QoS-aware shared cache management achieves cache resource allocation by keeping track of the space used in cache by each priority level and ensuring that this does not cross the maximum limits specified in the mapping table. In order to track the space usage for each priority level, every line in the cache is tagged by the C-ID that is currently using that line. When a new line is allocated, the counter associated with the new line's C-ID is incremented in the Cache-QoS mapping table. If a victim line was evicted from the cache for this allocation, then the counter associated with the victim line's C-ID is decremented from the Cache-QoS mapping table.

The NoC QoS is a two-level weight-based arbitration mechanism to provide guarantees for bandwidth allocation. The OS assigns each application class a NoC priority level encoded as NoC-ID (N-ID) and specifies the weights for arbitration in the NoC-QoS mapping table as shown in Figure 1(c). The higher NoC priority buffer is serviced first in any given cycle until it overflows its weighted threshold within a prespecified

duration of time. It should be noted that if the network bandwidth is underutilized by a high-priority class, then a lower-priority class can use those resources.

The memory QoS support is again weight-based scheduling approach at the memory controller. The OS assigns each application a memory priority level encoded as Memory-ID (M-ID) and specifies the weights for the arbitration in the Memory-QoS mapping table as shown in Figure 1(c). The incoming requests are directed towards different input queues in memory controllers based on the memory priority level (M-ID). When arbitrating amongst input queues, the request queue with the highest memory service level that has a pending request and has not exceeded its weights within a duration is chosen. It is ensured that during any specified duration, the bandwidth provided to each priority class is proportional to the weights indicated in memory mapping table. However, if the memory bandwidth is underutilized by a high-priority level class, then a lower-priority level class can use those resources.

About is a brief introduction about the CoQoS architecture which provides hardware support to jointly managing all the three critical shared resources. For more detailed information, please refer Li et al. [2011]. This CoQoS architecture can be used to statically manage the shared resources or dynamically manage them. In this article, we focus on dynamic QoS management and propose a dynamic QoS management policy that jointly managing three critical shared resources (cache, NoC and memory) simultaneously at run time. We use the CoQoS architecture with static QoS assignment policy as the baseline. Our proposed policy aims at keeping the performance for high-priority application that set at static QoS and improving the performance for other lower-priority applications, and thus improve system performance by stealing resources from high-priority applications and allocating them to other applications whenever possible.

3. PROPOSED DYNAMIC QOS MANAGEMENT POLICY

Figure 2 shows an overview of our proposed dynamic QoS management framework. It is composed of three major components: QoS assignment at scheduling time, QoS mapping at scheduling time, periodic performance monitoring and dynamic resource management at runtime.

In our QoS framework, we assume the platform supports three priority classes, high-priority applications (HPA), medium-priority applications (MPA), and low-priority applications (LPA). As it is highly likely that the CMP platform will run some parallel programs, the number of application running simultaneously on the platform can be much smaller than the total number of CPU cores. Here we illustrate our dynamic QoS design in a platform with three priority classes. For a system that must support more intermediate priority classes, our scheme can be simply extended to provide more classes with proper resources allocation. At scheduling time, the operating system (OS) assigns each application a priority class (HPA, MPA or LPA) based on its performance requirements. The OS also maps each priority class onto specific service IDs, one for each shared resource: C-ID (the quality-of-service level for cache), N-ID (the quality-of-service level for NoC) and M-ID (the quality-of-service level for memory), as shown in Table I, and assigns each ID a certain amount of shared resources (cache space, NoC weights, and memory weights), as shown in Table II.¹ Note a priority class may consist of one application, or a group of applications. These applications can be parallel or sequential. Within each priority class, if it consists of multiple applications, the

¹In this article, we assume three service priority levels for each shared resource, with 0 being the highest priority level, 1 the medium priority level and 2 the lowest priority level for C-ID, N-ID and M-ID, respectively. This can be extended to more priority levels in a similar way.

Fig. 2. Overview of our proposed Dynamic QoS management.

allocated shared resources for this priority class is shared by all applications running within this priority class.

After the assignments, the QoS information is encoded into the cache-QoS mapping table in each L2 bank, NoC-QoS mapping table in each router, and memory-QoS mapping table in each memory controller, as shown in Figure 1(c). In our proposed approach, the limits for cache are specified at scheduling time and dynamically reallocated at runtime. The weights for NoC and memory for each service level are not modified throughout the runtime of an application. Instead, the N-ID and M-ID priority levels assigned at scheduling time for each application are dynamically adjusted at runtime whenever possible. This is because NoC and memory access are prioritized based on levels, which more significantly influences the amount of bandwidth allocated than the actual weights assigned to each level.

Tables I to IV show an example of how our proposed approach assign priority levels and resources at scheduling time and a snapshot of a modified assignment at runtime. In this example, our QoS system determines that HPA has slack in its resource usage, and thus ups the resource allocation of MPA by increasing its cache limit from 35% to 42%, and moving its priority level in NoC and memory to the highest (1 to 0) temporarily, while dropping the resources allocated to HPA correspondingly.

4. HARDWARE PERFORMANCE MONITORING

Such adjustments at runtime are done by two components in our system: (1) the hardware performance monitor, which periodically tracks the behavior of applications in each priority class; and (2) the global resource manager, which dynamically allocates shared resources based on each application's performance in the past as well as the

Table I. Priority Level Assignments at Scheduling Time

	C-ID	N-ID	M-ID
HPA	0	0	0
MPA	1	1	1
LPA	2	2	2

Table II. Shared Resource Allocation at Scheduling Time

C-ID	Cache limits	N-ID	Weights(local/global)	M-ID	Weights
0	50%	0	9/36	0	10
1	35%	1	6/24	1	5
2	15%	2	3/12	2	2

Table III. Priority Level Assignments at Runtime

	C-ID	N-ID	M-ID
HPA	0	1	1
MPA	1	0	0
LPA	2	2	2

Table IV. Shared Resource Allocation at Runtime

C-ID	Cache limits	N-ID	Weights(local/global)	M-ID	Weights
0	40%	0	9/36	0	10
1	42%	1	6/24	1	5
2	18%	2	3/12	2	2

performance predicted for the next time window, by adjusting N-ID, M-ID and cache limits. The goal of our dynamic QoS management framework is to maintain the performance of HPA while improving the performance for MPA and LPA, by stealing resources from HPA for other applications whenever possible. We next introduce these components in more detail.

After the QoS assignment and mapping at scheduling time, the performance of each application is monitored periodically. This is performed by the hardware performance monitor. At the end of each time interval, the performance monitor summarizes the performance attributes recorded in the current time window, the average performance attributes recorded till current time window. It then calculates the exponential weighted average performance as the performance prediction for the next time window. After the calculation, the performance monitor sends all the recorded and calculated information to the global resource management model.

Here, we introduce the performance attributes needed for our dynamic resource management policy: (1) Instructions per Cycle (*IPC*) (This is the overall application performance metric (*OPM*)), (2) L1 miss rate (*L1MISS*), (3) L2 miss rate (*L2MISS*), (4) average NoC latency spent in the interconnection network (*NOCLAT*) and (5) memory bandwidth utilization rate (*MEMBW*). Each attribute is collected for the current time window (*current*), as well as collected over the application runtime (*avg*) by hardware counters. For each attribute, the exponential weighted average is also calculated as the prediction for the next time interval. Note that the information is collected and calculated for each priority class (HPA, MPA and LPA).

