Adaptive Timekeeping Replacement: Fine-Grained Capacity Management for Shared CMP Caches

CAROLE-JEAN WU, Princeton University
MARGARET MARTONOSI, Princeton University

In chip multiprocessors (CMPs), several high-performance cores typically compete for capacity in a shared last-level cache. This causes degraded and unpredictable memory performance for multiprogrammed and parallel workloads. In response, recent schemes apportion cache bandwidth and capacity in ways that offer better aggregate performance for the workloads. These schemes, however, focus primarily on relatively coarse-grained capacity management without concern for operating system process priority levels.

In this work, we explore capacity management approaches that are both temporally and spatially more fine-grained than prior work. We also consider operating system priority levels as part of capacity management. We propose a capacity management mechanism based on timekeeping techniques that track the time interval since the last access to cached data. This Adaptive Timekeeping Replacement (ATR) scheme maintains aggregate cache occupancies that reflect the priority and footprint of each application. The key novelties of our work are (1) ATR offers a complete cache capacity management framework taking into account application priorities and memory characteristics, and (2) ATR's fine-grained cache capacity control is demonstrated to be effective and important in improving the performance of parallel workloads in addition to sequential ones.

We evaluate our ideas using a full-system simulator and multiprogrammed workloads of both sequential and parallel applications. This is the first detailed study of shared cache capacity management considering thread behaviors in parallel applications. ATR outperforms an unmanaged system by as much as 1.63X and by an average of 1.19X. ATR's fine-grained temporal control is particularly important for parallel applications, which are expected to be increasingly prevalent in years to come.

Categories and Subject Descriptors: B.8.3 [**Hardware**]: Memory Structures—*General*

General Terms: Design, Performance

Additional Key Words and Phrases: Cache decay, capacity management, shared resource management

ACM Reference Format:
Wu, C., Martonosi, M. 2010. Adaptive Timekeeping Replacement: Fine-Grained Capacity Management for Shared CMP Caches. ACM Trans. Architec. Code Optim. 11, 11, Article 11 (February 2011), 27 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

It is common today to run multiple heterogeneous applications, such as web-server, video-streaming, graphic-intensive, scientific, and data mining workloads, on chip-multiprocessor (CMP) systems, where multiple cores share the last level on-chip cache. Conventionally, on-chip caches implement pseudo Least-Recently-Used (LRU) replacement policies. However, commonly-used LRU replacement policies do not distinguish

Author's address: Carole-Jean Wu, Department of Electrical Engineering, 34 Olden Street, Princeton University, Princeton, New Jersey, 08544; Margaret Martonosi, Department of Computer Science, 35 Olden Street, Princeton University, Princeton, New Jersey, 08544.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

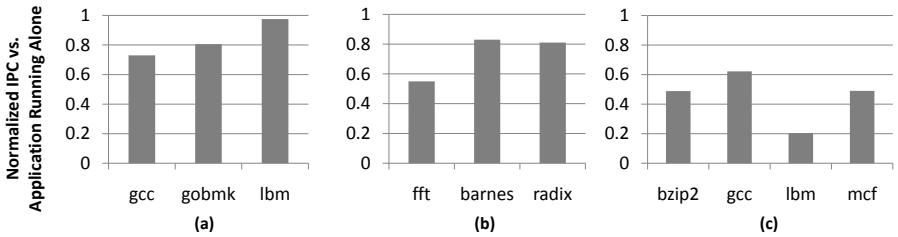

Fig. 1. **The bars show normalized instructions per cycle (IPC) when applications run as an unmanaged group for each individual workload. Performance degradation is experienced when multiple applications are running simultaneously. More importantly, performance degradation of a high-priority application, e.g.** `gcc` **in (a),** `fft` **in (b), or** `lbm` **in (c), is more significant than others in the workloads. However, the unmanaged scheme (LRU) does not take into account process priorities while allocating cache capacity.**

between processes and their different memory needs. In addition, current systems cannot explicitly assign shared cache capacity effectively based on process priority and characteristics of each process memory footprint. Consequently, while multiple processes have access to the shared cache, it is possible, for example, that a large-footprint but low-priority process uses the shared cache intensively, such that other high-priority processes are left with insufficient cache share throughout execution. This can result in performance degradation and unpredictability. In order to provide performance predictability, particularly for high-priority processes, we have to prioritize accesses to the shared cache and mitigate inter-process interference in the shared cache resource.

Not surprisingly, when multiple applications are running simultaneously, the performance of all applications is consistently degraded. Figure 1 shows this for three heterogeneous workloads. The bars show normalized instructions per cycle (IPC) when applications run as an unmanaged group for each individual workload over when each individual application runs alone. Performance suffers because all concurrent applications contend for the shared resources simultaneously in each workload. More importantly, performance degradation of a high-priority application is more significant than others in the workloads. This is because the unmanaged scheme (LRU) does not take into account application priorities while allocating cache capacity.

Furthermore, there are no detailed studies on shared resource management for parallel applications. Most of the existing proposals only target sequential applications. As mainstream workloads consist of more and more parallel applications today, it is important to explore the needs of shared cache capacity management for parallel applications. In particular, management approaches must acknowledge the differences between references coming from cooperating versus competing threads. This work is the first to do so.

In order to provide shared cache capacity management on CMPs, various shared resource management techniques have been proposed previously for shared caches [Bitirgen et al. 2008; Chang and Sohi 2007; Hsu et al. 2006; Iyer 2004; Iyer et al. 2007; Kim et al. 2004; Nesbit et al. 2007; Petoumenos et al. 2006; Qureshi and Patt 2006; Rafique et al. 2006; Zhao et al. 2007]. However, most of these techniques focus on studies of the high level policies in their frameworks and often ignore the effectiveness offered by different possible underlying capacity management mechanisms. If the chosen cache capacity management technique partitions the shared cache space in coarse granularity, the cache may not be utilized as efficiently. For example, way

partitioning is a commonly-used coarse-grained shared cache partitioning technique [Bitirgen et al. 2008; Nesbit et al. 2007; Qureshi and Patt 2006]. The major advantage of using way partitioning is its moderate overhead and simple implementation, but it has drawbacks as well. First, in order to have sufficient control in implementing performance isolation, it is preferable to have more ways in the set-associative cache than the number of concurrent processes. As the number of concurrent processes increases in today's CMP systems, this is difficult to achieve. In addition, way partitioning results in inefficient cache space utilization due to its coarse granularity in space allocation.

Other papers, which offer finer-grained cache capacity management, modify cache insertion or replacement policies to utilize shared caches more efficiently [Jaleel et al. 2008; Suh et al. 2001; Suh et al. 2002]. These proposals often relate the amount of cache space allocated to a process to its performance improvement directly but take into account temporal behavior of cached data implicitly (for example, with modified cache insertion/replacement policies). Our work is distinct in offering finer temporal control. Furthermore, these proposals aim at improving aggregate system performance, but do not distinguish between process priorities. Finally, there has been no work studying the shared cache capacity problem for parallel applications.

