Cache Latency Control for Application Fairness or Differentiation in Power-Constrained Chip Multiprocessors

Xiaorui Wang[1], Kai Ma[1], and Yefu Wang[2]
[1]*The Ohio State University, Columbus, OH 43210* [2]*University of Tennessee, Knoxville, TN 37996*

***Abstract*—Limiting the peak power consumption of chip multiprocessors (CMPs) has recently received a lot of attention. In order to enable chip-level power capping, the peak power consumption of last-level (e.g., L2) on-chip caches in a CMP often needs to be constrained by dynamically transitioning selected cache banks into low-power modes. However, dynamic cache resizing for power capping may cause undesired long cache access latencies, and even thread starving and thrashing, for the applications running on the CMP. In this paper, we propose a novel cache management strategy that can limit the peak power consumption of L2 caches and provide fairness guarantees, such that the cache access latencies of the application threads co-scheduled on the CMP are impacted more uniformly. Our strategy is also extended to provide differentiated cache latency guarantees that can help the OS to enforce the desired thread priorities at the architectural level and achieve desired rates of thread progress for co-scheduled applications. Our solution features a two-tier control architecture rigorously designed based on advanced feedback control theory for guaranteed control accuracy and system stability. Extensive experimental results demonstrate that our solution can achieve the desired cache power capping, fair or differentiated cache sharing, and power-performance tradeoffs for many applications.**

Keywords: power capping, cache latency, fairness, performance differentiation, control theory, chip multiprocessors.

I. Introduction

In recent years, limiting the peak power consumption of chip multiprocessors (CMPs) has received a lot of attention [2], [3], [4], [5]. With the increased levels of core integration and transistor density, the gap between the peak and average power consumption of a CMP widens and leads to unnecessarily more expensive cooling, packaging, and power supplies in the CMP design. To effectively reduce the costs while allowing higher computing densities with more cores and caches integrated on a single die, power capping can be enforced at different levels of a CMP for maximized performance under given power constraints. In addition, to enable chip-level power capping, it is preferable that power can be flexibly shifted among the CPU cores and the shared on-chip L2 caches[1] within a CMP for performance optimization. Therefore, the peak power consumption of on-chip L2 caches in a CMP often needs to be constrained below a budget that is determined at runtime.

An effective way of controlling the power of L2 caches is to dynamically switch selected cache units (*e.g.*, ways, banks, or lines) between high- and low-power modes [6], [7]. For example, recent work [8] has proposed power-efficient management schemes for the non-uniform cache architecture (NUCA) designed to reduce wire delays. However, as a tradeoff, putting cache units into low-power modes may cause undesired long cache access latencies for the applications running on the CMP, because a smaller on-chip cache size may lead to more expensive off-chip data accesses. In this paper, we define cache access latency as the average number of cycles when accessing the L2 cache for data. Note that the latency is from the moment when an L2 cache request (including both read and write) is issued to the moment when the request is returned to the core. Therefore, the L2 cache access latency includes the latency of memory access if L2 misses occur. As cache access latencies contribute significantly to the number of CPU stall cycles and thus are directly related to application performance such as IPC (instructions per cycle), how to select cache units for power capping is challenging. Simplistic solutions may allocate too few cache units to applications running on some cores, resulting in undesired thread starving and thrashing. Therefore, it is important to provide desired latency guarantees for the co-scheduled application threads running on the cores in a CMP, even when the active L2 cache size is being dynamically changed to enforce a runtime power budget.

There can be different ways to provide cache access latency guarantees in a power-constrained CMP. First, a straightforward way is to provide absolute latency guarantees for some applications based on their performance needs (*e.g.*, IPC requirements). However, since the active L2 cache size can be significantly reduced at runtime to enforce a low power budget in scenarios such as thermal emergencies, the available active cache size can become too small to guarantee the absolute latencies for any applications. Therefore, it may not be feasible to provide absolute latency guarantees in power-constrained CMPs. The second and a more reasonable way is to provide fairness guarantees such that the cache access latencies of the co-scheduled applications are impacted more uniformly by cache resizing. As a result, execution time fairness, *i.e.*, how uniformly the execution times of co-scheduled threads are changed, can be better achieved. While existing work addresses fairness in terms of cache miss rate [9], we choose to control the average access latencies because the latency of each single access can vary significantly in NUCA caches due to both wire delays and L2 cache miss rate. As a result, miss rate alone may not accurately indicate the impact on application performance.

Fair latency guarantees can also be extended to provide differentiated cache access latencies, since the applications co-scheduled on different cores in a CMP may have different performance needs or priorities. Differentiated latency guarantees

This paper is a significantly extended version of a conference paper [1].

[1]Our solution is designed to manage the last-level shared on-chip cache in a CMP, which can be L2 or L3 in different CMPs.

can allow higher priority applications to have shorter latencies and so less performance degradation under the impacts of cache resizing. Differentiated cache sharing is important because it can help the OS to enforce the desired thread priorities. The OS normally enforces thread priorities by assigning more time slices to higher priority threads, with the assumption that more time slices can lead to higher rates of progress among all the co-scheduled threads. However, as cache implementation is traditionally thread-blind and priority-blind, a novel scheme is needed to help higher priority threads to achieve shorter cache latencies and so higher rates of progress at the architectural level. Differentiated latency guarantees can also be useful in accelerating critical threads to reduce load imbalance. Existing work on thread criticality relies mainly on per-core dynamic voltage and frequency scaling (DVFS) and task stealing [10], [11]. However, some applications are not suitable for task stealing while some memory-intensive applications cannot be accelerated by per-core DVFS alone. In addition, per-core DVFS is not supported by most existing CMPs and can be expensive for future CMPs [12]. In this paper, we demonstrate that differentiated latency guarantees can provide another way to achieve desired rates of progress for application threads. As cache latencies may have small impacts on the performance of some applications, differentiated latency control should be combined with per-core DVFS and task stealing to accelerate threads based on application characteristics.

In this paper, we propose a novel L2 cache management strategy that can enforce the desired power budget for on-chip L2 caches in a power-constrained CMP by cache resizing. In order to achieve fair or differentiated cache sharing under the impacts of power capping, our strategy dynamically partitions the available active cache size among the threads on the cores for fair or differentiated cache access latencies. A key challenge in implementing fair or differentiated cache sharing in a power-constrained CMP is that the cache partition to achieve the desired degree of fairness or differentiation depends on two uncertain factors: workloads that are unknown a *priori* and the total cache size that is dynamically varying. A main contribution of this paper is the introduction of a coordinated feedback control architecture for adapting cache partitioning and resizing such that the desired latency fairness or differentiation among threads on different cores can be achieved, while the cache power consumption can be controlled. Specifically, this paper makes the following major contributions:

- We propose a novel L2 cache management strategy that provides fair or differentiated cache sharing for threads running on a CMP whose power consumption must be constrained. While prior research in this field focuses primarily on reducing the power consumption of L2 caches, to our best knowledge, this paper presents the first study to limit the peak power of L2 caches.
- We design a coordinated two-tier feedback control architecture to simultaneously limit the peak cache power consumption and achieve the desired latency differentiation/fairness among different threads by dynamically conducting cache resizing and partitioning.
- We use advanced *feedback control theory* as a theoretical foundation to design and coordinate the two control loops for theoretically guaranteed control accuracy and system stability. This rigorous design methodology is in sharp contrast to heuristic-based adaptive solutions that heavily rely on extensive manual tuning.

The rest of this paper is organized as follows. Section II introduces the proposed two-tier control architecture. Section III presents the modeling, design, and analysis of the latency fairness controller. Section IV introduces the cache power controller. Section V discusses possible applications of relative cache latency control. Section VI provides the implementation details of our control solution. Section VII presents our experimental results. Section VIII reviews the related work. Finally, Section IX summarizes the paper.

II. Two-tier Cache Control Architecture

In this section, we provide a high-level description of our cache control system architecture, which features a two-tier controller design.

As shown in Figure 1, the primary control loop, *i.e.*, the L2 cache power control loop, limits the peak L2 cache power consumption under the desired budget (*e.g.*, 90% of the peak power consumption) by manipulating the total number of active L2 cache banks. The goal of the power control loop is to find the number of L2 cache banks that can be used under current power budget. Note that we use cache banks in static-NUCA (*i.e.*, S-NUCA) caches as our actuation unit, but our controller can work with finer actuation granularities, such as cache lines or blocks, with only minor changes and slightly more hardware overhead. The key components in the cache power control loop include a *power controller*, a *power monitor*, and a *cache resizing* modulator. The control loop is invoked periodically. In each control period of the cache power control loop, the controller receives the total power consumption of the L2 caches from the power monitor. Note that although the power consumption of L2 caches cannot be directly measured in today's CMPs, it can already be precisely estimated with on-chip programmable event counters for control purpose [13]. The cache power is the *controlled variable* of this loop. Based on the difference between the measured power value and the desired power budget, the controller then computes the number of cache banks in the chip that should stay active in the next control period, which is the *manipulated variable*. The controller then sends the number to the cache resizing modulator. The modulator changes the power modes of selected cache banks accordingly. There is only one cache power control loop for a CMP.