After collecting the performance attributes at the end of each time window (*current* and *avg*) for each priority class (HPA, MPA and LPA), the hardware performance manager then predicts the performance for the next time window for each priority class. The prediction is calculated using the exponential weighted average (*ewa*) to take into account both current and past performance attributes. This allows the manager

to filter out transient fluctuations when making decisions.

$$F_{ewa}(W, V) = \frac{W \times V_{current} + V_{past}}{W + 1}, \quad (1)$$

where F_{ewa} is the predicted performance metric for the next time window using exponential weighted average, W is the weight used for exponential weighted average calculation, V is the performance attribute mentioned before, $V_{current}$ is the average performance metric recorded in the current time window, and V_{past} is the exponential weighted average calculated in the previous time window. Note that we have four pieces of information for each attribute: *current*, *avg*, *ewa*, *past*. *current* is the attribute recorded in the current time window by hardware counter; *avg* is the attribute collected from the beginning of the application runtime till the current time window, capturing long term program behavior; *ewa* is the predicted performance for the next time window, with its weight set to reflect short-term program behavior; and *past* is previous time window's *ewa*. The exponential weighted average *ewa* for each performance attribute can be expressed as follows:

$$IPC_{ewa} = F(W_{perf}, IPC) \quad (2)$$

$$L1MISS_{ewa} = F(W_{cache}, L1MISS) \quad (3)$$

$$L2MISS_{ewa} = F(W_{cache}, L2MISS) \quad (4)$$

$$NOCLAT_{ewa} = F(W_{noc}, NOCLAT) \quad (5)$$

$$MEMBW_{ewa} = F(W_{mem}, MEMBW), \quad (6)$$

where W_{perf} is the weight for performance, W_{cache} is the weight for cache, W_{noc} is the weight for NoC, and W_{mem} is the weight for memory. These equations calculate the predicted performance metric for the next time window (*ewa*) based on *current* and *past*. For example, IPC_{ewa} is the predicted performance for the next time window, it is calculated as $F(W_{perf}, IPC) = (W_{perf} \times IPC_{current} + IPC_{past}) / W_{perf} + 1$. Here $IPC_{current}$ is the performance recorded in the current time window, and IPC_{past} is the predicted IPC performance in the previous history window.

After the hardware performance manager collects and calculates these different performance attributes, the twenty information from each priority class are then sent to the global resource manager at the end of each time window.

5. GLOBAL RESOURCE MANAGER

From the hardware performance attributes collected and calculated by the hardware performance manager, the global resource manager then makes decisions about shared resource allocation for the next time window. The rationale behind our dynamic resource management is as follows: we monitor the different performance attributes of the HPA to track if its resource usage is currently low. If so, we steal some resources from the HPA and distribute them to the lower-priority applications. If this resource stealing affects the performance of HPA, we reset the resource allocation to that of the previous time window or reset it to the default allocation at scheduling time. If the resource stealing does not affect the performance of HPA, we continue to steal resources from HPA and reallocate them to other applications until this affects the performance of HPA or until HPA moves to a phase that requires more resources. We then reset the resource allocation to the previous time window or to the default allocation determined at scheduling time.

As mentioned, we use IPC as the overall performance metric. However, it is difficult to specify a single IPC to target across the entire application runtime, as many programs experience phase behavior during execution, so there can be a significant variance

in their IPC. Cazorla et al. [2004] observed that if the local target performance (the performance of HPA if it was allocated its full resource budget in recent time windows) can be achieved, then the final overall performance for a given job can be realized as well.

In our article, we apply the concept in Cazorla et al. [2004] in our study. Since we preallocate resources for each priority class at scheduling time, our HPA's local performance target is its IPC performance when given its full share of the resources. During dynamic resource allocation, we then aim to maintain the IPC of HPA as close as possible to this local target performance. We expect that if the local target performance can be achieved during dynamic resource allocation periods, then the final target performance can be achieved as well. In order to achieve this goal, we employ two phases modified from the mechanisms proposed in Cazorla et al. [2004].

To ascertain the local target performance for HPA, we use a *sample phase*, when we reset the resource allocation to the amount preassigned at scheduling time. We then profile and record the IPC for HPA during this sample phase. This is HPA's local performance target, IPC_{ref} . Thereafter, during the *dynamic resource allocation phase*, we dynamically allocate HPA's resources to lower-priority applications in the hope of improving the performance for lower-priority applications while maintaining the local targeted performance IPC_{ref} for HPA.

The flow diagram for our dynamic global resource management can be found in Figure 2, and a basic description of our proposed global resource management heuristic is shown in Figure 3. In the next two subsections, we discuss the *sample phase* and *dynamic resource allocation phase* in more detail.

5.1. Sample Phase

To reduce the interference among applications sharing resources, we employ a solution similar to Cazorla et al. [2004]. We assume each time window is Tw clock cycles. In the first time window of the sample phase (Tw clock cycles), we simply reset HPA to all its default allocated resources, but do not record its performance. Once the system is warmed up, in the second time window, we record the performance for HPA as a reference (IPC_{ref}) that is used in the *dynamic resource allocation phase*. Essentially, IPC_{ref} is the $IPC_{current}$ collected in the second time window of the sampling phase. Note that during this sample phase, all applications are still running simultaneously; only the resource allocations are reset to that when applications were first launched. The basic steps in *sample phase* are shown in Figure 3.

5.2. Dynamic Resource Allocation Phase

After every *sample phase* of two time windows, there follows the *dynamic resource allocation phase* of 30 time windows where we try to steal resources from HPA to MPA and LPA while maintaining the local target performance IPC_{ref} for HPA measured in the previous *sample phase*. We tested different time windows and arrived at 30 unit as a good trade-off.

In our mechanism, the dynamic resource allocation phase is composed of two steps. In the first coarse-grained dynamic resource allocation step, we decide whether to perform dynamic resource allocation for the next time window based on the current, predicted and averaged performance attributes of the HPA and assign an initial allocation for cache limits, NoC priority level and memory priority level. In the second step, we fine-tune the cache allocation amount to be stolen from the HPA based on the reference target (IPC_{ref}) obtained in the *sample phase*. This step critically ensures that the HPA's performance is not impacted too severely. The basic description for the *dynamic resource allocation phase* is shown in Figure 3 and more details follow.

collect $IPC_{current}$ for HPA at second time interval set $IPC_{ref} = IPC_{current}[HPA]$

Step 1: Coarse-grained resource allocation

```

cache:
if (  $L2MISS_{eval}[HPA] > L2MISS_{avg}[HPA]$  ) {
    restore to default settings at scheduling time
}
else if (  $L2MISS_{current}[HPA] > L2MISS_{past}[HPA]$  ) {
    roll back to previous time window cache allocation
}
else if (  $L2MISS_{eval}[HPA] < L2MISS_{avg}[HPA]$  and  $L2MISS_{avg}[HPA] < L2MISS_{avg}[MPA]$  ) {
    pre-allocate 5% as steal_amount from HPA, will fine tune in the PI control step next
    cache_steal = true
}