In this work, we propose a novel shared cache capacity management scheme called *Adaptive Timekeeping Replacement (ATR)*. ATR is a fine-grained mechanism that assigns a "lifetime" to cache lines according to operating system process priority and application memory reference patterns. ATR observes the memory needs of an application during different program phases and adjusts cache space allocation accordingly by varying the cache lifetime of each line. It does not only minimize unnecessary cache memory interference from running processes, but also allows more fluid apportioning of the shared cache space. Consequently, the shared cache space is utilized more effectively. As such, data remaining in the cache exhibit two critical characteristics: *high priority* and *temporal locality*. Finally, ATR is the first shared cache capacity management considering thread behaviors in parallel applications. This fine-grained cache capacity control improves the performance of parallel workloads by speeding up critical threads in parallel applications. We demonstrate ATR's fine-grained time-keeping feature is effective and important, particularly for parallel workloads.

The contributions of our work over the previously proposed capacity management techniques are:

(1) Our proposed Adaptive Timekeeping Replacement (ATR) scheme is effective in managing shared caches. We run full-system simulation to evaluate the ATR scheme and its performance effects, taking into account operating system influence on the problem. We demonstrate that ATR, which takes into account application priorities and manages capacity in a fine-grained manner, outperforms a baseline unmanaged system by as much as 1.63X and by an average of 1.19X for multiprogrammed workloads with less than 3% hardware overhead.
(2) ATR's fine-grained cache capacity management feature is particularly important for parallel threads. We are the first to investigate capacity management on a mixture of parallel applications. We show that ATR causes critical threads in a parallel application to be sped up by 7% resulting in better overall application performance.
(3) Furthermore, the ATR scheme can be used to implement a variety of high-level shared resource management policies implemented by the operating system, because of its flexibility and effectiveness. It can also be viewed as a building block and used along with other prior work on cache and network bandwidth management.

The structure of this paper is as follows. In Section 2, we give an overview of shared resource management. In Section 3, we introduce timekeeping techniques for ordering

cache replacement. Then, Section 4 offers a detailed description of ATR's algorithm and enforcement. In Sections 5 and 6, we describe our simulation framework, evaluate the shared resource management schemes, and analyze their performance in detail. Section 7 discusses related work on shared resource management. This is followed by Section 8, which presents design issues and future work. Finally, Section 9 offers our conclusions.

2. SHARED RESOURCE MANAGEMENT OVERVIEW

Shared resource management refers to partitioning common resources among multiple requesters. In the case of multiple applications running on CMP systems, multiple requests are made to the shared cache simultaneously. However, current systems cannot explicitly assign shared cache resources effectively based on process priority and characteristics of each process' memory footprint. Consequently, in order to arbitrate memory accesses from all running processes while taking into account heterogeneity and process priority, shared resource management is critical.

An important goal in shared resource management is performance predictability. While multiple processes have access to the shared cache in CMP systems, it is possible that a process, e.g. a memory-bound one, uses the shared cache intensively and other processes are left with insufficient cache share throughout execution. This can result in performance degradation and performance unpredictability. In order to provide performance predictability, particularly for high-priority processes, we have to prioritize accesses to the shared cache and isolate inter-process interference in the shared caches.

Typically, shared resource management can be divided into two parts: a shared resource management policy and its capacity management mechanism. Shared resource management policies define a performance goal, such as overall system throughput, fair sharing, and/or quality of service, for a targeted system by specifying the amount of resources allocated to all running processes in the system. Then an underlying capacity management mechanism implements the specified policies to achieve the desired performance goal. Our proposed *Adaptive Timekeeping Replacement (ATR)* scheme fits into the category of the capacity management mechanism. ATR is a hardware mechanism that can control shared caches effectively and enforce the high-level shared resource management policy.

There are some major design challenges faced by existing shared resource management techniques. One of these challenges is to preferentially partition shared caches based on process priorities assigned by operating systems. Another challenge is to design new management techniques for collaborative threads in parallel workloads, which are becoming more prevalent today. Next we discuss the two problems in more detail.

2.1. Process Priorities Assigned by Operating Systems

Traditionally operating systems play the role of assigning priorities to active processes and allocating system resources based on process priorities. This is typically done by allocating larger CPU time slices to high-priority processes and shorter time slices to low-priority processes. However, these operating system assigned priorities are not taken into account in today's shared cache capacity management schemes. Currently, existing schemes mainly focus on exploiting application memory characteristics while managing the shared CMP caches, but ignore application priorities. To address this issue, ATR accounts for application priorities in its capacity management while also responding to application memory characteristics.

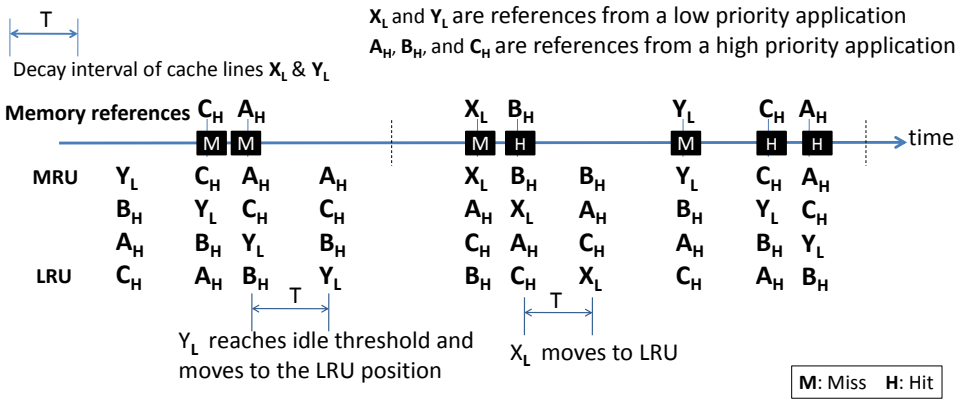

Fig. 2. **An illustrative case study, where the timekeeping replacement policy outperforms the LRU replacement policy for a 4-way set-associative cache. Suppose memory addresses X and Y are referenced by the lower-priority process. After some T cycles, X's and Y's cache lines will decay and become candidates for replacement. With the LRU replacement policy, all memory references will be misses. In contrast, 3 out of 7 memory references are hits with the timekeeping replacement policy. More significantly, these hits are for the high-priority process.**

2.2. Collaborative Parallel Threads in a Parallel Application

Another issue faced by shared cache capacity management techniques is that collaborative threads are treated equally in a parallel application. In existing techniques, for multiprogrammed workloads that consist of several parallel applications, all parallel threads freely compete for the shared cache. This causes sibling threads (threads in the same parallel application) to compete for shared CMP cache space rather than acting cooperatively. In order to tackle this issue, we need novel techniques which recognize collaborative threads in a parallel application and help to reduce the shared cache contention among all active threads. The proposed ATR scheme offers a solution for this issue by identifying high-priority critical threads in a parallel application and allocating the shared cache space preferentially to these threads.