The secondary control loop, *i.e.*, the latency fairness/differentiation control loop, achieves the desired latency fairness or differentiation for the threads on the cores by conducting cache partitioning at runtime on a smaller timescale. As shown in Figure 1, in a CMP with N CPU cores, we have $N-1$ latency fairness/differentiation control loops. There are two ways to implement the latency control loops. The first way is that we select a reference thread on a core and then control the latencies of other threads relative to the latency of this reference thread. The second way is that we can have a control

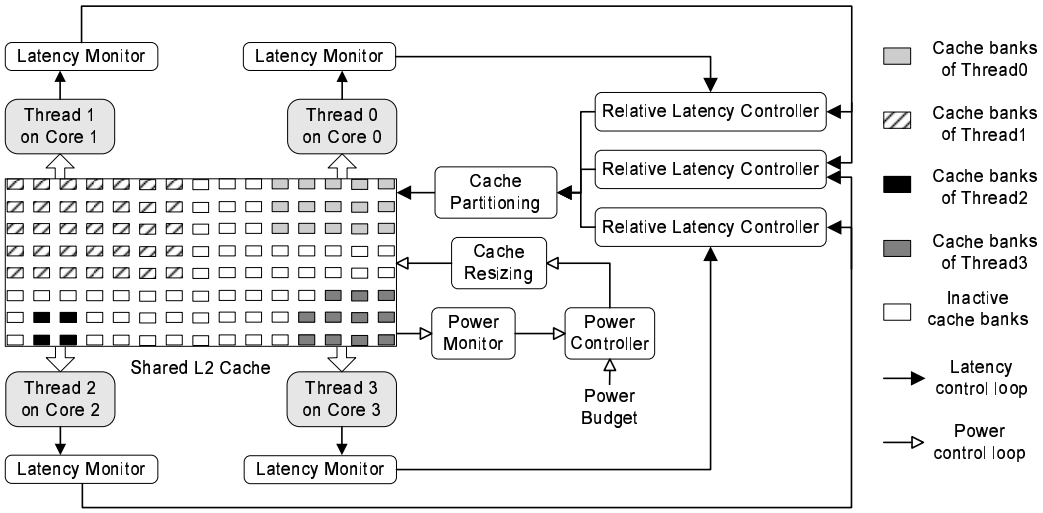


Fig. 1. Two-tier L2 cache control architecture for power capping and fair/differentiated sharing.

loop to control the latency ratio between the two active threads on every two adjacent cores. Since the controlled variable is the *relative* latencies among threads, the two implementations accomplish the same control functions. We plot the second way in Figure 1 for easier illustration. Specifically, to achieve fairness, we try to achieve approximately the same cache access latency for the active threads running on all the cores, despite the dynamically varying total cache size. Likewise, to achieve differentiation, we maintain the cache latency of a higher-priority thread shorter than that of a lower-priority thread. Note that previous fair cache sharing solutions [9], [14] try to manage the performance impacts of co-scheduling relative to the case when the applications are running alone on a CMP. As a result, they need to precisely know which applications are currently running on the CMP and then use corresponding off-line profiled performance measurements as references. A key advantage of our solution is that we do not assume such *a prior* knowledge and is thus more realistic. To handle multi-threaded applications, we can put cores running the same applications into groups and then conduct dynamic L2 cache partitioning among the groups.

The key components in a latency fairness/differentiation control loop for two cores include a *latency fairness/differentiation controller*, two *latency monitors* for the threads on the two cores, and a shared *cache partitioning* modulator. The control loop is also invoked periodically and its period is selected such that multiple cache access requests can be received during a control period and the actuation overhead is small compared with the period. The following steps are invoked at the end of every control period: 1) The latency monitors of the two cores measure the average absolute latencies of the two threads in the last control period and send the values to the controller through the feedback lane; 2) The controller calculates the relative latency of the two threads, which is the *controlled variable* of the control loop. The difference between the actual relative latency of the two threads and the desired value is the control error; 3) The controller then relies on control theory to compute the ratio between the numbers of cache banks to be allocated to the two threads, which is the *manipulated variable* of the control loop. The absolute number of cache banks can be calculated based on the total number of active banks determined by the power control loop; 4) The cache partitioning modulator then allocates the desired number of cache banks to each thread. We assume that a cache bank is not shared by multiple threads in this prototype design, but this assumption can be easily relaxed in a real system implementation by using finer actuation granularities with more hardware overhead or soft cache partitioning [9], [15].

Clearly, the two control loops need to be coordinated based on advanced control theory, because every invocation of the cache power control loop may change the total number of active cache banks of the CMP, and thus affect the stability of the latency control loops. Therefore, in order to achieve stability for the two-tier control architecture, we adopt the method recently proposed in [16] to configure the period of the power control loop to be longer than the settling time of the latency fairness/differentiation control loops. This guarantees that the latency loop can always enter its steady state within one control period of the power control loop, so that the two control loops are decoupled and can be designed independently. As long as the two control loops are stable individually, the coordinated control system is stable.

We now discuss the scalability of the proposed two-tier cache control solution. Since there is only one cache power control loop for a CMP, its overhead will not increase with the number of cores. Instead, with more cores in a CMP, the power control performance will improve due to the finer-grained actuation resulting from the increased number of cache banks. For a CMP with N CPU cores, we have $N-1$ latency fairness/differentiation control loops. Thus, the overhead of the latency control loops will increase linearly with the number of cores. However, as we discuss in Section III, the computational overhead of a latency control loop is negligible with just several multiplications and additions. Therefore, the proposed control solution is scalable for many-core CMPs with tens or hundreds of cores.

Since the core of each control loop is its controller, we introduce the design and analysis of the two controllers in the next two sections, respectively. The details of cache resizing and partitioning in NUCA caches and the implementation

details of other components in the control loops are discussed in Section VI.

III. Latency Fairness/Differentiation Controller

In this section, we present the modeling, design and analysis of the cache latency controller.

Given a CMP with N cores, as an example, we present the design process of the cache latency fairness/differentiation controller between Core i and Core j. The controller design between other pairs of cores is the same. We first introduce the following notation. $l_i(k)$ is the absolute access latency of the thread running on Core i in the k^{th} control period. $l(k)$ is the relative latency ratio between the two threads on Core i and Core j. Specifically $l(k) = l_i(k)/l_j(k)$. L is the reference latency ratio between the two threads on Core i and Core j. To achieve fairness, we can set $L = 1$. To achieve differentiation, the ratio L can be set (*e.g.*, by the OS) proportionally to the thread priorities or progress rates. For example, $L < 1$ means that the Core i thread should have a shorter latency (*i.e.*, a higher priority or a faster progress rate) than the Core j thread. It is important to note that L is not necessarily a constant and can be changed at runtime based on thread progress rates. For example, the thread criticality predictor in [10] can be used to predict how critical each of the co-scheduled threads is, such that proper reference ratios can be assigned to accelerate critical threads at different times. The focus of this paper is on providing a control scheme that can achieve the desired differentiation specified by the OS. We plan to investigate advanced algorithms for the OS to dynamically determine the reference ratio based on the desired thread priorities and/or progress rates in our future work. $e(k)$ is the difference between the reference ratio and the actual latency ratio. Specifically $e(k) = L - l(k)$. $c_i(k)$ is the number of active cache banks allocated to the thread on Core i in the k^{th} control period. $c(k)$ is the ratio between the numbers of cache banks partitioned to the two threads on Core j and Core i, specifically $c(k) = c_j(k)/c_i(k)$.

In the k^{th} control period, given the current access latency ratio $l(k)$, the control goal is to dynamically choose cache partition ratio $c(k)$ such that $l(k+1)$ can converge to the set point L after a finite number of control periods.

A. System Modeling

We now model the dynamics of the controlled system, namely the relationship between the controlled variable (*i.e.*, $l(k)$) and the manipulated variable (*i.e.*, $c(k)$). However, a well-established physical equation is unavailable to capture the relationship between the latency ratio of two threads and the ratio of cache banks partitioned to them, due to the complexity of cache systems in today's computers. Therefore, we use a standard approach called *system identification* [17] to this problem. Based on control theory, we use the following standard difference equation to model the controlled system:

$$l(k) = \sum_{j=1}^{n_1} a_j l(k-j) + \sum_{j=1}^{n_2} b_j c(k-j) \quad (1)$$

where n_1 and n_2 are the orders of the system output (*i.e.*, $l(k)$) and system input (*i.e.*, $c(k)$), respectively. a_j and b_j are system parameters whose values need to be determined by system identification. $l(k-j)$ and $c(k-j)$ are the values of the relative latency ratio (*i.e.*, $l(k)$) and the ratio of cache banks (*i.e.*, $c(k)$), respectively, in the $k-j$ control period.

The model can be generally explained from the systems perspective. The current latency ratio between two threads on two cores, $l(k)$, is mainly determined by three factors: 1) the ratio $l(k-j)$ in the previous n_1 control periods, 2) the cache partitioning actions $c(k-j)$ happened in the previous n_2 control periods, and 3) the cache latency properties of the workloads, which are captured by the system parameters a_j and b_j through system identification. The system parameters may change for two reasons: 1) workloads may have phase changes at runtime, resulting in different latency properties and 2) the system may run different workloads. The variations of the system parameters must be theoretically analyzed to guarantee system stability.