```

```

if (  $L1MISS_{avg}[HPA] < L1MISS_{avg}[MPA]$  and  $L1MISS_{avg}[MPA] < L1MISS_{avg}[MPA]$  and
 $NOCLAT_{avg}[HPA] < NOCLAT_{avg}[MPA]$  ) {
     $N-ID[HPA] = 1$  //set to medium service level
     $N-ID[MPA] = 0$  //set to high service level
}
else {
    reset to default settings
}

```

```

if ( $L2MISS_{ewa}[HPA] < L2MISS_{avgl}[HPA]$  and  $L2MISS_{avgl}[HPA] < L2MISS_{avgl}[MPA]$ 
and  $MEMBW_{ewa}[HPA] < MEMBW_{avgl}[HPA]$ ) {
     $M-ID[HPA] = 1$  //set to medium service level
     $M-ID[MPA] = 0$  //set to high service level
}
else {
    reset to default settings
}

```

```

if (cache_steal == true in coarse-grained resource allocation phase) {
    Δcache_tune = output from PI controller
    steal_amount[HPA] = pre_allocation_amount + Δcache_tune
    cache_next[HPA] = cache_current[HPA] - steal_amount[HPA]
    cache_next[MPA] = cache_current[MPA] + steal_amount[HPA] * split_ratio / (1 + split_ratio)
    cache_next[LPA] = cache_current[LPA] + steal_amount[HPA] / (1 + split_ratio)
    set cache_next as cache_current
    set cache_steal = false
}

```

5.2.1. Coarse-Grained Dynamic Resource Management Policies. During the coarse-grained dynamic resource allocation step, we first check if HPA’s predicted IPC (IPC_{ewa}) for the next time window is higher than its average IPC recorded so far (IPC_{avg}). We use this as an indication that HPA’s resource requirements will dip in the next time window, and as a trigger to allocate HPA’s resources to MPA and LPA applications. The shared resources in our scheme are L2 cache capacity, NoC bandwidth and memory bandwidth. We next discuss each one in more detail. Note that for each shared resource, the decision

is not made using only its own resource information, but includes its interactions with other shared resources so that the decision can be made in a coordinated fashion.

Cache capacity reallocation. We use L2 miss rate as an indication of the application's cache demands. If L2 miss rate is low, we expect that program's working set fits into the L2 cache right now and its demands on the cache is low. We also compare the L2 miss rate between HPA and MPA. If the L2 miss rate for HPA is lower than L2 miss rate for MPA, we expect that MPA has higher demands on cache than HPA, we then steal cache capacity from HPA and allocate to MPA and LPA in the hope that this can improve performance for lower-priority applications. If the L2 miss rate for HPA is similar to or higher than MPA, this means that MPA's demand on cache is low as well. So we do not steal cache from HPA because this may not benefit MPA.

The granularity at which the cache space is stolen from HPA is prespecified in this step (*pre_allocation_amount*, 5% of HPA's current cache usage) and is fine-tuned by the formal controller in the next step. If this stealing causes the L2 miss rate for HPA to increase significantly in the next time window, we then reset the cache allocation to the previous time window or reset back to the default value at scheduling time. This reallocation can be done by modifying the mapping table to specify the cache capacity for each priority class at L2 banks. The basic description of cache space reallocation is shown in Figure 3.

NoC priority level adjustment. For interconnection networks, we keep the weights for each N-ID unchanged, but dynamically assign different N-IDs to different applications based on their performance behavior in the recent history windows and the prediction for the next time window. The reason why weights are left unmodified is that when the bandwidth allocated for HPA is left idle, it is automatically used by other applications if they have requests. As a result, reassigning weights does not significantly improve the performance of lower-priority applications. By changing the N-IDs instead, the requests from MPA and LPA can be served ahead of the HPA.

We allocate the NoC priority level for each application based on two indicators: network latency and L1 Miss rate. L1 miss rate is an important factor for network traffic, as L1 misses lead to communications within the network to the shared L2 cache and can trigger L2 misses that lead to further communications to off-chip memories. It is used as a secondary indicator, whereby N-ID priority level is only adjusted if the network latency for HPA is predicted to be low in the next time window. The basic description for the proposed N-ID reassignment decision is shown in Figure 3.

Memory priority level adjustment. Similar to the NoC, we keep the weights for each M-ID unchanged at runtime, and instead, change the assignment of M-IDs to applications. The reason is similar: unused memory bandwidth is automatically given to lower-priority applications, hence changing the priority levels have a more significant impact.

For each application, we use the memory bandwidth utilization rate as well as the L2 miss rate as the indicators for memory bandwidth prediction. We adjust M-ID priority level only when we predict the memory bandwidth usage for HPA will be low in the next time window. The basic description for the proposed M-ID reassignment decision is shown in Figure 3.

In summary, here, the decision for each shared resource is not only based on individual resource information, but also on global information as well as the interaction with other shared resources to ensure that overall performance degradation for HPA is minimized. The decisions for NoC and memory priority-level assignment in this step is the final decision and will be updated for the next time window. For the cache, the final decision will be based on the fine-tuning process that will be described next.

5.2.2. Applying Formal Control to Fine Tuning Cache Capacity Reallocation: PI Controller design. The goal in this fine-tuning process is to keep the performance of HPA as close as

Fig. 4. General PI-control system.

Fig. 5. Feedback control block diagram for PI-control system.

possible to the reference performance (IPC_{ref}) collected in the *sample phase*. In order to achieve this goal, we use a control-theoretic approach to determine the appropriate cache size allocation. For the NoC and memory, we retain the decisions made in the *coarse-grained dynamic resource allocation step* because we only adjust the priority levels for the NoC and memory and keep the weights unchanged. As a result, we only fine-tune the cache capacity in this step.

When designing our fine-grained cache allocation controller, we apply closed-loop control theory [Kuo and Golnaraghi 2003]. Closed-loop control is a popular method to control complex systems so that the controlled value rapidly converges to the desired target value. Figure 4 illustrates a commonly used feedback controller: the proportional-integral (PI) controller. Another widely used controller is proportional-integral-derivative (PID) controller, but we found that a PI controller is relatively simple to design and adequate for our system. In Figure 4, R is the reference signal, M is the measured feedback signal. The error signal e is the difference between the measured signal and the reference signal. And e is the input to the PI controller. The controller attempts to minimize the error between M and R by calculating and outputting a control signal U that drives the controlled system accordingly and rapidly. The standard PI controller equation can be described as follows:

$$\Delta U_k = K_p(e_k - e_{k-1}) + K_i e_k, \quad (7)$$

where K_p is the proportional gain which determines the reaction to the current error, K_i is the integral gain which determines the reaction based on recent errors, e_k is the current error signal, e_{k-1} is the previous error signal, ΔU_k is the adjusted amount for the prior control value that will be applied to the system.

There have been numerous works using formal feedback control in architecture and systems [Donald and Martonosi 2006; Wu et al. 2005; Suh and Dubois 2009; Skadron et al. 2002; Varma et al. 2003] recently. Most works use formal feedback control for dynamic voltage and frequency scaling control [Donald and Martonosi 2006; Wu et al. 2005; Suh and Dubois 2009; Varma et al. 2003], whereas we apply it for dynamic cache capacity allocation.