3. TIMEKEEPING TECHNIQUES FOR ORDERING CACHE REPLACEMENT

ATR builds on the cache decay concept [Kaxiras et al. 2001] which was originally proposed to reduce leakage power consumption of cache memories on chip. We first give background on cache decay's original use for leakage control.

3.1. Cache Decay

Cache decay exploits the generational behavior of data stored in a cache. That is, often data in the cache have a burst of frequent reuse due to data temporal locality followed by a period of "dead time" before they are evicted. Although implementations vary, conceptually cache lines are associated with a *decay counter* which is decremented periodically over time. Each time a cache line is referenced, its decay counter is reset back to a *decay interval*, T, which is the number of idle cycles this cache line is allowed to remain in the cache. In other words, if a cache line is frequently referenced, its decay counter will not reach 0. When the decay counter reaches 0, it implies that the associated cache line has not been referenced for the past T cycles and its data are unlikely to be referenced again in the near future. In the original cache decay proposal, this timer would signal when the power supply of the cache line should be turned off

(with loss of data) to save leakage power. Our proposal does *not* turn off the cache line nor lose data; rather, we use the generational timer construct to prioritize data for upcoming evictions as discussed next. Furthermore, as Section 4.4 describes, our implementation has a much lower overhead than the conceptual example's full counter per cache line.

3.2. Shared Cache Capacity Management

In ATR, the generational behavior exploited by timekeeping leakage control helps determine which lines of a cache should be considered for early replacement. ATR suggests to evict the data of low-priority processes more aggressively than that of high-priority processes. This approach can be used to control cache space occupancy of multiple processes.

When a cache line has not been referenced for the past T cycles, it can be thought of as moving to the LRU position in a replacement "queue". That is, it becomes an immediate candidate for replacement regardless of its LRU status. The key observation is that we can use different decay intervals for each process. The decay counters of cache lines associated with high-priority processes receive a longer decay interval, so over time more cache resources are allocated to these processes. Similarly, the system may assign a shorter decay interval to cache lines associated with streaming-memory applications or low-priority processes, in order to release their cache space more frequently. Consequently, higher-priority data tend to stay in the shared cache for a longer period of time.

In Figure 2, assuming a 4-way set-associative cache, we show how cache decay can be individualized to different applications in a mixed workload. Suppose memory addresses X and Y are referenced by the lower-priority process. After some T cycles without being re-referenced, X's and Y's cache lines will decay and become candidates for replacement whether or not they are *least* recently used. With an LRU replacement policy for the example shown in Figure 2, all memory references would be misses. In timekeeping management, 3 out of 7 memory references are hits. More importantly, these hits are for the high-priority process. This is because cache lines associated with the high-priority process do not decay. In this example, LRU replacement works poorly because it treats all memory references equally. In contrast, the timekeeping replacement has more hits because it adjusts replacement based on process priority while responding to data temporal locality. Its fine-grained control in *time* and in *space* enables the effectiveness of the enforcement. Furthermore, unlike a strict priority-based eviction scheme, timekeeping control lets low-priority applications make use of cache space at times when the high-priority applications are not actively referencing their cache lines.

While Petoumenos et al. [2006] explored some of the basic trends for such a technique, they focused on a statistical model that predicts thread behaviors. Our work employs the timekeeping concept of cache decay to cache capacity management considering process priority. We offer direct performance studies, implementation details, and consider operating system effects. Finally and importantly, our work further studies the impact on workloads with parallel applications. The timekeeping replacement ordering coupled with adaptive decay intervals forms the core of ATR.

4. ATR IMPLEMENTATION

This section first introduces the key components in the ATR framework. Then we explain ATR's algorithm and its enforcement for selected performance targets. Finally we discuss the hardware implementation and overhead for ATR.

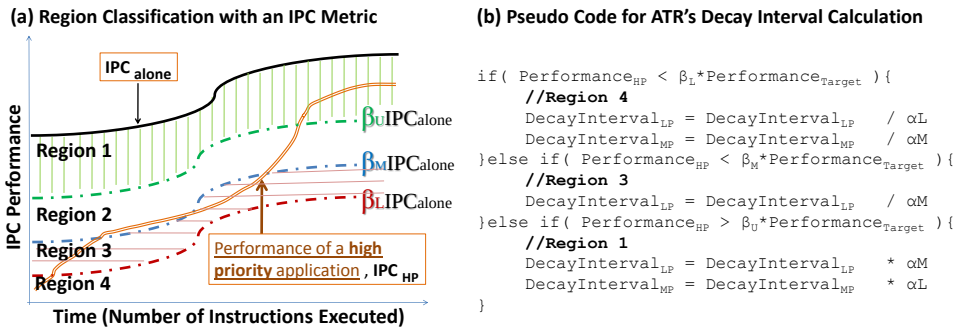

Fig. 3. **(a) Region classification with an IPC target described in Section 4.3.1. (b) Pseudo code for ATR's decay interval calculation.**

4.1. Implementation Overview

In order to adapt to memory needs during different program phases of an application, ATR monitors performance of applications of interest based on process priorities and aims to provide performance expectation defined by shared resource management policies. ATR observes a metric of choice and adjusts decay intervals of applications accordingly. Depending on the metric chosen, the ATR scheme can support different high-level capacity management policies, such as to maximize overall system throughput, to satisfy quality of service goals, and/or to provide fairness in cache sharing. Possible metrics include cache space allocation per process, the number of cache misses, or instructions committed per cycle.

4.2. ATR Algorithm and Enforcement

This section explains ATR's parameters and algorithm in detail. There are three priority levels in the ATR framework: high-priority (HP), medium-priority (MP), and low-priority (LP). Cache lines associated with HP applications never decay, whereas those associated with MP and LP do.

With the three priority levels, there are five parameters. α_M and α_L are weighting factors which determine how fast decay intervals change for MP and LP applications. Cache lines associated with *lower priority processes* have larger α values, so the decay intervals are *decremented faster*. Thus the lower-priority cache lines *move more quickly to the LRU position* in the shared cache. For a system that must support more intermediate priority levels, the ATR scheme can be simply extended to provide more intermediate decay factors. In addition, we define β_U, β_M, and β_L as the tolerance thresholds for a target metric. The β values determine how much performance degradation is tolerable for HP applications.

Typically, the operating system annotates its policies to the underlying hardware mechanisms. As a result, it is the most suitable candidate to set α and β values for active processes based on their priorities. Then the underlying ATR hardware mechanism interprets the policies defined by the operating system and guarantees the quality of service goal for processes in different priority levels. In the case of the ATR scheme in this work, we set the α and β parameters according to application priorities to demonstrate how ATR can effectively enforce the performance goal defined by the preset parameters.