For system identification, we need to first determine the right orders for the system, *i.e.*, the values of n_1 and n_2 in the difference equation (1). The order values are normally a compromise between model simplicity and modeling accuracy. In this paper, we test different typical system orders. For each combination of n_1 and n_2, we use white noise to generate a series of system inputs (normalized to the allowed ranges) in a random fashion to stimulate the system and then measure the system output in each period. Based on the collected data, we use the *Least Squares Method (LSM)* to iteratively estimate the values of parameters a_i and b_i. We select four typical applications (*mcf*, *gap*, *ammp*, and *crafty*) in the SPEC CPU2000 benchmark suite as our workloads, based on the method introduced in [15]. *mcf* represents a workload with a data set larger than the available cache size. *gap* has a large number of compulsory misses. *ammp* continuously benefits as the cache size is being increased. *crafty* has a small working set but needs to frequently request data from L2 caches. We combine the four applications in 10 groups, as shown in Table I, to conduct the system identification experiments. The average values of the system parameters a_j and b_j resulting from the 10 groups are used as the parameters in the system model. Table II estimates the accuracy of the system model with different system orders (*i.e.*, n_1 and n_2 in (1)), in terms of Root Mean Squared Error (RMSE). A smaller RMSE indicates a higher modeling accuracy. Based on our results, we choose to have the orders of $n_1 = 1$ and $n_2 = 1$ because this combination has a reasonably small error (only 0.0285 or 19% higher than the third-order model), while keeping the model orders low. Therefore, the *nominal* system model resulted from our system identification is:

$$l(k) = a_1 l(k-1) + b_1 c(k-1) \quad (2)$$

where $a_1 = 0.595$ and $b_1 = 0.954$ are system parameters resulting from our experiments with the 10 groups of applications. To verify the accuracy of the model, we change the seed of the white noise signal to generate a different sequence of system inputs and then rerun the experiments to validate the results of system identification by comparing the actual system outputs and the estimated outputs based on the nominal model

TABLE I
APPLICATION COMBINATIONS AND MODEL VALIDATION RESULTS.

	Group1	Group2	Group3	Group4	Group5	Group6	Group7	Group8	Group9	Group10
Combi-nation	mcf gap	mcf ammp	mcf crafty	gap ammp	gap crafty	ammp crafty	mcf mcf	gap gap	ammp ammp	crafty crafty

TABLE II
THE RMSE VALUES OF DIFFERENT MODEL ORDERS.

	$n_1 = 0$	$n_1 = 1$	$n_1 = 2$	$n_1 = 3$
$n_2 = 1$	0.2500	0.1714	0.1704	0.1621
$n_2 = 2$	0.2211	0.1691	0.1684	0.1619
$n_2 = 3$	0.1793	0.1455	0.1455	0.1429

(2). Our results show that the differences are sufficiently small (with $R^2 = 0.87$).

Note that the real model of the system may be different from the nominal model (2) at runtime due to workload differences or parameter variations. To quantitatively analyze the impact of workload variations on the controller performance, we model the variations caused by different workloads and prove that the system controlled by the controller designed based on the nominal model can remain stable as long as the variation of a_1 is within $[-0.69a_1, 1.18a_1]$.

B. Controller Design

The goal of controller design is to achieve system stability, zero steady-state error, and short settling time. Following standard control theory [17], we design a *Proportional-Integral (PI)* controller to achieve the desired control performance. An important advantage of PI control is that it can provide robust control performance despite considerable modeling errors. In addition, PI control has a small computational overhead and is thus suitable to be implemented in the cache system. A more sophisticated PID (Proportional-Integral-Derivative) controller is not used because it is determined by the system model (2) that a derivative (D) term is not needed for the desired control performance. Having a derivative term unnecessarily might amplify the noise in the cache latency ratio. The designed PI controller in the Z-domain is:

$$F(z) = \frac{K_p(z - K_i)}{z - 1} \tag{3}$$

where $K_p = 0.49$ and $K_i = 0.37$ are control parameters that are analytically chosen using the standard Root Locus design method [17]. The corresponding closed-loop poles are $0.5638 \pm 0.3247i$. Since both the poles are inside the unit circle in the complex plane, the system is guaranteed to be stable when the nominal system model (2) is accurate (*i.e.*, a_1 and b_1 are exactly as estimated). In this case, we have proven that the closed-loop system can precisely converge to the desired set point with a zero steady state error. The detailed proofs are skipped due to space limitations. The time-domain form of the designed PI controller is:

$$c(k) = c(k-1) + K_p e(k) - K_p K_i e(k-1) \tag{4}$$

Therefore, the computational overhead of the designed controller is just several multiplications and additions.

IV. CACHE POWER CONTROLLER

In this section, we present the modeling, design, and analysis of the cache power controller.

We first introduce some notation. P is the desired power budget for the L2 caches in the CMP. $p(k)$ is the actual power consumption of the L2 caches in the k^{th} control period. $r(k)$ is the difference between the power budget and the actual power, specifically $r(k) = P - p(k)$. $s(k)$ is the total number of active cache banks in the k^{th} control period. $\Delta s(k) = s(k) - s(k-1)$ is the number difference of total active cache banks between the k^{th} and $k - 1^{th}$ control periods.

A. System Modeling

According to the investigation in previous work [18], cache leakage power has a linear relation with active cache size. The total power consumption of L2 caches can be the sum of the two parts: one part varies with the cache size while the other part remains approximately the same regardless of the cache size. Therefore, the L2 cache power consumption can be approximately modeled as:

$$p(k) = c * s(k) + d \tag{5}$$

where c is a model parameter determined by the workload and d represents the part that does not vary with cache size. Similar to the system identification experiments in Section III-A, we select four typical benchmarks (*mcf*, *gap*, *ammp*, and *crafty*) as our workloads to perform the experiments. In particular, we generate a series of system inputs (*i.e.*, $s(k)$) to stimulate the system and then measure the system output (*i.e.*, $p(k)$) in each control period. Based on the collected data, we use the *Least Squares Method (LSM)* to iteratively estimate the values of parameters c and d. Our results show that $c = 0.24$ on average for all the selected workloads. The value of d varies within a small range from 0.02 (with application *ammp*) to 0.58 (with application *mcf*). Note that a key advantage of the control-theoretic design approach is that it can tolerate a certain degree of modeling errors and can adapt to online model variations based on dynamic feedback [17]. As a result, our solution does not need to rely on power models that are 100% accurate, which is in sharp contrast to open-loop solutions that would fail without an accurate model. Hence, the dynamic system model as a difference equation is:

$$p(k) = p(k-1) + c\Delta s(k) \tag{6}$$

The real power model of the L2 caches may be different from the nominal model (6) at runtime due to several factors. For example, the leakage power of a cache bank can be sensitive to the system temperature. In the next subsection, we show that the closed-loop system controlled by a controller designed based on the nominal model (6) can remain stable as long as c varies within a certain range.

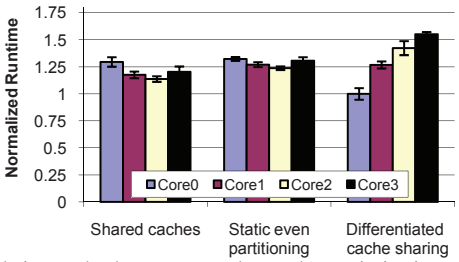


Fig. 2. Relative cache latency control to reduce priority inversion in EDF real-time scheduling. The thread on Core 0 has the highest priority and should finish first. Runtime is normalized to Core 0 in the controlled group.

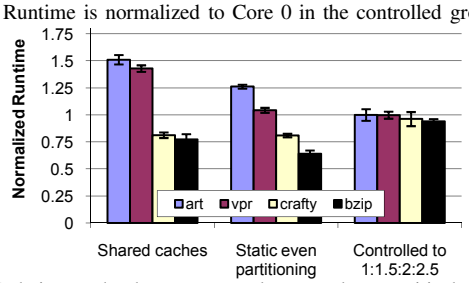


Fig. 3. Relative cache latency control to accelerate critical threads (*art* and *vpr*) for reduced load imbalance. Runtime is normalized to *art* in the controlled group.

B. Controller Design and Analysis

The goal of the controller design is to achieve system stability, zero steady-state error, and short settling time when the nominal system model (6) is accurate. Similar to the design of the cache latency controller, we design a Proportional-Integral (PI) controller to achieve the desired control performance because of the robustness and negligible overhead of PI control. The time-domain form of the designed PI controller is

$$s(k) = s(k-1) + K_1 r(k) - K_1 K_2 r(k-1) \tag{7}$$

where $K_1 = 3$ and $K_2 = 0.73$ are control parameters that are analytically determined using the standard Root Locus design method. The corresponding closed-loop poles are $0.7349 \pm 0.2033i$. Since both the poles are inside the unit circle, the system is stable. We have also proven that the closed-loop system can precisely converge to the desired set point with a zero steady state error. The worst-case settling time of the system is 12 control periods.

Although the controlled system is guaranteed to be stable when the system model (6) is accurate, stability has to be reevaluated when the model parameter c changes. Based on control theory, we have proven that the system can remain stable and achieve zero steady-state error when $c < -0.76$ or $c > 0$. Similarly, to handle systems with a modeling error that is outside this stability range, an online model estimator [19], [5] can be adopted to dynamically correct the system model based on real power measurements.

V. Discussion

In this section, we discuss possible applications of cache latency differentiation control in power-constrained CMPs.

First, cache latency differentiation control can help reduce priority inversion in real-time operating systems. Classic real-time theory is developed mainly for single-core systems with the assumption that the active thread has the full use of L2 caches. For example, in the well-known EDF (Earliest Deadline First) scheduling algorithm [20], a thread with the earliest absolute deadline has the highest priority and thus can get the CPU. However, in CMPs, the highest priority thread running on each core now needs to compete for the shared cache resources. As cache implementation is traditionally thread-blind and priority-blind, a thread with the overall earliest deadline may fail the cache competition and finish later than a thread running on another core with a later deadline, resulting in potential priority inversion and deadline misses. A naive solution is to allocate almost all the cache banks to the highest-priority thread. However, resource management in such a greedy way often leads to extremely long worst-case response times for the lower-priority threads and can make schedulability analysis very difficult, which are both highly undesirable for real-time systems (*e.g.*, the well-known release guard design) [20]. Therefore, in this case, relative cache latency control can be used by the OS to enforce thread priorities by setting the cache latency ratio among the threads proportional to their deadlines. As a result, a thread with an earlier deadline can have a shorter cache latency and thus a higher probability of meeting its deadline, while the lower-priority threads can also have chances to meet their deadlines. To validate this hypothesis with simulation, we run 4 copies of the benchmark `applu` on the 4 cores of a CMP for 200M instructions. We assume that the thread on Core 0 has the earliest deadline and thus the highest priority. The thread on Core 1 has the second highest priority while Core 3 has the lowest priority. With both the traditional shared L2 caches and even cache partitioning in a static way, the Core 0 thread is not guaranteed to have the earliest completion time. In contrast, as shown in Figure 2, if we set the cache latency ratio based on their priorities for differentiated cache sharing, the threads can finish in the desired sequence. Note again that the purpose of this paper is to provide a control scheme as a tool for OS to enforce desired differentiation. We plan to design algorithms that dynamically determine the desired cache access latency ratio between co-scheduled threads for improved real-time performance in our future work.