Figure 5 shows the control block diagram we used in our system. In our system, the reference signal R_{ref} is the IPC (IPC_{ref}) determined for HPA during the *sample phase*. The input to the PI controller e_k is the normalized difference between $IPC_{current}$ and IPC_{ref} for HPA at the end of each time window, as shown in Equation (8). Here, we use normalized performance difference as the error signal because the absolute difference between $IPC_{current}$ and IPC_{ref} can vary substantially across different applications. The

output from the PI controller is the adjusted cache percentage amount that needs to be applied on top of the allocated amount prespecified in coarse-grained allocation step as shown in Equation (9).

$$e_k = \frac{IPC_{current} - IPC_{ref}}{IPC_{ref}} \quad (8)$$

$$\Delta cache_tune_k = K_p(e_k - e_{k-1}) + K_i e_k. \quad (9)$$

The final cache amount to be stolen from HPA (*steal_amount*) is the sum of $\Delta cache_tune_k$ and the prespecified cache stolen amount (*pre_allocation_amount*) in coarse-grained allocation phase. Given this cache allocation amount that is to be stolen from HPA, a split-ratio parameter is needed to indicate how these resources will be distributed between MPA and LPA. In our scenario, we use the cache allocation ratio of MPA and LPA at scheduling time as the split ratio. The basic description for the proposed fine-grained cache size adjustment by PI-controller is shown in Figure 3.

Note that this PI-controller only applied to HPA. By tuning the two PI gains in Equation (9), the controller can provide control action designed for the performance requirements. If the controlled system can be expressed in mathematical format, then PI gains can be determined mathematically with the aid of control theory. When the controlled system cannot be specified mathematically, a *trial-and-error* approach is commonly used to tune the gains. In our study, the system performance depends on the cache, NoC, and memory systems, as well as the behavior of the applications. Hence, the characteristic of the CMP system cannot be specified mathematically. Thus, we use a *trial-and-error* method to determine the two PI gains in our system.

The *trial-and-error* method to tune K_p and K_i can be summarized as follows: (1) select K_p value with $K_i = 0$. Given the input and output range of the PI controller (most of the times within $[-1, 1]$ for input, within $[-0.05, 0.05]$ for the output in our system), select different values for K_p and monitor the response of the system. A high K_p will result in system oscillation, while a low K_p has too little impact on the system. In our PI controller, because input is the normalized performance difference rather than absolute difference, the input does not vary a lot across different applications. Thus, the optimal K_p lies in a small range for the different applications running on the system. (2) select K_i value. Here, we retained the K_p value as selected in (1) and vary the K_i value. K_i helps accelerate the convergence time to the reference value. A high K_i will cause the system to be unstable while a low K_i cannot converge to the reference value quickly. We tested different K_i values and selected the one that gives a quicker response time while keeping the performance closer to the reference (IPC_{ref}). Following this method, the final values we chose were $K_p = 0.1$ and $K_i = 0.05$. Note that these values are not unique, nor optimal. However, because of our normalized input and small output range, the optimal K_p and K_i lie in a very small range close to 0.1 and 0.05 and work well across a wide range of applications in the system. This also illustrates the robustness of control-theoretic techniques.

During the coarse-grained dynamic resource management step, if we decide to decrease the cache size for HPA, we use the given PI controller to fine-tune how much cache to steal from HPA. This control step is to ensure that the performance of HPA is kept close to its reference value. Note that the final cache allocation amount for HPA has an upper bound and lower bound in our system. The upper bound is the allocated cache size at scheduling time. This is to ensure that HPA will not get more than its allocated cache amount at scheduling time when our PI controller is applied. We also set a lower bound for cache size for HPA. This lower bound is the minimum cache size that HPA needs to ensure its performance. We empirically chose 20% of the allocated cache size at scheduling time for HPA as the lower bound.

After the coarse-grained analysis and fine-grained control based on the information collected for each application, if new allocations are determined for the next time window, the updated cache limits, N-IDs, and M-IDs will be written to the table in the global resource manager, triggering the passing down of these limits to each L2 cache bank, NoC and memory mapping registers. Each message injected into the network will also be attached with a new N-ID number during the next time window.

6. IMPLEMENTATION

In this section, we describe the hardware implementation and the OS support needed for our proposal.

6.1. Hardware Implementation

The runtime dynamic resource manager is composed of two components: (1) the hardware performance monitor and (2) the global resource manager. We introduce the implementation of these two components in detail.

6.1.1. Hardware Performance Monitor Implementation. For performance-related information, we rely upon on-chip performance monitoring counter hardware. In most modern processors, performance counters are provided to allow monitoring of specific hardware events for system tuning or debugging. For example, the Intel Nehalem architecture supports different hardware performance counters to count hundreds of different hardware events [Intel Corporation 2009]. In our framework, the attributes that need to be calculated are the performance in terms of IPC (*IPC*), L1 miss rate (*L1MISS*), L2 miss rate (*L2MISS*), memory bandwidth utilization rate (*MEMBW*) and NoC latency (*NOCLAT*). Most of these can be calculated with existing provided hardware performance counters. For example, the performance counters in Intel Core i7 processor [Intel Corporation 2009] can be used to calculate the L1 miss rate, L2 miss rate, memory utilization rate and IPC (we assume these performance counters are recorded for each priority class when QoS is supported in future architectures). However, the NoC performance counters that track NoC metrics such as packet latency are not yet available commercially. We thus propose an implementation for collecting and tracking packet latency. Network traffic is the combined flow from different sources, such as data requests, data responses, acknowledgements, etc. Recording the NoC latency for each packet will have prohibitively high overhead. To simplify the implementation, we randomly sample traffic from each source router for NoC latency calculation. This is realized by randomly selecting packets at network injection queues at the network interface, and padding them with one more flit storing the current global clock time at the source at injection. When this packet arrives at the destination, the latency for this packet is calculated by subtracting the global clock at the destination with the injection time encoded into the packet. Two registers are needed at each destination router: one to count the accumulated latency for randomly selected packets, the other to count the number of packets recorded. The average NoC latency can thus be calculated by dividing the accumulated latency by the number of packets recorded.

At the end of each time window, the hardware performance monitor collects the performance event counts for the current sampling period and converts these event counts into the *current* metric for each attribute. The performance monitor also calculates the accumulated event counts for each event by adding the performance event counts collected for the current sampling period to the recorded accumulated event counts for all previous time windows. It then converts these accumulated event counts into the *avg* metric for each attribute. All event counters are set to zero at the end of each time window to be ready for the next time window. We assume *current* and *avg* attributes calculations are supported in current or future architectures. Hence, the hardware we

need to add is the circuit to calculate the prediction for the next time window ewa where $F_{ewa}(W, V) = (W \times V_{current} + V_{past}) / (W + 1)$. We set weight W to be $2^k - 1$ so that division can be implemented as a shift operation (k is a constant). Thus, the predictor can be implemented by one multiplier, one adder and one shifter. Since we have five performance attributes to predict, they can be calculated in a pipelined fashion with a single circuit component.