Next, we explain ATR's algorithm. Figure 3(a) depicts four regions into which the performance of the application(s) of interest can fall at any point in time. Figure 3(b) provides the pseudo code for the ATR algorithm. In the example, we use application

IPC to represent the application performance. The different regions in Figure 3(a) correspond to how the HP application's performance compares to a target level.

Region 1 represents a better-than-target performance region, where IPC_{hp} approaches the performance of the application running alone, IPC_{alone}, and is greater than the upper performance tolerance threshold, β_U*IPC_{alone}. Because the performance of the HP application(s) is close to the performance upper bound, ATR can allocate more cache space to the MP and LP applications running concurrently by increasing their associated decay intervals with factors of α_L and α_M respectively.

Region 2 represents a controlled performance region, where IPC_{hp} is between the upper performance degradation tolerance, β_U*IPC_{alone}, and the mid performance degradation tolerance, β_M*IPC_{alone}. When the performance of the HP application(s) falls in region 2, decay intervals of the MP and LP applications can remain unchanged. This is because the performance of the HP application(s) lies in the controlled performance region and meets the QoS goal defined by the β values.

Region 3 represents a slight-degradation performance region. When IPC_{hp} falls in region 3, the HP application(s) experience performance degradation: IPC_{hp} falls in between the mid performance degradation tolerance, β_M*IPC_{alone}, and the lower performance degradation tolerance, β_L*IPC_{alone}. In order to catch up with IPC_{alone}, cache lines associated with LP applications are to be evicted more frequently by a factor of α_M under the ATR scheme. As a result, HP applications can utilize more shared cache space.

Finally, region 4 represents a severe-degradation performance region. When in region 4, IPC_{hp} is classified as severely degraded. Here, ATR will decrease decay intervals of both MP and LP applications by factors of α_M and α_L respectively, so the associated cache lines are evicted more frequently. As a result, the HP application(s) are allocated more shared cache space.

ATR decay interval adjustments continue dynamically either until IPC_{hp} increases or until the MP and LP intervals reach their preset lower bounds. These lower bounds provide worst-case guarantees for MP and LP applications. In general, the HP applications are allocated more shared cache space and can recover from the performance loss caused by the interference of others. On the other hand, the lower bounds stabilize the control algorithm and ensure minimal cache space for MP and LP applications.

Although IPC is used as the performance metric for the performance target in this example, the ATR scheme is flexible and can enforce other performance metrics defined by a specific shared resource policy. Section 4.3 discusses this in more detail.

4.3. Performance Target

The previous example used IPC as the target metric, but ATR is general and can use other performance metrics as well, such as cache miss counts, or cache occupancies. A performance target based on an IPC metric tracks application performance more directly whereas performance targets based on cache resource metrics, such as cache miss counts and cache occupancies, are less direct, but sometimes easier to obtain dynamically. In Sections 4.3.1 and 4.3.2, we explain how the ATR framework obtains its performance target using IPC and cache resource metrics respectively.

4.3.1. IPC Target. Deducing the target IPC for an application can come from an application performance goal (e.g. a soft-realtime video game or media player) or from profiling. We use profiling in our work. The application of interest is first run alone owning the entire shared cache. Every 100 million cycles, its performance in IPC is recorded. This is essentially the most direct method for knowing how the application would run in isolation. When the target application runs concurrently with other applications, its instruction counts are lined up with the counts when it is running alone,

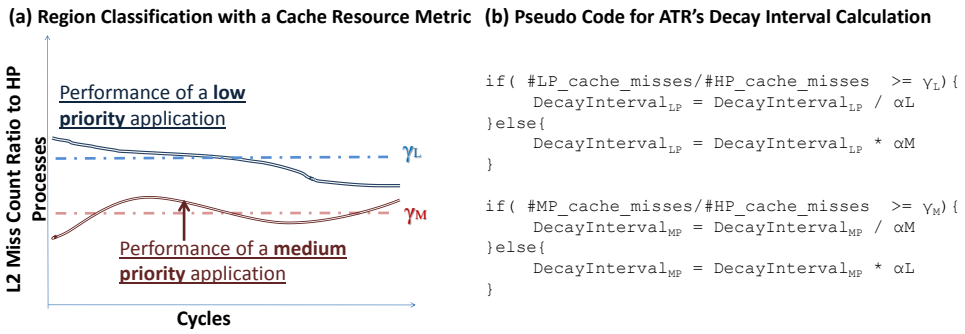

Fig. 4. **(a) Region classification with a cache resource target described in Section 4.3.2. (b) Pseudo code for ATR's decay interval calculation.**

so IPC_{alone} is used for the same program section. Nevertheless, the IPC_{alone} can also be obtained by dynamic performance prediction using, for example, the techniques proposed in [Isci et al. 2006; Tam et al. 2009] with some modification.

For a parallel application, we set the performance target based on the IPC of the critical thread(s). This is because the performance of a parallel application is often bounded by the slowest running threads. There has been other work [Bhattacharjee and Martonosi 2009] which focuses on predicting critical threads in a parallel application based on thread cache behavior at runtime. Our work uses a similar but simpler technique which determines thread criticality by the lowest-IPC thread.

4.3.2. Other Targets. An alternative to using profiled IPC as ATR's performance target is to use other dynamic metrics, such as the number of L2 misses or cache occupancy. Work, such as [Kim et al. 2004; Jaleel et al. 2008], also uses a dynamic cache miss count metric to guide the proposed cache partitioning algorithm. Here we offer an example policy that uses cache miss counts to guide ATR's enforcement mechanism. Similarly, we can fit the performance curves of MP and LP applications at any point in time in Figure 4(a). Here γ_M and γ_L represent the L2 miss count ratio of MP and HP applications and that of LP and HP applications respectively. When an inter-process conflict miss occurs to a HP process, ATR decreases MP's and LP's decay intervals by factors of α_M and α_L. In return, the HP process can utilize more shared cache space. ATR then periodically checks whether the L2 miss proportions of MP/LP and HP processes exceed γ_M and γ_L, the preset target. If it does, the performance of MP or LP processes is degraded more than desired, so ATR will increase MP's and LP's decay intervals to allow more cache utilization. As a result, miss count ratios of MP/LP and HP processes stay close to γ_M and γ_L. Figure 4(b) provides the pseudo code for the decay interval calculation with a cache resource metric.