Second, cache latency differentiation control can also be applied to reduce load imbalance by accelerating critical threads. In many parallel programs, it is often preferable that threads (or processes) running on different cores can reach the same synchronization point (*e.g.*, a barrier or shared lock) at the same time, because otherwise faster threads would have to spend a large amount of time waiting for other threads, resulting in wasted CPU resources [11]. Therefore, in order to better exploit thread-level parallelism, it is important to dynamically identify the slowest (or critical) threads in those programs and then try to speed them up for improved performance. A recent study [10] demonstrates that substantial performance improvements can be achieved by accelerating critical threads. Existing work on thread criticality relies mainly on per-core dynamic voltage and frequency scaling (DVFS) and task stealing [10], [11]. However, some applications are not suitable for task stealing while some memory-intensive applications cannot be accelerated by per-core DVFS alone. In addition, per-core DVFS is not supported by most existing CMPs and can be expensive for future CMPs [12]. With relative cache latency control, we can effectively speed up critical threads

by shortening their cache access latencies based on their progress, thus providing another way to achieve desired rates of progress for application threads. As cache latencies may have small impacts on the performance of some applications, differentiated latency control should be combined with per-core DVFS and task stealing to accelerate threads based on application characteristics. To validate this hypothesis, we run 4 benchmarks: `art`, `vpr`, `crafty`, and `bzip` on the 4 cores of a CMP to simulate a 4-thread program. We assume that each thread needs to run 250M instructions and reach a synchronization point after every 50M instructions. Figure 3 plots the normalized average runtime between two synchronization points (with standard deviation) of each thread under different cache management policies. With the traditional shared L2 caches, we can observe load imbalance as `art` and `vpr` have longer runtime than `crafty` and `bzip`. The undesired load imbalance still exists if we evenly partition the caches in a static way. However, if we control the latency ratio to $1 : 1.5 : 2 : 2.5$, all the four threads can finish approximately at the same time, resulting in a 50% runtime speedup compared to the shared caches case. Note that the latency ratio can be dynamically adjusted by the OS based on the progress of the threads to achieve even better synchronization and a slightly higher speedup.

In addition to the two example applications, cache latency differentiation control can also be used in other applications such as desired thread speedup on selected cores. A lot of today's application software is developed with a single thread (called worker thread) doing most computation-intensive work. Therefore, it is often preferable to accelerate the worker thread in order to have a shorter execution time for the entire program. Furthermore, relative cache latency control can also be used to provide cache sharing fairness guarantees, as discussed before. As power is becoming a major constraint for CMPs in the near future, relative cache latency control can be used to ensure desired application fairness or differentiation under the impact of power capping.

VI. Implementation

In this section, we introduce our simulation environment and the implementation details of the control loops.

A. Simulation Environment

We extend the SimpleScalar simulator with *static*-NUCA cache configuration [21] as our simulation environment. In our simulations, we configure the CMP floor plan to have 4 Alpha 21264-like cores (65nm) with a frequency of 5GHz. We calculate the leakage power using HotLeakage [22] and calculate dynamic power using Wattch (built on CACTI) [23]. In all our experiments, we use the SPEC CPU2000 benchmark suite (V1.0) to test our two-tier cache control solution. The working temperature of the L2 cache is set to 80°C to simulate a typical real CMP setting based on the study in [24]. The L2 cache is configured to be 16MB (8-way set associative), because large caches are expected to be incorporated into future chips due to the shrinking of process technologies [21]. The main memory latency is 300 cycles. The L2 cache replacement algorithm is LRU (Least Recently Used). The L1 i-cache and d-cache are both 32KB (2-way, 64 byte block size) with a 3-cycle hit latency. The detailed simulation configuration parameters are listed in Table III.

B. Implementation of Control Loops

We now introduce the implementation details of each component in the two control loops.

Cache Access Latency Monitor. We add two counters for each core: one counter counts the number of stall cycles induced by cache accesses while the other counts the total number of cache accesses. We use the average access latencies in one control period to calculate the relative latency between the threads on two cores.

Cache Power Monitor. We use HotLeakage to calculate the leakage power consumption of the L2 caches and Wattch to calculate the dynamic power. We then use the sum of the two parts as our cache power reading. Note that the power consumption of L2 caches can be precisely estimated based on measurements feasible on physical chips [13].

Controllers. As introduced in Sections III and IV, both the latency controllers and cache power controller are PI controllers. Each controller is invoked once in one of its control period to receive the measured value of its controlled variable (*i.e.*, relative latency or cache power) from the monitor, and then conduct its control algorithm to compute the corresponding control output. As shown in the controller functions (4) and (7), each invocation of the controller only executes a couple of multiplications and additions. Therefore, the computational overhead of the controllers is small enough to be implemented in a real cache system. The control period of the latency control loop is set to 1M CPU cycles as a compromise between the system response speed and actuation overhead. A longer control period may fail to promptly react to some system dynamics while a shorter period may lead to increased actuation overhead. Based on our stability analysis, the control period of the cache power control loop is set to 10 times that of the latency control loop, *i.e.*, 10M cycles, in order to decouple the two controllers for global system stability.

In a real system, there can be two ways to implement our controllers. First, as discussed in Section III, the computational overhead of the two designed controllers is several multiplications and additions. Given the control periods of 10M and 1M cycles, the total time overhead of our controllers is less than 0.01%. The dominant circuit in our control solution is the fixed-point multiplication in the controllers, which is estimated to have an area overhead of $0.0057mm^2$ with the 65nm process technology [25] or a hardware overhead of 0.003% on a typical $200mm^2$ die. As a result, the controllers can be implemented on chip (as in Intel ItaniumII) to allow cache management on a fine timescale. Second, our controllers can also be implemented in service processor firmware (similar to the Thermal and Power Management Device (TPMD) used for IBM POWER7 [26]) and so their power and computational overheads will not directly affect the main CMP. This implementation is more flexible since the firmware is programmable but the control periods cannot be too small due to the communication delay between the service processor and the main CMP.

TABLE III
SIMULATOR CONFIGURATION PARAMETERS FOR SIMPLESCALAR.

Parameter Summary			
Technology	65nm	Temperature	80°C
Vdd	1.2V	Frequency	5GHz
Main memory latency	300 cycles	L2 cache	16MB, 8-way set associative
Fetch queue size	64	Issue queue size	60 (int and fp, each)
Register file size	100 (int and fp, each)	Width	8-wide
Branch predictor	comb. bimodal and 2-level	Bimodal predictor size	16K
Level 1 predictor	16K entries,history 12	Level 2 predictor	16K entries
Branch mispredict penalty	at least 12 cycles	Reorder Buffer size	80
BTB size	16K sets, 2-way	I and D TLB	128 entries, 8KB page size
L1 i-cache	32KB,2-way, 64 bytes block, 3 cycles hit latency		
L1 d-cache	32KB,2-way, 64 bytes block, 3 cycles hit latency		
L2 interconnection	16 x 16 banks, switch network, 2 cycles latency per hop		

The relative cache latency controllers need to interact with the OS to receive the desired set points via on-chip registers. If the OS scheduler changes the thread-core mapping, it needs to reset the desirable set points (*i.e.*, the latency ratio among each pair of cores). We assume that the OS scheduling period is much longer than our control period, similar to [14], [15]. As a result, the thread-core mapping will not change during the settling time of each cache latency controller. Once the OS changes the mapping or the set points, the cache latency controllers will re-converge to the new set points to enforce the desirable cache latency ratios.

Cache Resizing and Partitioning Modulator. In order to implement cache resizing and partitioning for S-NUCA caches in a CMP with N cores, we add $\lg(N+1)$ flag bits for each cache bank, which have only negligible gate overhead to be implemented in real CMP chips. If a cache bank is allocated to Core i, we set the flag to the value i. A non-zero value indicates that the bank should be active. If the cache bank is not allocated to any core, we switch the cache bank to a low-power mode (*i.e.*, inactive). We use the Gated-Vdd technology [27] to implement the power mode switch of a cache bank. We assume that a cache bank is not shared by multiple threads in this prototype design, but a finer actuation granularity (*e.g.*, cache lines or blocks) is also available. For example, Kaxiras et al. discuss a technology to transition selected cache lines into a low-power mode [28] and their granularity (cache lines) can be used in our control solution for even better control accuracy with a higher hardware overhead.