6.1.2. Global Resource Manager Implementation. The major circuit in the global resource manager is the PI controller. To implement the PI controller, we need to first calculate the normalized performance difference as the input to the PI controller. This can be implemented with one adder and one divider. To calculate the PI output, we need two multipliers and two adders which is similar to the implementation in Skadron et al. [2002]. In addition to the PI controller, the global resource manager also requires some combinational logic for comparisons to make the decision. Besides, storage is needed to record various attributes. This includes the twenty attributes calculated by the hardware performance monitor. Assume each attribute is 16 bits.² This requires 120B of storage (20 attributes for each priority class, three priority classes in our case). We also need to record the decisions made at the end of each time window for roll-back. This requires 2 bits to record each priority level for NoC and memory decisions and 7 bits to encode the cache capacity as percentage between 0 and 100. Hence, this consumes less than 5B in total for three priority levels.

6.2. Power and Area Overhead

As we explained previously, the hardware performance monitor and global resource manager can be implemented with four adders, three multipliers, one divider, one shifter, some combinational logic, and less than 150B storage. Bitirgen et al. [Bitirgen et al. 2008] estimated the power for one 16-bit fixed-point multiplier to be 32 mW at 65nm technology, with an area of 0.057 mm². In our experiments, we applied this estimation for the multipliers as well. Thus, power for the three multipliers is 96 mW while area is 0.171 mm². We use ORION 2.0 [Kahng et al. 2009] to estimate the power and area for the 150B storage (assuming SRAM-based storage). The estimated power for 150B storage is 61 mW and the estimated area is 0.342 mm². We assume the power and area for a divider are of the same order as a multiplier, while an adder, shifter, and comparator consume much less power and area than a multiplier. Hence, we estimate the power for the four adders, one divider, one shifter and comparators to be within 100 mW and the area to be within 0.2 mm². Thus, the total power consumption of our proposed hardware (including both computation and storage) is estimated to be within 300 mW and the total area is estimated to be within 1 mm². Assuming the total power for a 16-core chip to be 60 W, the total power overhead of our proposal is about 0.5% which is negligible. Also, the total area overhead of our proposal is about 0.5% for a 200 mm² chip.

6.3. Delay Overhead

In our proposal, we assume 4 cycles for multiplication, 1 cycle for add, 8 cycles for divide, and 1 cycle for shifter. Hence, our decision can be made and take effect within 300 cycles (20 cycles to make the performance prediction, 15 cycles for the PI controller, and assume within 200 cycles round-trip network latency in a 4 × 4 network to collect performance attributes and send out new control parameters through the network,

²The largest number of attributes should be average network latency. For a 4 × 4 network with five pipeline stages, average network latency should be within 100 cycles most of the time. Thus, 16 bits are sufficient for recording the various attributes.

along with some margins). As a result, our decision can be made in a timely manner (within 1% for a 100K time interval window). Note that the applications continue to run while the decision is made. Hence, the decision making time overhead does not affect the performance of executing applications.

6.4. Software Implementation: OS Interface

In our implementation, we expect the user or administrator to supply the required QoS information (such as the priority level, the amount of resources required, etc.) to the OS. The OS maps the information to the corresponding priority class. The OS also allocates to each priority service class appropriate amounts of shared resources in terms of limits or weights that will be mapped onto cache, NoC and memory. Our scheme enables the OS to decide if a priority class consists of a thread, a process, an application, a group of applications, or another well-defined group. If the priority class consists of multiple applications, the allocated shared resources for this priority class is shared by all applications running within this priority class. The OS also controls the number of applications to run on the system simultaneously based on the required resources to make sure the performance can be satisfied by the system [Guo et al. 2007]. The OS also decides whether to enable dynamic resource management. If the HPA has hard real-time requirements, the OS can disable the dynamic resource manager, that is, the shared resource allocation will be static during execution time. If the HPA can tolerate a small performance degradation, the OS enables dynamic resource management and the shared resource is dynamically adjusted by our global resource manager during execution. At runtime, if dynamic resource management is enabled, our global resource manager dynamically updates the resource allocation information. The updated information is also sent to the OS whenever changes are made so that the OS is aware of the modification by the hardware.

6.5. Implementation Comparison

In Section 2, we discussed that Bitirgen et al. [2008] proposed an approach based on artificial neural network (ANN) to jointly manage multiple on-chip resources to improve system-level performance. Here, we compare the differences between Bitirgen's ANN work [Bitirgen et al. 2008] and our work: (1) in the ANN work, the authors consider a 4-core CMP system, while in our work, we consider a much larger system with 16-core CMP; (2) in the ANN work, the authors consider a single shared L2 cache, while in our work, we consider a shared distributed L2 cache; (3) there is no on-chip interconnection network in the ANN work, while we consider a 16-router mesh network with detailed contention modeled; (4) in the ANN work, the authors use ANN for resource allocation, while we use a PI control based technique for resource allocation; (5) the QoS goal in the ANN work is to maximize total system performance, while the QoS goal in our work is to provide differentiated performance to each individual application; (6) resources controlled in the ANN work are cache, memory, and power, while resources controlled in our work are cache, NoC and memory. The method proposed in the ANN work can potentially be expanded to solve the problems presented in this article. In order to do so, the ANN method in the ANN work needs to be extended to model NoC resource allocation first. This can be done by adding NoC latency and router weights as inputs to the ANN model. The QoS goal in the ANN work also needs to be expanded to provide differentiated performance for each application while improving system performance with the proposed ANN approach. To maintain the performance of HPA while improving the runtime of lower-priority applications as in our design, the work in Bitirgen et al. [2008] can be modified as follows: At each sample phase, they can collect HPA's performance as the reference. During the dynamic resource allocation phase, they can search the allocation space to find the minimum resource allocation

Fig. 6. Overview of CMP architecture.

required for HPA that delivers the output performance equivalent to the reference performance.

To support ANN, the proposal in the ANN work requires 52 multipliers, four dividers, several adders, and 9KB of storage for hardware implementation. If network latency and NoC weights are added, the hardware cost will be even higher. In our proposal, we require four adders, three multipliers, one divider, one shifter, some combinational logic, and less than 150B storage for implementation. Even compared to the current ANN model, our hardware cost is only about 7% of the ANN cost in the ANN work. Hence, our proposal is much more cost-effective. Furthermore, it takes 25,000 cycles to process the query for each control point in the ANN work. In our proposal, decisions can be made and effected within 300 cycles.

7. EVALUATION

In this section, we first present our simulation framework as well as workloads used for our evaluation, we then discuss the simulation results.

7.1. Simulation Framework

For our evaluations, we developed a detailed trace-driven cycle-accurate simulator based on ASPEN [Moses et al. 2004] and GARNET [Agarwal et al. 2009]. ASPEN is a trace-driven multi-core, multi-threaded platform simulator. It models generic out-of-order processor cores, a detailed cache hierarchy, coherence protocol and memory subsystem. However, ASPEN lacked a detailed interconnect model. To address this issue, we integrated GARNET into ASPEN. GARNET is a detailed interconnection network model that models detailed packet-switched network. It allows different router parameters such as router pipeline depth, buffer sizes, virtual channels, etc. We integrated ASPEN and GARNET to form a detailed performance simulator for CMP.