4.4. Hardware Implementation and Overhead

This section discusses the hardware implementation and overhead for ATR in detail. The ATR hardware interprets process priorities assigned by the operating system to decay threshold with an N-entry lookup table, where N is the number of processors. This small hardware lookup table maps PIDs to decay intervals for all active processes. Then, at every sampling interval, the ATR scheme compares the performance of HP applications to its performance target and the tolerance thresholds as described in Section 4.2. It also calculates the next decay intervals based on the comparison result and the α values. Since we only allow α factors to be multiples of 2, the calculation for new decay intervals is a simple shift and, as a result, this calculation is not a

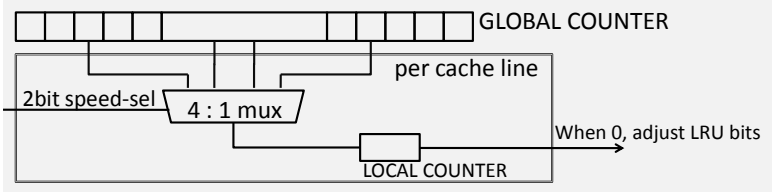

Fig. 5. **Decay counter implementation with 2-bit speed counter per cache line incurs less than 1% overhead for 64-byte cache lines. The overhead is even less for longer cache lines: 0.1% for 256-byte cache lines.**

performance bottleneck. Alternatively, the decay interval calculation is only performed every 10 million cycles, so this calculation can be done in software. In this work, we simulate all modifications in hardware with a full-system simulator (Section 5.1).

For decay counter implementation, the ATR scheme primarily relies on a single global cycle counter. Some of the global counter's middle or high-order bits are used to control the per-line or regional counters in the cache, as discussed in [Hu et al. 2002]. Only a small number of coarse-grained decay bits, e.g. 2 bits, are stored per cache entry along with a subset/hash of PID. In this two-level scheme, when four decay intervals are required (i.e. 64K, 256K, 1M, and 4M), the 4th, 6th, 8th, and 10th bits of the global counter are used to control the local counter. The single global counter decrements every cycle, but local per-entry counters only decrement every 4096 cycles. When a local counter reaches zero, its associated cache line becomes an immediate candidate for replacement. Our work has explored design options and determined that four different decay intervals are sufficient.

When only a few representative decay intervals are required, ATR can be implemented with a 2-bit speed counter per cache line as shown in Figure 5. This incurs less than 1% hardware overhead for 64-byte cache lines. For longer cache lines, the overhead is even less. If more fine-grained decay interval increments are desired, the two-level counter scheme incurs no more than 3% overhead for 64-byte cache lines, which is still reasonably low overhead.

Last but not least, although we do not model this, the same decay counter hardware can be used both for ATR's capacity management and for timekeeping leakage energy control. This amortizes overhead over two benefits and allows systems to optimize for power, cache capacity management, or both.

5. EXPERIMENTAL SETUP

5.1. Simulation Framework

We evaluate the proposed ATR scheme using GEMS [Martin et al. 2005]. GEMS is built upon Virtutech Simics [Virtutech Simics 2010], a full-system multi-core processor simulator. We simulate both 4-core (for sequential workloads) and 16-core (for parallel workloads) CMP systems based on the Sparc architecture running Solaris 10 operating system. We elaborate the workload setup in Section 5.2.

The microarchitectural parameters for our baseline memory sub-system are based on Intel's Core2 Quad chips clocked at 2.83 GHz on a 45 nm technology node [Intel Corp. 2010]. Each processor core has a 4-way set-associative, 32KB private L1 cache with 64B cache lines. All cores share the two 2MB L2 cache banks (4MB in total). The L2 cache banks are 16-way set-associative with 64B cache lines.

CACTI 5.3 determines the L2 cache access latency to be 2.87ns. This means, it takes 8.12 cycles to access the L2 cache. We round up to a higher access latency of 10 cy-

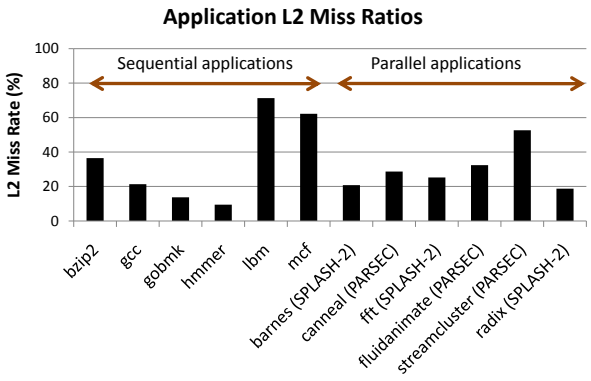

Fig. 6. **Application L2 cache miss ratios of sequential and parallel applications used in our workloads.**

cles for the experiments in the paper to represent CMP systems running at a higher frequency.

GEMS memory modules add 4 more cycles to determine and access the cache banks. Furthermore, a simple interconnection network is modeled, which includes 2 more cycles for network link latency. Thus, overall, the L2 cache access latency is 16 cycles. This also matches well with Core2 Quad's design parameters. The measured L2 cache access latency for Core2 Quad machines is 15 cycles [Intel Corp. 2011]. Similarly, we use CACTI 5.3 to determine the access latencies for other memory components in the system, where L1 cache access takes 3 cycles and DRAM access takes 268 cycles respectively.

5.2. Workload Construction

We use a combination of sequential and parallel applications to evaluate shared resource management mechanisms. The sequential applications come from the SPEC 2006 benchmark suite; the parallel applications are from the SPLASH-2 [Woo et al. 1995] and PARSEC [Bienia et al. 2008] benchmark suites. To better suit today's CMP cache sizes and configurations, the SPLASH-2 applications use the scaled input datasets suggested in [Bienia et al. 2008]. The PARSEC applications use the `simlarge` input datasets. Figure 6 shows the L2 cache miss ratios of sequential and parallel applications used in our workloads. The L2 cache miss ratio represents an application's memory characteristics and is provided as background information to help understand the mixed memory characteristics for the entire aggregate workloads.

Tables I and II then summarize the seven workloads we study. Five of these are multiprogrammed mixtures of sequential applications (SPEC2006); the other two workloads are multiprogrammed mixtures of parallel applications. The workloads representing multiprogrammed mixes of sequential and parallel applications are intended to illustrate several scenarios.

First, Workloads 1 and 2 are general sequential scenarios. Here, we generate heterogeneous memory requests to the shared cache by running `gcc`, `gobmk`, and `lbm` in Workload 1 and `bzip2`, `gcc`, and `mcf` in Workload 2. Each of the three applications is pinned to an individual core (in the 4-core CMP system). The remaining core is dedicated for operating system activities, so the OS interference on the running applications is minimized.

Workloads 3 and 4 are general purpose parallel workloads. Workload 3 consists of three parallel applications, `fft`, `barnes`, and `radix`, from the SPLASH-2 benchmark

Table I. Workloads reflect general sequential and general parallel scenarios. The application priority in all workloads is randomly chosen to mimic the OS priority assignment.