Since the S-NUCA caches have non-uniform cache access latency related to the wire length, we have a cache resizing policy to ensure that the active cache banks of each core are always the ones that are closest to the cache I/O port of the core. We adopt this policy for three reasons. First, in S-NUCA caches, a cache bank that is farther away from the core normally has a longer access latency due to the longer wire. Second, a longer routing distance from the core may also lead to more routing hops, an increased probability of more contention, and thus reduced on-chip network routing capability. Third, if the dynamic data migration policy is enforced in the S-NUCA caches, cache banks that are far away from cores may get much less frequently accessed [29]. Therefore, switching those cache banks to the low-power mode will lead to considerable power savings with only slight impacts on cache performance. After receiving the desired total active cache size from the power controller and the desired cache size ratios among threads on different cores from the latency controllers, the resizing and partitioning modulator calculates the number of active cache banks to be allocated to each thread. The cache resizing and partitioning modulator then uses the cache resizing policy to enforce the cache allocation. To implement the resizing policy, we maintain a table for each thread that records the cache banks allocated to the thread. In addition, in order to ensure that the active cache banks of a thread on a core are the ones that have the shortest distances to the core, the active cache area around the core needs to expand or shrink in both dimensions of the cache bank array. Therefore, we keep track of the cache row and column that are farthest from the core. Each core will expand or shrink in the farthest row and column alternatively to enforce the desired cache bank allocation. After the modulator allocates the active banks to the thread on each core, all the unallocated banks are switched to the low-power mode for power management.

Since we use Gated-Vdd technology, the cache contents are lost when we turn off cache banks. Therefore, all the dirty cache lines in those banks are written back to the memory during the mode transition. Since we only write back the dirty cache lines, the time and power overheads to write back a cache bank depend on how many cache lines in the to-be-turned-off bank are dirty. Timing and active power are simulated in the same way as for normal write-backs. Since SimpleScalar cannot simulate memory systems, our simulation considers the average effect of bus contention by having an increased average memory penalty to account for the average impact of bus contention [21]. Note that we assume static-NUCA instead of dynamic-NUCA (*i.e.*, D-NUCA) in this paper. The address mapping is modified according to the available cache size allocated to each core. First, to implement core-to-bank affinity of each bank in cache partitioning, we use an affinity register for each bank to claim the affinity. Second, to deal with cache banks that are turned off, we modify the cache mapping scheme to use (desired-cache-address mod active-cache-size) as the real address. The cache coherence is maintained by a snoopy bus protocol implemented in [21].

We plan to evaluate our control solution with D-NUCA L2 cache in our future work. However, it is well-known that address remapping and lookup in D-NUCA are challenging,

because it is difficult to come up with a way to maintain cache coherence, facilitate fast lookup to nearby banks, and avoid broadcast when trying to locate a particular line across many banks that might potentially store it. Therefore, implementing D-NUCA itself can be a serious challenge. In addition, to avoid the overheads of frequent remapping operations in D-NUCA, the period of cache resizing used by the power control loop in our solution may need to be reconsidered. Despite those challenges, D-NUCA may allow our solution to have better performance by turning off remote banks and turning on the banks close to cores, because D-NUCA policy is designed to put most recently used data to close banks and less used data to remote banks.

VII. Experimental Results

In this section, we present the results of our experiments conducted using the SimpleScalar simulator. We first evaluate the cache power controller. We then examine the latency differentiation controller. Finally, we test the two-tier cache control solution for fair and differentiated cache sharing in power-constrained CMPs.

We randomly select a SPEC CPU2000 benchmark, *crafty*, as the default application in our experiments unless otherwise noted. We test two different kinds of application cases: single benchmark on all the 4 cores and mixed benchmarks. For mixed benchmarks, we run the first benchmark on Cores 0 and 2 and the second benchmark on Cores 1 and 3. To our best knowledge, there are no existing solutions that can control cache latencies for fair or differentiated sharing. Therefore, we compare our solution with two baselines: even cache partitioning in a static way (denoted as Static) and a state-of-the-art cache partitioning solution (denoted as Priority) [30]. Priority is an ad hoc cache control solution that dynamically reallocates a fixed amount of cache size (i.e., 10%) between high and low priority applications on different cores to achieve differentiated cache miss rates. In contrast to our solution that explicitly controls cache latencies, Priority does not consider that the long access time of distant cache banks may lead to long core stall time in S-NUCA caches due to wire delays. For each benchmark, we first run 2 billion instructions and then start to measure the cache access latency and IPC values for the next 100 control periods of the power control loop (*i.e.*, 1 billion cycles). In our experiments, we assume that the threads on the cores run to the synchronization points without being preempted in the middle (*e.g.*, due to their highest scheduling priorities on their cores). However, please note that our solution is *not* restricted to this assumption. When thread preemption happens in a real system, based on the priorities of the new active threads, a new relative latency ratio set point can be used by the OS for desired thread performance differentiations.

A. Cache Power Control

In this section, we examine the cache power controller without enabling the latency controllers. All the cache banks are initially active in this set of experiments.

We first test the power controller in a common scenario where the power budget of the L2 caches needs to be reduced from 60 W to 40 W at the 500^{th} sampling point due to various reasons (e.g., thermal emergency). The power budget is then raised back to 60 W at the 1000^{th} sampling point after the emergency is resolved. This scenario is interesting because the power consumption of a CMP often needs to be capped and reduced at runtime [2], [5]. Reducing the power of L2 caches is an important way to maintain load balancing between CPU cores and L2 caches for optimized processor performance under the reduced budget. Figure 4(a) shows that the proposed PI power controller effectively controls the cache power to the desired budget by adjusting the active cache size. This experiment demonstrates that adjusting active cache size (and thus leakage power) is a promising way to conduct cache power management because leakage power contributes significantly to overall power consumption in large cache systems [31].

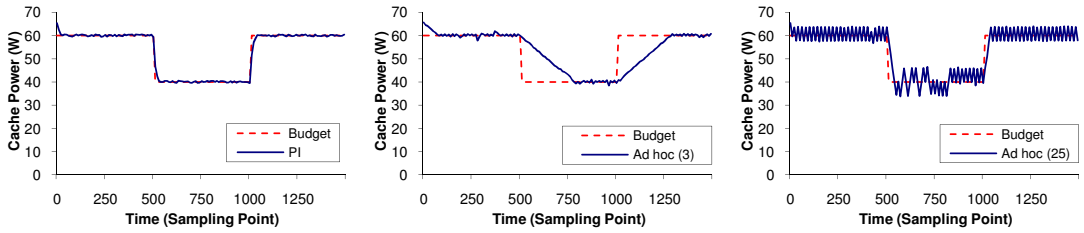
One may easily think that simplistic control solutions may also work well to control the cache power or achieve the desired fair or differentiated cache sharing. For example, a typical ad hoc power control solution (denoted as Ad hoc) would be to turn off M cache banks if the cache power violates its budget. However, without a theoretical foundation, it is commonly difficult to find a good value of M that would achieve the desired control performance in different cases. As shown in Figure 4(b), if M is too small (*e.g.*, 3 banks), Ad hoc may take an unnecessarily long settling time for power to converge back to the budget, resulting in overheating and even undesired processor shutdown. On the other hand, as shown in Figure 4(b), if the step size M is too large (*e.g.*, 25), Ad hoc may have large oscillations and overshoots. Therefore, it is undesirable to use Ad hoc in practice, which is consistent with the observations in [32], [5]. A fundamental benefit of the control-theoretic approach is that it provides standard ways to choose the right control parameters and gives us confidence for system stability and control accuracy. In addition, without well-designed coordination, naive control solutions may also cause different control loops to conflict with each other.

The cache power controller is designed based on a system model whose parameters are the results of system identification. Therefore, in order to test the robustness of the power controller when it is used in a system that is running a different workload, we conduct a set of experiments with different workload combinations under different power budgets: 20W, 35W, and 50W. Figure 5 shows that the average power of the L2 caches can be precisely controlled to the desired budget (with the maximum standard deviation smaller than 2% of the budget). The experimental results demonstrate that the power controller can precisely control the power consumption of the L2 caches for different workloads and power budgets with only small deviations.

B. Relative Cache Latency Control

In this experiment, we disable the cache power controller to focus on testing a relative latency controller (*i.e.*, cache latency differentiation controller) between the threads on two cores in two scenarios with runtime variations.

In the first scenario, we control the relative cache latency between two threads to the desired set point and then change



(a) PI cache power controller (b) Ad Hoc with 3 banks (c) Ad Hoc with 25 banks

Fig. 4. Typical runs of the PI cache power controller and a baseline under a power budget reduction.

the set point at runtime. We run application *crafty* on Cores 0 and 1 with an initial set point of 0.8, which means that the thread on Core 0 currently has a higher priority than that on Core 1 and thus should have a shorter cache access latency. As discussed in Section III, the reference ratio can be dynamically determined by the OS based on the desired thread priorities and/or progress, which is our future work. Figure 6 shows that the relative latency is controlled to the desired set point after a short settling time. Therefore, the desired latency differentiation has been achieved. At the 100^{th} sampling point, we assume that the priorities of the two threads have changed and the Core1 thread now has a higher priority. As a result, the set point dynamically changes to 1.2. As shown in Figure 6, the controller promptly achieves the new set point and enforces the desired new latency ratio. At the 300^{th} sampling point, the set point changes back to 0.8 and the controller achieves the set point again. This experiment demonstrates that the latency controller provides a flexible solution to enforce desired cache latency differentiation or fairness (if the ratio is set to one) at runtime based on the OS scheduling needs.