Figure 6 presents an overview of a typical CMP architecture with shared L2 cache expected for future high-performance computing platform. We use this CMP architecture in our evaluation. We simulated a tile based 16-core CMP system and the parameters of our simulation are given in Table V. Each tile consists of a processing core, a 16KB private L1 cache and a 256KB shared distributed L2 cache. So the total L2 cache is 4MB. The on-chip network was a 4×4 mesh topology with deterministic X-Y routing. DRAM was attached to the CMP through two memory controllers at two corners of the chip (one at upper left, one at lower right of the chip).

Table V. Simulation Parameters

L1 cache	16KB, 2-way set associative, 64-byte line, 1-cycle latency
L2 cache	256KB per node, 8-way set associative, 64-byte line, 5-cycle latency
Coherence Protocol	Directory based MESI
NoC	4 × 4 2D mesh, 4 router pipeline stages, 6 virtual channels(2 per priority class), 5 buffers per virtual channel, 1 cycle on-chip link latency, X-Y routing
Memory	2 memory controllers, 4.2GB/s maximum sustainable memory bandwidth each, 100 cycles access time

Table VI. Programs Chosen from Different Benchmarks

category	program
PARSEC	fluidanimate, dedup, bodytrack, vips, canneal, streamcluster, ferret, x264
Server	sjbb, sjas, tpc, sap
SPEC2006	HPA (SPEC_H): sphinx, leslie3d, GemsFDTD, milc, libquantum, lbm MPA (SPEC_M): gcc.expr2, gobmk, gcc.c-typeck, soplex.pds-50, soplex.ref, xalancbmk LPA (SPEC_L): mcf, gcc.166, dealII, perlbench

Table VII. Simulated Applications

	group 1	group 2	group 3	group 4	group 5	group 6	group 7	group 8	group 9	group 10	group 11
HPA	fluidanimate	dedup	bodytrack	vips	fluidanimate	dedup	SPEC_H	SPEC_H	vips	sjas	canneal
MPA	sjbb	tpc	sap	sap	sap	sjas	tpc	sjbb	SPEC_M	SPEC_M	tpc
LPA	canneal	ferret	fluidanimate	canneal	SPEC_L	SPEC_L	canneal	dedup	tpc	x264	streamcluster

7.2. Workloads

We ran different application combinations on the aforementioned CMP platform. The applications are chosen from PARSEC benchmarks [Bienia et al. 2008], server benchmarks [Standard Performance Evaluation Corporation. 2005, 2004 Sap America Inc. 2008; TPC. 2009] and SPEC2006 benchmarks [Standard Performance Evaluation Corporation. 2006]. We selected applications from different benchmarks to reflect a diversity of applications. The PARSEC benchmark suite focuses on emerging parallel workloads for CMPs, the server benchmarks are multi-threaded commercial workloads (TPC-C [TPC. 2009], SAP [Sap America Inc. 2008], SPECjbb2005 [Standard Performance Evaluation Corporation. 2005] and SPECjappserver2004 [Standard Performance Evaluation Corporation. 2004]), while the SPEC2006 benchmarks are sequential workload benchmarks. The PARSEC, SPEC2006 and server benchmark traces were all collected from real machine runs. The traces collected are Rich Workload Trace (RWT) format as introduced in Moses et al. [2004].

In each experiment, three applications (or application groups) run simultaneously on the platform and are assigned as HPA, MPA and LPA class, respectively. For PARSEC and server benchmarks, since they are parallel, multi-threaded benchmarks, each application runs on several cores simultaneously. For SPEC2006 benchmarks, we group several applications together to form the different priority groups: HPA(SPEC_H), MPA(SPEC_M) and LPA(SPEC_L) and each allocated core runs one program. Table VI lists the programs selected from each benchmark. Table VII shows the different applications used in our evaluation. In our experiments, we assume HPA applications were assigned to tiles 0 to 5 (6 cores), MPA applications to tiles 6 to 11 (6 cores), and LPA applications to tiles 12 to 15 (4 cores), as shown in Figure 6. Other mappings can be configured as well. At scheduling time, the OS assigns the HPA the highest service level of all resources, as shown in Table VIII. The OS also assigns the resources allocated for each priority service class, as shown in Table IX, at scheduling time. These assignments were derived empirically to represent the best static assignments for the applications. It forms our baseline static QoS scheme. Our dynamic resource management policy is

Table VIII. Priority Levels Initial Assignments

	CoS-ID	C-ID	N-ID	M-ID
HPA	0	0	0	0
MPA	1	1	1	1
LPA	2	2	2	2

Table IX. Shared Resource Allocation

C-ID	Cache limits	N-ID	Weights(local/global)	M-ID	Weights
0	60%	0	12/48	0	8
1	30%	1	6/24	1	4
2	10%	2	2/8	2	2

Table X. Parameters for Different Weights Used in Calculation of Exponential Weighted Averages (*ewa*)

W_{cache}	3
W_{noc}	3
W_{mem}	3
W_{perf}	3

initialized with these allocations, with the global resource manager then dynamically adjusting them during execution. The time window used in the experiments is 100K cycles, determined empirically. The different weight parameters of our policy are given in Table X. In each experiment, we run the applications until HPA finish executing 1 billion instructions. We use IPC as the metric for evaluation.

7.3. Experiment Results

In this section, we present experimental results to illustrate the effectiveness of the proposed dynamic resource management strategy.

7.3.1. Evaluating Dynamic QoS Management Impacts. Figure 7 shows the performance in terms of IPC for each application group with our proposed dynamic QoS management scheme compared to the baseline static QoS scheme. The first bar in each group shows the IPC when only static QoS configuration is applied. The second bar in each group shows the IPC with the proposed dynamic QoS management policy. The results are normalized to the static QoS scheme. From these experiments, we can see that with the dynamic QoS management scheme, the IPC for MPA improves by 12.5% on an average (up to 26.6%) and improves by 4.3% for LPA on an average (up to 17.6%) compared to the static resource allocation configuration across all benchmark groups. For HPA, we aim to keep the performance degradation small, and the results reflect that: the IPC is reduced by only 1.9% on an average compared to static QoS, and the maximum degradation is only 3.8%. These results demonstrate the effectiveness of the proposed dynamic resource management policy for improving the performance for MPA and LPA while maintaining the performance for HPA.

From Figure 7, we notice that for workload group 10 (*sjas*, *SPEC_M*, *x264*) and group 11 (*cannal*, *tpc*, *streamcluster*), the dynamic resource management impact on MPA and LPA is small. This is because with workloads *sjas* and *cannal* as HPA, their *current* program metrics fluctuate around *avg* and do not exhibit obvious phase changes during execution time. Hence, their resource allocation to lower-priority applications can only be done for short time periods each time, leading to lower benefits to other applications. However, we can see that even though the dynamic QoS management policy cannot further improve the performance of lower-priority applications in these two cases, it can still maintain their performance.

We also measure overall system performance using weighted speedup metric, which is commonly used multiprogram performance metric. This speedup metric is based on

Fig. 7. Dynamic QoS management performance comparison with static QoS scheme. Results normalized to static QoS scheme.

Fig. 8. Weighted speedup comparison.

comparing the IPC of an application versus when it runs together with other applications. $Weighted\ speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$. Figure 8 shows the effects of our dynamic QoS policy on overall system performance over the base line static QoS policy. From Figure 8, we can see that dynamic QoS can further improves system speedup by 3.9% on average over baseline static QoS scheme. And in some workloads (e.g., group 5), it can improves system speedup by 12.7%. This improvement is because the proposed dynamic QoS policy can improve performance for MPA and LPA while maintain the performance for HPA.