Workload 1: General, Sequential			
SPEC2006	gcc	gobmk	lbm
Priority Level	HP	MP	LP
Workload 2: General, Sequential			
SPEC2006	bzip2	gcc	mcf
Priority Level	HP	MP	LP
Workload 3: General, Parallel			
SPLASH-2	barnes	fft	radix
Priority Level	HP	MP	LP
Input Dataset	65,536 particles	4,194,304 points	8,388,608 points
Domain	scientific computing	scientific computing	sorting
Workload 4: General, Parallel			
PARSEC	streamcluster	canneal	fluidanimate
Priority Level	HP	MP	LP
Input Dataset	simlarge	simlarge	simlarge
Domain	data mining	engineering	animation

Table II. Workloads reflect high-contention scenarios. The application priority in all workloads is randomly chosen to mimic the OS priority assignment.

Workload 5: High Contention, Sequential				
SPEC2006	bzip2	gcc	mcf	lbm
Priority Level	HP	HP	LP	LP
Workload 6: High Contention, Sequential				
SPEC2006	gcc	mcf	mcf	mcf
Priority Level	HP	LP	LP	LP
Workload 7: High Contention, Sequential				
SPEC2006	hmmer	bzip2	gobmk	gcc
Miss Ratio	HP	MP	MP	LP

suite. Each is run with four threads and each thread is pinned to an individual core in the 16-core CMP system. The second multithreaded workload includes three different parallel applications, `canneal`, `streamcluster`, and `fluidanimate`, from the PARSEC benchmark suite. Similarly, each application is run with four threads and each thread is pinned to an individual core.

Finally, Workloads 5-7 model the high-contention scenarios to the shared cache. These high-contention scenarios are the ones that most fundamentally motivate cache resource management approaches. Workload 5 models high contention memory access to the shared cache by running `bzip2`, `gcc`, `mcf`, and `lbm`, where both `mcf` and `lbm` are memory-bound applications. Workload 6 also models the high contention scenario by including one instance of `gcc` along with three instances of `mcf`. The first three applications are pinned to individual cores in the 4-core CMP system and the last application share its core with OS activities. Finally, the last high-contention workload, Workload 7, consists of four SPEC2006 applications, `bzip2`, `gcc`, `gobmk`, and `hmmer`. This workload is used to demonstrate ATR's enforcement for both an IPC and a dynamic cache miss metric. Table III summarizes ATR's configurations for all workloads.

5.3. Performance Metric

In the multiprogrammed sequential workloads, Workloads 1, 2, 5, 6, and 7, we measure the performance in IPC of each application for the same program section following the method discussed in Section 4.3.1. We show speedup for each application individually. We also compute a weighted speedup metric related to application priorities. The

Table III. ATR configuration for all workloads. In a full implementation, the OS would set and adjust these parameters.

IPC Control	α_L	α_M	β_L	β_M	β_H
Sequential Workloads	4	2	0.93	0.96	0.97
Parallel Workloads	4	2	0.90	0.95	0.97
	α_L	α_M	γ_L	γ_M	γ_H
Cache Miss Control	4	2	4	2	N/A

Table IV. STR configuration for all workloads. Application priorities are given in Tables I and II. HP application cache lines are never decayed.

	LP Decay Interval (in cycles)	MP Decay Interval (in cycles)
Workload 1	4096	8192
Workload 2	4096	1,048,576
Workload 3	1,048,576	8,388,608
Workload 4	1,048,576	4,194,304
Workload 5	4,096	8,192
Workload 6	4,096	N/A
Workload 7	262,144	4,194,304

speedup of high-, medium-, and low-priority applications are weighted 3:2:1. Furthermore, for Workload 7, we also use a dynamic cache resource metric (in number of L2 cache misses) to measure the performance of each application individually in addition to weighted mean.

For Workloads 3 and 4 that consist of parallel applications, we run each application to completion and use execution time speedup as the performance metric. If one of the applications finishes earlier than other applications in the workload, it restarts and reruns until all applications finish at least once. This ensures a degree of multiprogrammed contention throughout the full simulation.

6. PERFORMANCE EVALUATION

We investigate a static, non-adaptive scheme, called Static Timekeeping Replacement (STR). Then we compare the performance results for STR and ATR to a baseline system which uses the LRU replacement policy for its level-two cache. Furthermore, we implement another recently proposed cache capacity management scheme, TADIP-F [Jaleel et al. 2008]. We extend TADIP-F to account for application priorities and give a detailed comparison to ATR in Section 6.4.

6.1. STR Scheme

We first consider STR, a static version of ATR, in order to show the importance of ATR's ability to adaptively change decay intervals at runtime. For STR, we experiment with 36 pairs of decay intervals for MP and LP applications ranging from 4K cycles to 64M cycles. Based on these results, we pick the single decay interval combination giving the best weighted average performance for applications. Thus, the STR results show an idealized best-case performance for a static decay interval, and can be used as a comparison point. Table IV summarizes the decay interval combinations chosen.

6.1.1. **General, Sequential Workloads: Workloads 1 and 2**. Figure 7 shows that, for all workloads except Workload 2, STR significantly improves performance compared to the unmanaged baseline. In Workload 1, STR preferentially allocates more shared cache space to the high-priority applications, `gcc`. In return, the performance of `gcc` and `gobmk` is improved by 17% and 16% relative to the unmanaged case although the performance of `lbm` is degraded by 1%, as shown in Figure 8(a). This minimal performance tradeoff of `lbm` and significant performance gain of `gcc` and `gobmk` comes from

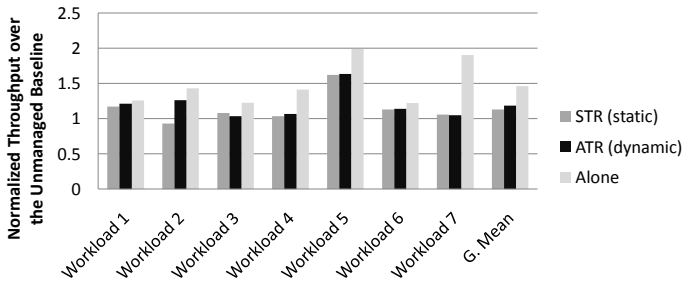

Fig. 7. **Performance comparison for all workloads under the unmanaged baseline, STR, and ATR schemes. We also illustrate the performance of the workload would attain if each application runs alone. ATR outperforms the unmanaged baseline by as much as 1.63X and by an average of 1.19X.**

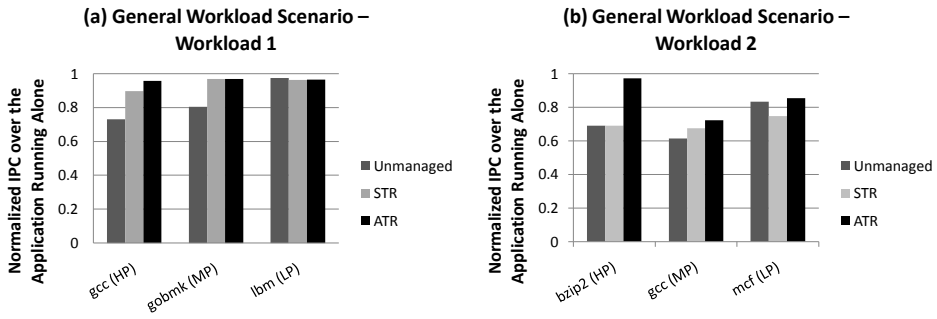

Fig. 8. **Performance comparison for (a) Workload 1 and (b) Workload 2.**

the fine-grained capacity control of STR and its ability to eliminate memory interferences in the shared cache between `lbm` and the other applications.