In the second scenario, we change the workload at runtime. We run application *crafty* on Cores 0 and 1 with a total active cache size of 128 banks for the two cores, and run application *apsi* on Cores 2 and 3 with a cache size of 64 banks for each core. Initially, we use the controller to control Cores 0 and 1 (with a set point of 1.2), which have the same workload. At the 200^{th} sampling point, we dynamically reconnect the controller to control the latency ratio between Cores 0 and 2, which are running different workloads. This change simulates a sudden workload change on Core 1 in a real CMP. As shown in Figure 7, this change introduces a significant control error because the latency of the thread on Core 2 is not controlled previously. After a short settling time, the controller enters the steady state again. The latency ratio between the threads on the two cores has been controlled to the desired set point. This experiment demonstrates that the latency controller can handle runtime workload variations.

As discussed in Section III, our system model is the result of system identification with four selected workloads. To test the robustness of the relative latency controller when it is used in a system that is running a different workload, we conduct a set of experiments with different workload combinations. Figure 8 shows the control accuracy under different benchmarks, where accuracy is defined as the average relative latency between Core 0 and Core 1, which is normalized to its set point. The standard deviations are also plotted. The experiments demonstrate that the relative latency controller can approximately achieve the desired latency ratio for workloads that are significantly different from the one used to design the controller.

C. Differentiated Cache Sharing

In this experiment, we evaluate the differentiated cache sharing provided by the cache latency controller with the cache power controller disabled. As discussed in Section I, differentiated cache sharing is important to help the OS to enforce the desired thread priorities at the architectural level and achieve desired rates of thread progress for co-scheduled applications. Note that fair sharing is a special case of differentiated sharing, so we show results for fair sharing only in Section VII-D due to space limitations.

Figure 9(a) shows the cache access latencies of the two threads on Cores 0 and 1 for different benchmarks under differentiated sharing and the two baselines. The benchmarks are split into two sub figures because their IPC values are at different scales. With differentiated sharing (*i.e.*, Controlled), we successfully achieve the desired latency differentiations by controlling the latency of the Core0 thread to be shorter than that of the Core1 thread. Specifically, because different benchmarks have different cache latency characteristics, we set the set point of cache latency ratio between Core0 and Core1 to be $1:1.4$ for *apsi*, *bzip2+art*, and *twolf+swim*, $1:1.5$ for *crafty+bzip2* and *gap+galgel*, and $1:1.3$ for all other benchmark combinations. In contrast, with the baseline Static, the Core1 thread has a shorter cache latency than the Core0 thread for 7 (out of 12) benchmark combinations (*e.g.*, *gcc*, *gzip2*, and *apsi*), which may result in lower-priority threads running faster at the OS level. Since the baseline Priority controls cache miss rates instead of cache latencies, even though Priority can help the higher-priority thread on Core0 to have a lower cache miss rate, the Core0 thread may still have a longer cache access latency in NUCA caches (*e.g.*, for *bzip2+art*).

Figure 9(b) shows the IPC values of the two threads on Cores 0 and 1 under the three cache management policies. As a result of latency differentiations, the Core0 thread is able to achieve a better IPC than the Core1 thread in all the single benchmark cases, for both memory-intensive benchmarks and randomly selected benchmarks. The Core0 thread also has a better IPC in most mixed benchmark cases, except for *mgrid+mesa*. Since *mgrid* and *mesa* have significantly different cache-access properties, the Core1 thread (running *mesa*) has a higher IPC than the Core0 thread (running *mgrid*) though it has a longer latency. However, even in this case, latency control helps to improve the IPC of the Core0 thread (running *mgrid*) as it has the highest IPC compared with the two baselines. Note that in some mixed benchmark cases

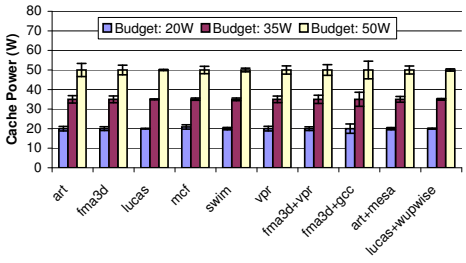


Fig. 5. Control accuracy under different workloads and budgets.

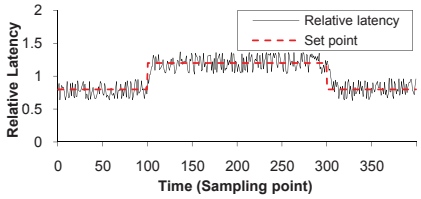


Fig. 6. A typical run of the relative latency controller when set point varies at runtime due to priority changes.

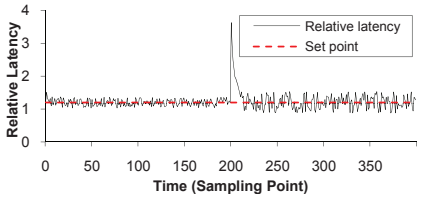


Fig. 7. A typical run of the relative latency controller when workload varies at runtime.

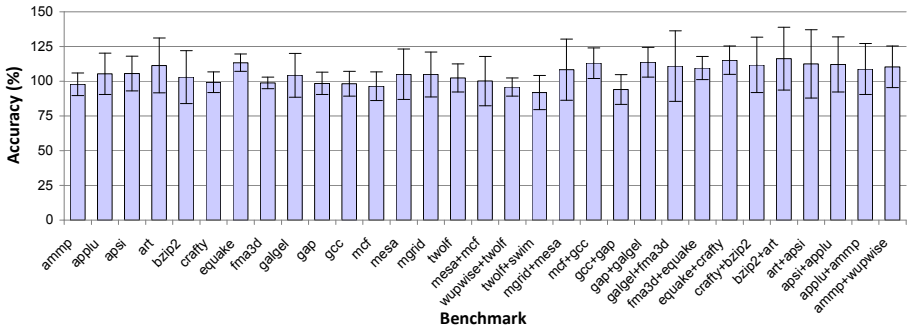
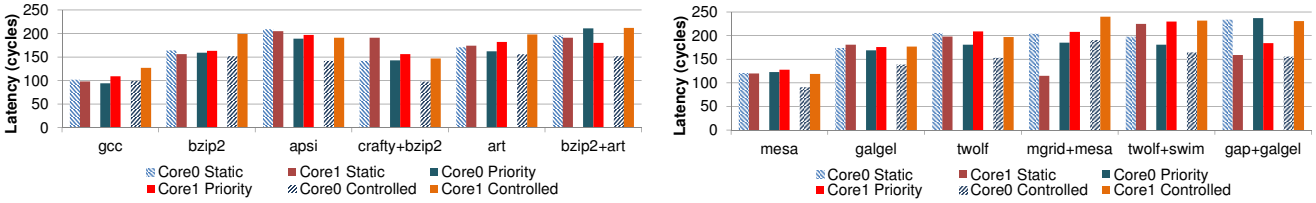
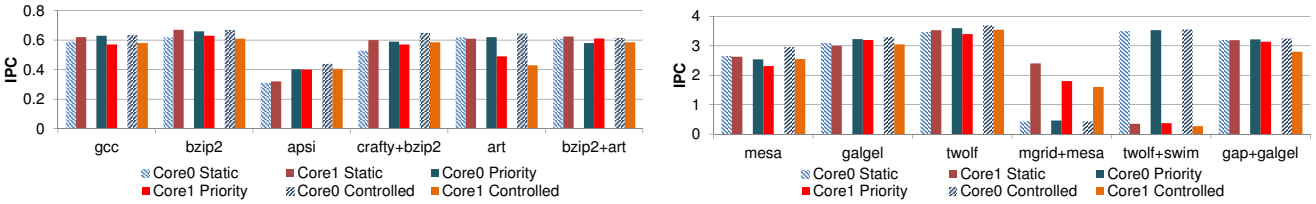


Fig. 8. Control accuracy of the relative latency controller under different workloads.



(a) Cache access latencies. Core 0 has a higher priority and so a shorter cache latency under Controlled.



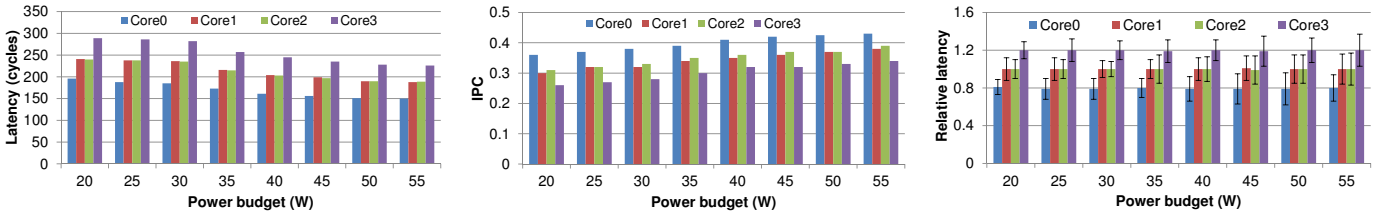
(b) IPC values. Latency differentiations can help to achieve desired IPC differentiations.

Fig. 9. Cache access latencies and IPC values of the two threads on Core 0 and Core 1 under three policies: shared L2 caches, static even cache partitioning, and differentiated sharing (i.e., Controlled).

similar to *mgrid+mesa*, we can still achieve a higher IPC for the higher-priority thread by allowing the OS to dynamically adjust the latency set point at runtime based on the thread progress. However, in some cases, due to the significantly different cache-access properties of different benchmarks, it may be infeasible for differentiated cache sharing to achieve the desired IPC differentiation, as the controller can enter saturation regions. Saturation can be caused by two reasons. First, the low-priority threads already have the minimum number of cache banks so that the controller can no longer reallocate cache banks to high-priority threads. In our implementation, the minimum number of cache banks for each thread is set to 4 in order to avoid cache starvation. Second, allocating more cache banks is no longer beneficial to the performance improvement of an application. In the case of controller saturation, differentiated cache sharing should be combined with per-core DVFS and task stealing to accelerate threads based on application characteristics.