Fig. 9. Effects of PI controller for performance control of HPA.

In summary, for HPA applications, if their requirements on shared resources vary significantly during execution, our proposed dynamic resource management policy is very effective in maintaining the performance for HPA while improving the performance for other lower-priority applications running concurrently on the platform. In situations where HPA does not have dynamic requirements on shared resources, our proposed strategy can still maintain the performance of each application as in static QoS configuration. This illustrates the effectiveness and robustness of our proposed dynamic resource management strategy.

7.3.2. Effects of the PI-Controller. We next evaluate the effects of the proposed PI-controller in maintaining the HPA performance in our dynamic resource management framework. We ran two sets of experiments. The first experiment steals a fixed percentage of cache capacity from the HPA (5% of the current cache size in our case), as specified in the coarse-grained resource allocation stage (*No PI control*). The second experiment first allocates an initial cache amount during the coarse-grained resource allocation step (5% of the current cache size), and then fine-tunes this amount in the PI-controlled step to ensure the performance of HPA (*With PI control*). Figure 9 shows the comparison results. The first bar shows the IPC for HPA when there is no PI control to fine-tune the cache space allocation. The second bar shows the IPC for HPA when the PI controller fine-tunes the cache allocation to ensure the performance of HPA. The results are normalized to the static QoS scheme. From the figure, we can see that, without the PI controller, the performance degradation for HPA is 8.6% on an average compared to the static QoS scheme. When using the PI controller to fine-tune the cache space allocation, the performance degradation for HPA is 1.9% on an average and 3.8% maximum in our experiments compared to the static QoS scheme. This demonstrates that the PI controller is highly effective in maintaining the performance for HPA in dynamic resource allocations. For workload group 8, we can see that without PI control, HPA's IPC decreases by 14%, while the PI controller decreases HPA's IPC by only 1.8%. This is because without PI control, we compare IPC_{ewa} with IPC_{avg} and allocates 5% of the cache from HPA to other applications each time. However, this cache allocation may result in more cache misses for HPA, and this in turn decreases its IPC_{avg} . The decrease in IPC_{avg} gives more opportunity for the condition $IPC_{ewa} > IPC_{avg}$ to be true, so that cache space can be stolen from HPA for other applications in more time windows, thus causing a vicious circle in feedback. When the PI controller is used,

Fig. 10. Contribution of each individual dynamic QoS to performance improvement of lower-priority applications.

since it aims to achieve the local target performance IPC_{ref} for HPA, and adjust cache space allocation accordingly, it avoids the accumulated negative feedback for IPC_{avg} .

7.3.3. Resource Sensitivity Analysis. This section studies the contribution of each resource's dynamic QoS management to the overall performance improvements for MPA and LPA applications. We run three experiments where we allow just one resource's allocation to dynamically vary while the other two resources' allocations remain static: dynamic cache QoS only (with static NoC and memory QoS), dynamic NoC QoS only (with static cache and memory QoS), dynamic memory QoS only (with static cache and NoC QoS). Here, we only show the results for workload group 1 (*fluidanimate*, *sjbb* and *canneal*) as an example. Other groups show similar trends.

Figure 10 shows the comparison results. The results are normalized to the static QoS scheme. From Figure 10, we can see that for workload group 1, with dynamic cache QoS only, the IPC for MPA improves 4% compared to the static QoS scheme. With dynamic NoC QoS only, the IPC for MPA improves 8% over the static QoS scheme. With dynamic memory QoS only, the IPC for MPA improves 3% compared to the static QoS scheme. However, the IPC latency for MPA improves by 13% when dynamic cache, NoC and memory QoS work jointly, as proposed in our scheme. From this example, we can see that all the three shared resources (cache, NoC, memory) in our dynamic QoS management framework play an important role in improving the performance of lower-priority applications.

7.3.4. Sensitivity to Time Window Intervals. In this section, we study how different time window intervals affect the performance for our dynamic QoS policy. The global resource manager makes decision about whether to steal resources from HPA and allocate them to MPA and LPA at the end of each time window. While longer time window may miss the opportunities for resource stealing, shorter time window can lead to fluctuation in the system. We run four experiments with time windows vary from 10K cycles, 100K cycles, 1M cycles to 10M cycles. Figure 11 shows the performance comparison with these different time windows. Here, we show the results for workload group 1 (*fluidanimate*, *sjbb* and *canneal*), and other groups have similar trends. From Figure 11, we can see that when the time window is small (10K), even though the performance improvement for MPA and LPA are higher, the performance degradation for HPA is also high, 7.4% in this case. This is because short time window can capture

Fig. 11. Time window sensitivity analysis.

Fig. 12. Weight value sensitivity analysis.

transient fluctuations, thus leading to more opportunities for resource stealing from HPA. On the other hand, when we increase the time window interval, we can see that the performance improvement for MPA and LPA are reduced. This is because longer time window filters out some short term program behaviors, and thus has less opportunities for resource stealing. We can see 100K cycle time window arrives at a good trade-off. The performance improvements for MPA and LPA are fairly good, while the performance degradation for HPA is kept low.

7.3.5. Sensitivity to Weight Value. In this section, we study the effects of different weight values on the performance improvement. In our method, we calculate predicted performance metric using exponential weighted average $F_{ewa}(W, V) = (W \times V_{current} + V_{past}) / (W + 1)$. For larger weight value W , it reflects more on the short term program behavior. For smaller weight value W , it taking more consideration on long term program behavior as well. We run four experiments here. In each experiment, we use the same W value for W_{cache} , W_{noc} , W_{mem} and W_{perf} . We compare performance with the following weight W settings: 1, 3, 7, and 15. Again we show results for workload group 1 (*fluidanimate*, *sjbb* and *canneal*) as an example.

Figure 12 shows the performance comparison with these different weight value settings. From Figure 12, we can see that as we increase the weight value from 1 to 15, the

performance improvement for MPA and LPA keep increasing, while the performance for HPA keep decreasing. This is because larger weight value W reflects more on short term program behavior, thus capturing more opportunities for resource stealing. However, the performance degradation for HPA can be undesirable as well. A good weight value W should balance both short term and long term performance behavior and still keep the performance for HPA. In this experiment study, we find that weight value W 3 is a good trade-off.

8. CONCLUSIONS

In this article, we presented a novel dynamic QoS management framework that dynamically steal resources from HPA and distribute to lower-priority applications. Through a detailed simulation-based evaluation, we showed that the proposed framework is quite effective in maintaining the performance of HPA while improving performance for MPA and LPA whenever possible. We showed that significant performance improvements (12.5% for MPA and 4.3% for LPA on average) can be achieved by dynamic resource allocation while the performance degradation for HPA is minimal (1.9% on average). We also showed that PI controller based cache space fine tuning is very effective in maintaining the performance for HPA. Besides, we showed that all the three shared resources play an important role in achieving higher performance improvement of MPA and LPA in QoS environment and demonstrates the importance of coordinated management of multiple shared resource (cache, NoC, and memory).