However, for Workload 2, STR is not flexible enough and cannot improve the performance of the high-priority application as illustrated in Figure 8(b). For workload 2, the performance of `bzip2`, the high-priority application, remains the same as it is in the unmanaged baseline. Furthermore, the performance of `mcf`, the low-priority application, is degraded by 9%. This is because STR is unable to adjust the decay intervals to adapt to changes in memory needs over the program execution. ATR's dynamic decay interval selection addresses this problem.

6.1.2. **General, Parallel Workloads: Workloads 3 and 4**. Next, we investigate STR's effectiveness for workloads comprised of parallel applications, Workloads 3 and 4. As illustrated in Figure 9, when the shared cache is unmanaged, `radix` uses most of the shared cache over the program execution, leaving almost no cache space for `fft` and very little for `barnes`, the high-priority application. In STR, `radix`'s cache space occupancy is constrained when other higher priority applications request more shared cache space. In return, the performance of `fft` and `barnes` is improved by 5% and 1%. More importantly, because of the fine-grained control of the STR scheme, the performance of `radix` experiences no additional degradation compared to the baseline scheme. Similarly, in Workload 4, the STR scheme can improve the performance of `streamcluster`, the high-priority application, and `canneal` by 4% and 13% respectively without additionally harming the performance of `fluidanimate`, the low-priority application. Figure 10 illustrates this.

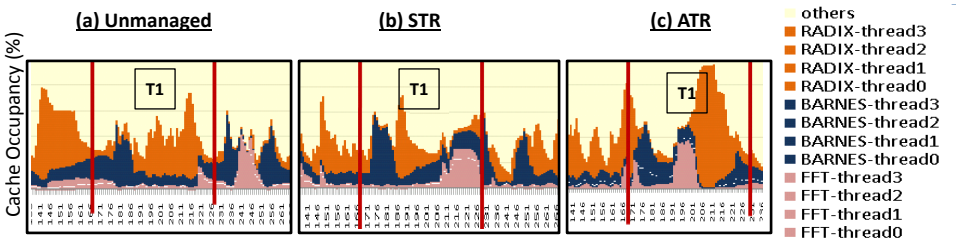

Fig. 9. **Cache space utilization for Workload 3 under (a) the baseline scheme, (b) STR, and (c) ATR. During time *T1*, ATR prefers to retain `barnes` (the HP application) cache lines, then `fft`'s (the MP application), and last `radix`'s (the LP application) as shown in (c), whereas the allocation in the baseline scheme is on-demand (by LRU).**

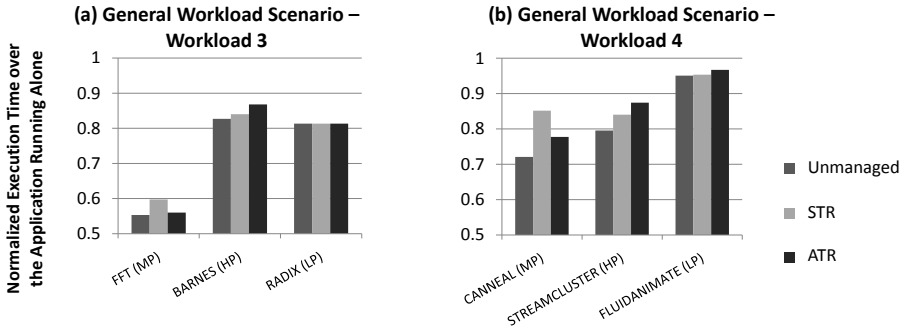

Fig. 10. **Performance comparison for Workloads 3 and 4.**

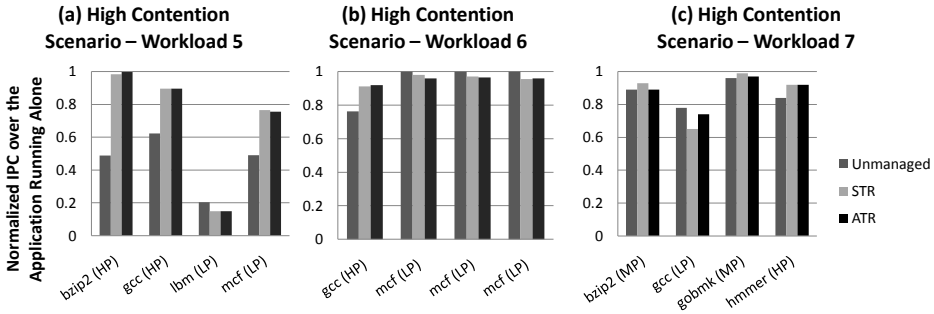

Fig. 11. **Performance comparison for (a) Workload 5, (b) Workload 6, and (c) Workload 7.**

6.1.3. **High Contention, Sequential Workloads: Workloads 5-7**. Finally, we illustrate that STR can improve the performance of Workloads 5-7 in the high contention scenarios significantly. In Workload 5, STR preferentially allocates more shared cache space to the high-priority applications, `bzip2` and `gcc`, while retaining data that exhibit good temporal locality from `lbm` and `mcf`. In return, the performance of `bzip2` and `gcc` is improved by 50% and 27% relative to the unmanaged case although the performance of `lbm` is degraded by 5%, as shown in Figure 11(a). More interestingly, the performance of the other low-priority application, `mcf`, is *improved* by 28%. This results in a net average performance improvement. If `lbm`'s small performance degradation is problematic,