In contrast to differentiated sharing, with the baseline Static, either of the two threads on Cores 0 and 1 can have a higher IPC value regardless of their priorities, even when they are running the same benchmark on homogeneous cores. As a result, the uncontrolled cache latencies can result in an *unpredictable* thread completion sequence. This unpredictability is highly undesirable for the OS-level priority-based thread scheduling. With the baseline Priority that provides a lower cache miss rate for the higher-priority thread on Core0, the number of undesired priority inversions has been reduced. However, since Priority does not control cache latencies directly, the Core0 thread may still have a lower IPC value than the Core1 thread in some cases (*e.g.*, for *bzip2+art*).

This experiment demonstrates that differentiated cache sharing can be used as one of the effective ways to achieve desired rates of thread progress for some co-scheduled appli-



(a) Cache access latencies (b) IPC values (c) Relative latencies

Fig. 10. Cache access latencies and IPC values of the four threads (running *crafty*) on the four cores of the CMP. When power budget varies from 20W to 55W, all the threads have reduced cache latencies and increased IPCs, while the desired latency differentiations and fairness are maintained.

cations.

D. Power-Performance Tradeoffs

In the previous sections, we have demonstrated that the cache power controller and the latency controller achieve their desired control functions individually. In this section, we enable all the controllers to demonstrate that our two-tier control solution can achieve desired power-performance tradeoffs.

In the first experiment, we run application *crafty* on all the 4 cores. To investigate the impacts of power constraint on fair and differentiated cache sharing, we increase the power budget from 20W to 55W with an increment of 5W. The cache sharing priority of Cores 0, 1, and 3 is set to *high, medium*, and *low*, respectively, to test the differentiated cache latency guarantees, while the priority of Core 2 is also set to *medium* to test fair cache sharing in terms of cache latency. Specifically, the set point of relative cache latency ratio among the four cores is set to $1 : 1.25 : 1.25 : 1.5$. Figure 10(a) shows that all the four threads on the four cores have reduced cache latencies when the power budget increases from 20W to 55W, due to increased number of total active cache banks. As a result, Figure 10(b) shows that all the threads have increased IPC values. In the meantime, as shown in Figure 10(c), the desired cache latency priorities have been strictly enforced (with a maximum error rate smaller than 1% from the set point). This allows the OS to achieve the desired rates of progress among the co-scheduled applications, despite that all the threads have improved performance with the increased power budget. In particular, by setting the same priority for Cores 1 and 2 for fairness, the latency differences between Core0 and Core1 are less than 5 cycles and the IPC differences are less than 0.01 under all the power budgets. This demonstrates that fair sharing, as a special case of differentiated sharing, can be guaranteed by our control solution. As discussed in Section I, differentiated sharing can help the OS to enforce the desired thread priorities at the architectural level.

The benchmark used in this experiment, *crafty*, is a chess game playing application and is one of the SPEC integer benchmarks. *crafty* represents applications with a significant number of logical operations that are relatively simple but need to frequently request data from the caches. As a cache-sensitive application whose execution is dominated by cache accesses [33], *crafty* achieves an almost reverse linear relationship between cache latency and IPC, *i.e.*, the decrease rate of the cache latency is approximately proportional to the increase rate of the IPC. While this experiment demonstrates the effectiveness of our control solution with cache-intensive benchmarks, it is important to investigate other benchmarks with different cache-access properties.

In the second set of experiments, we conduct the same experiment with different benchmarks for three power budgets: 20W, 35W, and 50W. Our workloads include memory-intensive benchmarks such as *apsi*, *art*, *bzip2*, and *swim*, as well as randomly selected benchmarks. We test two kinds of workload combinations: 1) four copies of a single benchmark running on the 4 cores and, 2) one benchmark running on Cores 0 and 2 while the other benchmark running on Cores 1 and 3.

To save space, we only present the results of the two threads on Cores 0 and 1. Cores 0 is configured to have a higher cache sharing priority than Core 1. The desired set point of cache latency ratio between Core 0 and Core 1 is set in the same way as in the experiments presented in Section VII-C. As shown in Figure 11(a), when the power budget increases from 20W to 50W, the cache latencies of the two threads on Cores 0 and 1 decrease for all the benchmark combinations and decrease in an approximately linear way for many benchmark combinations, such as *bzip2+art*, *twolf*, and *gap+galgel*. In addition, for all the 36 test cases (12 benchmark combinations and 3 power budgets), the Core0 thread successfully achieves a shorter cache latency than the Core1 thread. We calculate the error (i.e., deviation from the set point) of each case as (the resultant latency ratio - set point)/ set point. The average error of the 36 cases is only 0.2% with a small standard deviation of 1.2%. The maximum error is 3.1% for benchmarks *mesa* at the power budget of 20W. The results demonstrate that the relative latency controller can precisely achieve the desired latency differentiation. Figure 11(b) shows that the IPC values of the two cores increase as the power budget increases for all the benchmark combinations. For IPC values, the Core0 thread achieves a higher IPC value than the Core1 thread for 33 cases. The only exception is for *mgrid+mesa*. As analyzed in Section VII-C, in that case, a higher IPC may still be achieved for the Core0 thread by allowing the OS to dynamically adjust the reference set point at runtime based on the measured thread progress or combining per-core DVFS or task stealing. The experiments demonstrate that our two-tier control solution can be used to achieve the desired power-performance tradeoffs and maintain performance differentiations at the same time.

It is important to note that the controllers may saturate for some benchmark combinations. For example, although the two-tier control solution can precisely achieve the desired latency priorities for most benchmark combinations, the latency controller saturates for some benchmarks, such as *mesa* when the power budget is 20W. Note that it is easier for the latency

controller to saturate under small power budget, because when the power budget is small, the number of active cache banks is small, resulting a limited range for the latency controller to conduct dynamic cache partitioning. Consequently, it may become infeasible for the latency controller to achieve the desired priorities. Figure 11(a) shows that the Core0 thread only has a slightly shorter cache latency than the Core1 thread for *mesa* due to saturation when the power budget is 20W. As a result, the IPC values of the two threads are close in this case. However, when the power budget increases to 35W and 50W, the latency controller has more active cache banks for partitioning and thus can achieve the desired latency priorities by allocating more cache to the high-priority thread (*i.e.*, the Core0 thread). Consequently, the Core0 thread can have a much shorter cache latency and thus a much higher IPC value than the Core1 thread.

We acknowledge that cache latencies may have small impacts on the performance of some applications (*e.g.*, CPU-intensive ones). The goal of our paper is to demonstrate that differentiated latency control can provide another way to achieve desired rates of progress for some co-scheduled application threads (*e.g.*, cache-intensive ones). In a real system, differentiated latency control should be combined with per-core DVFS and task stealing to accelerate threads based on application characteristics, which is our future work.

VIII. Related Work

Power has become an important design constraint for CMPs. Some recent work has studied power capping for CMPs. For example, Intel's Foxton technology [34] has successfully controlled the power and temperature of a processor by using chip-wide DVFS. Isci et al. [2] proposed both a closed-loop algorithm and a prediction based algorithm to control the power of a CMP to stay below a power budget. Meng et al. [4] presented heuristic-based algorithms to use both per-core DVFS and cache resizing as actuators to control power. Teodorescu et al. [3] developed an optimization algorithm based on linear programming to provide power management for a CMP based on both DVFS and thread mapping. Wang et al. [5] developed a control-theoretic power controller for improved system performance. While the related work mainly focuses on adapting the dynamic power of the CPU cores in a CMP, we use cache resizing to control the cache power and provide performance differentiations. Our scheme can be combined with those CPU core power control solutions for chip-level power management.

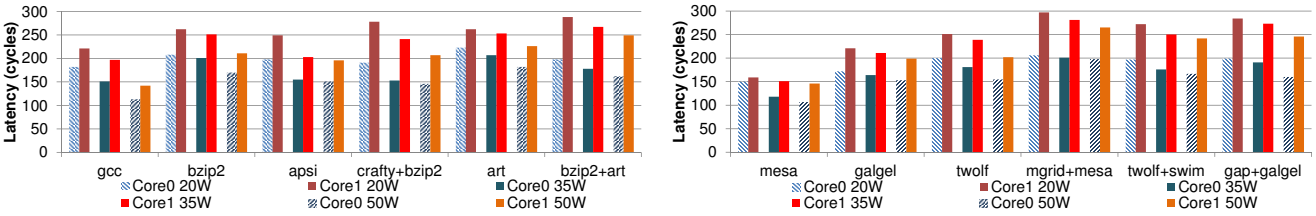
Adaptive cache partitioning for CMPs has recently received a lot of attention. This paper is different because we use cache partitioning to achieve performance differentiation in NUCA caches for threads with different performance needs and priorities in a CMP with power constraints. Some studies have been conducted to dynamically adapt the cache size for power savings. For example, Albonesi et al. proposed to turn off cache ways for reduced dynamic power [35]. Bardine et al. explored the possibility of applying this technique to NUCA caches [8]. Kobayashi et al. presented a heuristic-based control algorithm based on locality metrics [36]. Our paper is different because 1) we provide differentiated performance guarantees in addition to power management, and 2) we rely on control theory as a theoretical foundation for theoretically guaranteed control accuracy and system stability.

Feedback control theory has been successfully applied to control temperature, power, and performance in computer architecture research. For example, Skadron et al. [37] used control theory to dynamically manage the temperature of microprocessors. Likewise, Wu et al. [38] managed power using dynamic voltage scaling by controlling the synchronizing queues in multi-clock-domain processors. In contrast to their work that relies on basic control theory to design a single control loop, we coordinate two control loops in a hierarchical way for guaranteed stability.