REFERENCES

- AGARWAL, N., KRISHNA, T., PEH, L.-S., AND JHA, N. K. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 33–42.
- AKESSON, B. 2010. Predictable and composable system-on-chip memory controllers. Ph.D. thesis, Department of Electrical Engineering, Eindhoven University of Technology.
- AKESSON, B., GOOSSENS, K., AND RINGHOFER, M. 2007. Predator: A predictable SDRAM memory controller. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. IEEE, 251–256.
- BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 72–81.
- BITIRGEN, R., IPEK, E., AND MARTINEZ, J. F. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*. 318–329.
- BJERREGAARD, T. AND SPARSO, J. 2005. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Vol. 2. 1226–1231.
- BOLOTIN, E., CIDON, I., GINOSAR, R., AND KOLODNY, A. 2004. QNoC: QoS architecture and design process for network on chip. *J. Syst. Archit.* 50, 2–3, 105–128.
- CAZORLA, F. J., KNIJNENBURG, P. M., SAKELLARIOU, R., FERNÁNDEZ, E., RAMÍREZ, A., AND VALERO, M. 2004. Predictable performance in SMT processors. In *Proceedings of the 1st Conference on Computing Frontiers*. 433–443.
- DAS, R., MUTLU, O., MOSCIBRODA, T., AND DAS, C. R. 2009. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 280–291.
- DONALD, J. AND MARTONOSI, M. 2006. Techniques for multicore thermal management: Classification and new exploration. *SIGARCH Comput. Archit. News* 34, 2, 78–88.
- EBRAHIMI, E., LEE, C. J., MUTLU, O., AND PATT, Y. N. 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. ACM, New York, NY, 335–346.

- GOOSSENS, K., DIELISSSEN, J., AND RADULESCU, A. 2005. Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test Comput.* 22, 5, 414–421.
- GROT, B., HESTNESS, J., KECKLER, S. W., AND MUTLU, O. 2011. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 401–412.
- GROT, B., KECKLER, S. W., AND MUTLU, O. 2009. Preemptive virtual clock: a flexible, efficient, and cost-effective qos scheme for networks-on-chip. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, NY, 268–279.
- GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. 2007. A framework for providing quality of service in chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 343–355.
- HANSSON, A., GOOSSENS, K., BEKOOL, M., AND HUISKEN, J. 2009. CoMPSoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.* 14, 1, 1–24.
- INTEL CORPORATION. 2009. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Volume 3B: System Programming Guide, Part 2. <http://www.intel.com/Assets/PDF/manual/253669.pdf>.
- IPEK, E., MUTLU, O., MARTÍNEZ, J. F., AND CARUANA, R. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE, 39–50.
- IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. 2007. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM SIGMETRICS Conference*. 25–36.
- KAHNG, A., LI, B., PEH, L.-S., AND SAMADI, K. 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. 423–428.
- KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. 111–122.
- KIM, Y., HAN, D., MUTLU, O., AND HARCHOL-BALTER, M. 2010a. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*. 1–12.
- KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. 2010b. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'43)*. IEEE, 65–76.
- KUO, B. C. AND GOLNARAGHI, F. 2003. *Automatic Control Systems*, 8th ed. John Wiley and Sons, Inc., New York.
- LEE, J. W., NG, M. C., AND ASANOVIC, K. 2008. Globally-synchronized frames for guaranteed quality-of-service in on-chip networks. In *Proceedings of 35th International Symposium on Computer Architecture*. 89–100.
- LI, B., ZHAO, L., IYER, R., PEH, L.-S., LEDDIGE, M., ESPIG, M., LEE, S. E., AND NEWELL, D. 2011. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *J. Paral. Distrib. Comput.* 71, 5, 700–713.
- MILLBERG, M., NILSSON, E., THID, R., AND JANTSCH, A. 2004. Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Vol. 2. 890–895.
- MOSES, J., ILLIKKAL, R., IYER, R., HUGGAHALI, R., AND NEWELL, D. 2004. ASPEN: Towards effective simulation of threads and engines in evolving platforms. In *Proceedings of the 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. 51–58.
- MUTLU, O. AND MOSCIBRODA, T. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 146–160.
- MUTLU, O. AND MOSCIBRODA, T. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, 63–74.
- NESBIT, K., MORETO, M., CAZORLA, F., RAMIREZ, A., VALERO, M., AND SMITH, J. 2008. Multicore resource management. *IEEE Micro* 28, 3, 6–16.
- NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. 2006. Fair queuing memory systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 208–222.
- NESBIT, K. J., LAUDON, J., AND SMITH, J. E. 2007. Virtual private caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 57–68.
- QURESHI, M. K. AND PATT, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, run-time mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 423–432.

- RAFIQUE, N., LIM, W.-T., AND THOTTETHODI, M. 2006. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. 2–12.
- SAP AMERICA INC. 2008. SAP standard benchmarks. <http://www.sap.com/solutions/benchmark/index.epx>.
- SHARIFI, A., SRIKANTIAH, S., MISHRA, A. K., KANDEMIR, M., AND DAS, C. R. 2011. METE: meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.* 39, 13–24.
- SKADRON, K., ABDELZAHER, T., AND STAN, M. R. 2002. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. 17–28.
- SRIKANTIAH, S., KANDEMIR, M., AND WANG, Q. 2009. SHARP control: Controlled shared cache management in chip multiprocessors. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 517–528.
- STANDARD PERFORMANCE EVALUATION CORPORATION. 2004. SPECjAppServer java application server benchmark. <http://www.spec.org/jAppServer>.
- STANDARD PERFORMANCE EVALUATION CORPORATION 2005. SPECjbb2005. <http://www.spec.org/jbb2005>.
- STANDARD PERFORMANCE EVALUATION CORPORATION 2006. SPEC CPU2006. <http://www.spec.org/cup2006>.
- SUH, G. E., DEVADAS, S., AND RUDOLPH, L. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th International Conference on Supercomputing*. 1–12.
- SUH, J. AND DUBOIS, M. 2009. Dynamic MIPS rate stabilization in out-of-order processors. *SIGARCH Comput. Archit. News* 37, 3, 46–56.
- TPC. 2009. TPC-C design document. <http://www.tpc.org/tpcc>.
- VARMA, A., GANESH, B., SEN, M., CHOUDHURY, S. R., SRINIVASAN, L., AND BRUCE, J. 2003. A controltheoretic approach to dynamic voltage scheduling. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. 255–266.
- WEBER, W.-D., CHOU, J., SWARBRICK, I., AND WINGARD, D. 2005. A quality-of-service mechanism for interconnection networks in system-on-chips. In *Proceedings of the Conference on Design, Automation and Test in Europe*, Vol. 2. 1232–1237.
- WU, C.-J. AND MARTONOSI, M. 2011. Adaptive timekeeping replacement: Fine-grained capacity management for shared CMP caches. *ACM Trans. Archit. Code Optim.* 8, 1.
- WU, Q., JUANG, P., MARTONOSI, M., PEH, L.-S., AND CLARK, D. W. 2005. Formal control techniques for power-performance management. *IEEE Micro* 25, 5, 52–62.
- YEH, T. Y. AND REINMAN, G. 2005. Fast and fair: data-stream quality of service. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 237–248.

Received November 2009; revised November 2011; accepted March 2012