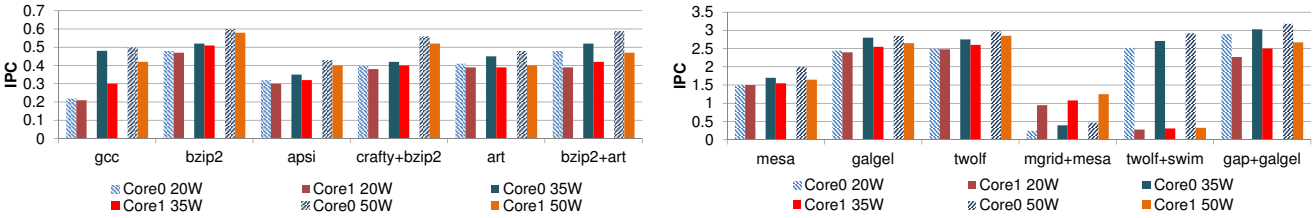
Our work is also related to thread criticality research. Bhattacharjee et al. [10] proposed a novel way to predict thread criticality. Cai et al. [11] used DVFS to slow down non-critical threads for power savings. In our work, we provide cache latency differentiations to accelerate critical threads. Our scheme can also be used to guarantee cache sharing fairness. While Kim et al. proposed partitioning policies with five fairness metrics defined based on cache miss rates [9], we address average latencies because the latency of each single access can vary significantly in the NUCA caches. As a result, miss rate may not accurately indicate the impact on application performance. Zhou et al. [14] also proposed fair cache sharing to have the same impacts on execution times. However, similar to [9], they need to precisely know which applications are currently running on the CMP such that they can use corresponding off-line profiled execution times as references.

IX. Conclusions and Future Work

In order to enable chip-level power capping, the peak power consumption of on-chip L2 caches in a CMP often needs to be constrained by dynamically transitioning selected cache banks into low-power modes. While prior research in this field focuses primarily on reducing the power consumption of L2 caches, this paper aims at limiting the peak power consumption of L2 caches. To avoid undesired thread starving and thrashing caused by dynamic cache resizing for power capping, our strategy can provide fairness guarantees such that the cache access latencies of the application threads co-scheduled on the CMP are impacted more uniformly. Furthermore, our strategy is also extended to provide differentiated cache latency guarantees that can help the OS to enforce the desired thread priorities at the architectural level and achieve differentiated rates of thread progress for co-scheduled applications. Our solution features a two-tier control architecture rigorously designed based on advanced feedback control theory for guaranteed control accuracy and system stability. Extensive experimental results demonstrate that our solution can achieve the desired cache power capping, fair or differentiated cache sharing, and power-performance tradeoffs for many applications. Our results also demonstrate that differentiated latency guarantees can provide another effective way to achieve desired rates of progress for application threads. As cache latencies may have small impacts on the performance of some applications, differentiated latency control should be



(a) Cache access latencies. Reduced latencies and desired latency differentiations are achieved with increased power budget.



(b) IPC values. Increased IPC values and desired progress differentiations are achieved with increased power budget.

Fig. 11. Cache access latencies and IPC values of the higher-priority thread on Core 0 and the lower-priority thread on Core 1 under different benchmarks, when the power budget increases from 20W to 50W.

combined with per-core DVFS and task stealing to accelerate threads based on application characteristics, which is our future work.

ACKNOWLEDGEMENTS

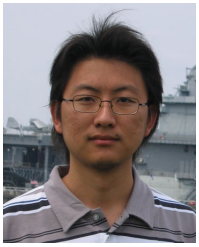
We thank Dr. Naveen Muralimanohar at HP Labs for providing the source code of SimpleScalar S-NUCA cache implementation. This work is funded in part by NSF under CNS-0720663, CNS-0845390, CNS-0915959, and CCF-1017336, and by ONR under N00014-09-1-0750.

REFERENCES

[1] X. Wang, K. Ma, and Y. Wang, "Achieving fair or differentiated cache sharing in power-constrained chip multiprocessors," in *ICPP*, 2010.
[2] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *MICRO*, 2006.
[3] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *ISCA*, 2008.
[4] K. Meng, R. Joseph, R. P. Dick, and L. Shang, "Multi-optimization power management for chip multiprocessors," in *PACT*, 2008.
[5] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *ISCA*, 2009.
[6] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *ISCA*, 2002.
[7] Y. Meng *et al.*, "Exploring the limits of leakage power reduction in caches," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, 2005.
[8] A. Bardine, P. Foglia, G. Gabrielli, C. A. Prete, and P. Stenström, "Improving power efficiency of D-NUCA caches," *SIGARCH Comput. Archit. News*, vol. 35, no. 4, 2007.
[9] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *PACT*, 2004.
[10] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, 2009.
[11] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González, "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *PACT*, 2008.
[12] K. R. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: Fine-grained power management for multi-core systems," in *ISCA*, 2009.
[13] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *MICRO*, 2003.
[14] X. Zhou, W. Chen, and W. Zheng, "Cache sharing management for performance fairness in chip multiprocessors," in *PACT*, 2009.
[15] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006.
[16] Y. Wang, X. Wang, M. Chen, and X. Zhu, "Power-efficient response time guarantees for virtualized enterprise servers," in *RTSS*, 2008.
[17] G. F. Franklin, D. J. Powell, and M. Workman, *Digital Control of Dynamic Systems, 3rd edition*. Addition-Wesley, 1997.
[18] K. Skadron *et al.*, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, 2004.
[19] X. Liu *et al.*, "Optimal multivariate control for differentiated services on a shared hosting platform," in *DC*, 2007.
[20] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.
[21] N. Muralimanohar and R. Balasubramonian, "Interconnect design considerations for large NUCA caches," in *ISCA*, 2007.
[22] Y. Zhang *et al.*, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Tech Report, Univ. of Virginia," 2003.
[23] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *ISCA*, 2000.
[24] A. Greenhill, *Power Saving in the UltraSPARC T1 Processor*. Sun Microsystem Whitepaper, 2005.
[25] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *MICRO*, 2008.
[26] M. Ware *et al.*, "Architecting for power management: The IBM POWER7 approach," in *HPCA*, 2010.
[27] M. Powell *et al.*, "Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories," in *ISLPED*, 2000.
[28] S. Kaxiras, Z. Hu, G. J. Narlikar, and R. McLellan, "Cache-line decay: A mechanism to reduce cache leakage power," in *PACS*, 2001.
[29] A. Bardine, M. Comparetti, P. Foglia, G. Gabrielli, C. A. Prete, and P. Stenström, "Leveraging data promotion for low power D-NUCA caches," in *DSD*, 2008.
[30] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," in *SIGMETRICS*, 2007.
[31] Y. Meng, T. Sherwood, and R. Kastner, "On the limits of leakage power reduction in caches," in *HPCA*, 2005.
[32] X. Wang and M. Chen, "Cluster-level feedback power control for performance optimization," in *HPCA*, 2008.
[33] N. Muralimanohar *et al.*, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.
[34] M. Rich *et al.*, "Power and temperature control on a 90-nm itanium family processor," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 1, 2006.
[35] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *MICRO*, 1999.
[36] H. Kobayashi, I. Kotera, and H. Takizawa, "Locality analysis to control dynamically way-adaptable caches," *SIGARCH Comput. Archit. News*, vol. 33, no. 3, 2005.
[37] K. Skadron, T. Abdelzaher, and M. R. Stan, "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management," in *HPCA*, 2002.
[38] Q. Wu, P. Juang, M. Martonosi, L.-S. Peh, and D. W. Clark, "Formal control techniques for power-performance management." *IEEE Micro*, vol. 25, no. 5, 2005.

Xiaorui Wang received the BS degree from Southeast University, China, in 1995, the MS degree from the University of Louisville in 2002, and the PhD degree from Washington University, St. Louis, in 2006, all in computer science. He is an associate professor in the Department of Electrical and Computer Engineering at The Ohio State University. He is the recipient of the US Office of Naval Research (ONR) Young Investigator (YIP) Award in 2011, the US NSF CAREER Award in 2009, the Power-Aware Computing Award from Microsoft Research in 2008, and the IBM Real-Time Innovation Award in 2007. He also received the Best Paper Award from the 29th IEEE Real-Time Systems Symposium (RTSS) in 2008. He is an author or coauthor of more than 70 refereed publications. From 2006 to 2011, he was an assistant professor at the University of Tennessee, Knoxville, where he received the EECS Early Career Development Award, the Chancellor's Award for Professional Promise, and the College of Engineering Research Fellow Award in 2008, 2009, and 2010, respectively. In 2005, he worked at the IBM Austin Research Laboratory, designing power control algorithms for high-performance computer servers. From 1998 to 2001, he was a senior software engineer and then a project manager at Huawei Technologies Co. Ltd., China, developing distributed management systems for optical networks. His research interests include power-aware computer systems and architecture, real-time embedded systems, and cyber-physical systems. He is a member of the IEEE and the IEEE Computer Society.

Kai Ma Kai Ma received the BS and MS degrees in Electrical Engineering from Zhejiang University, Hangzhou, China in 2004, and Tongji University, Shanghai, China in 2007, respectively. He is a PhD candidate in computer engineering in the Department of Electrical and Computer Engineering at The Ohio State University. His current research focuses on power management of multi-core computing systems.



Yefu Wang received the BS and MS degrees from Harbin Institute of Technology, Harbin, China, in 2003 and 2006. He is currently a PhD candidate in computer engineering in the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. His current research focuses on system-level power management of computing systems, with applications to virtualized enterprise server environments.