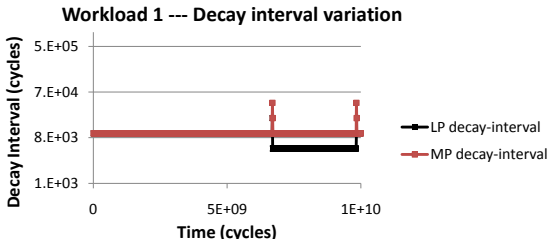

Fig. 12. **Decay interval variation by ATR for Workload 1.**

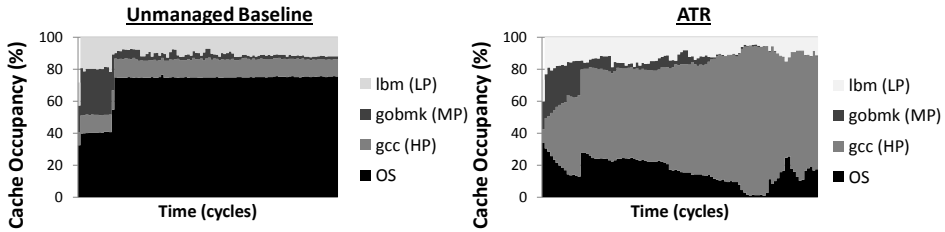

Fig. 13. **Cache space distribution over time for Workload 1: (a) under the baseline scheme and (b) under the ATR scheme.**

the operating system could categorize it as an MP application, which would help protect it from `mcf`'s heavy use of the cache. Another alternative is to let the operating system decrease the α values at runtime. Thus `lbm`, the LP application in the workload, is allowed to use more shared cache. The minimal performance tradeoff of `lbm` and significant performance gain of `mcf` comes from the fine-grained capacity control of STR and its ability to eliminate memory interferences in the shared cache between `lbm` and all other applications. We see similar performance result for Workloads 6 and 7 as illustrated in Figures 11(b) and (c).

While STR improves the performance for all workloads except Workload 2, its inability to adjust decay intervals on the fly results in less effective memory allocation. As a result, the performance of the high- and the medium-priority applications is slightly degraded in Workload 2. Furthermore, to determine the optimal combination of the decay intervals for MP and LP applications in the STR scheme requires multiple profiling runs. This imposes a constraint on STR's feasibility. In the following section, we will demonstrate ATR can detect the change of memory needs for all applications at runtime, dynamically adjust decay intervals, and, as a result, allocate the shared cache space more effectively to all running applications.

6.2. ATR Scheme

ATR's dynamic decay interval adjustments reduce the need for profiling and increase its ability to exploit application program phases. When the shared cache is managed under the ATR scheme, Figure 7 shows that ATR offers a significant performance improvement across all workloads. The performance benefit is especially large for workloads where memory needs change over different program phases.

6.2.1. **General, Sequential Workloads: Workloads 1 and 2**. When compared to the STR scheme, ATR can further improve the performance of `gcc`, the high-priority application in Workload 1, by additional 6% without degrading the performance of the other applications. Figure 12 shows that this is because ATR can allocate cache memory effectively based on its performance monitoring by quickly adjusting the decay intervals

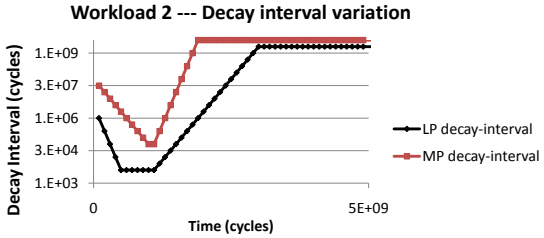

Fig. 14. **Decay interval variation by ATR for Workload 2.**

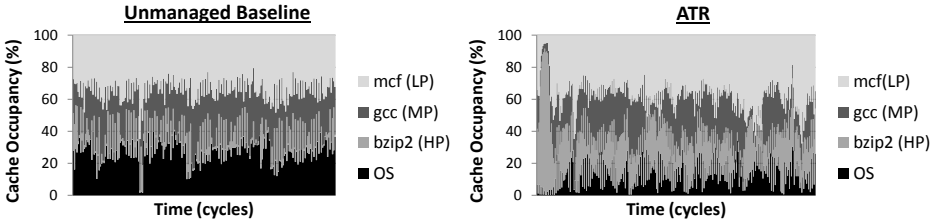

Fig. 15. **Cache space distribution over time for Workload 2: (a) under the baseline scheme and (b) under the ATR scheme.**

of the MP and LP applications. As a result, the high-priority application receives more cache space in the ATR technique [Figure 13].

With Workload 2, we demonstrate ATR's ability to adapt to memory needs of all applications over different program phases is critical and essential. When `bzip2`, the high-priority application, is running along with `gcc` and `mcf` in Workload 2, its performance is degraded by 31%. This performance degradation is caused by the memory interference in the shared cache. When `bzip2`'s memory requirement decreases, ATR immediately increases the decay intervals associated with `gcc`'s and `mcf`'s cache lines as illustrated in Figure 14, so more shared cache can be utilized by `gcc` and `mcf`. Figure 15 illustrates that when the shared cache is managed under the ATR scheme, `bzip2`, the high-priority application, immediately receives more shared cache space as required throughout the execution. Then, when its memory need decreases, ATR immediately allows the other applications to use more shared cache space. As a result, the performance of `bzip2` and `gcc` is improved significantly by 28% and 11% respectively in the ATR scheme. In contrast, STR's inability to adapt to different program phases results in slight performance degradation in Workload 2.

6.2.2. **General, Parallel Workloads: Workloads 3 and 4**. Next, we explore how the ATR scheme manages the shared cache space to improve the performance of workloads comprised of parallel applications, especially for the high-priority ones. Figures 16 and 17 depict the performance of critical threads in Workloads 3 and 4 over time. As discussed in Section 4.3.1, the ATR logic monitors the performance of the critical threads for the high-priority applications, `barnes` in Workload 3 and `streamcluster` in Workload 4. Once it observes a slowdown beyond the tolerance threshold, the ATR hardware immediately decreases the decay intervals of other running applications.

Figure 17(b) illustrates that, during time interval *T1*, for Workload 4, the performance of the critical thread of `streamcluster` is rising, so the ATR logic increases the decay intervals for `canneal` and `fluidanimate`. This allows both applications to utilize more shared cache space. Then, during time interval *T2*, the performance of the critical thread is declining. As a result, ATR decreases decay intervals of `canneal` and

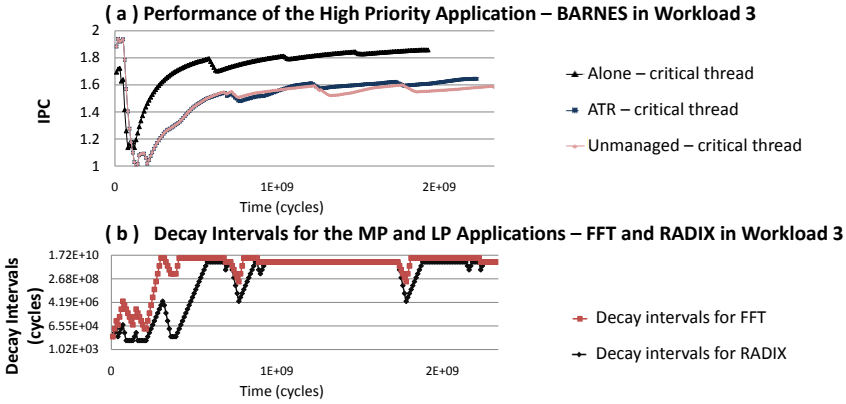

Fig. 16. **(a) The IPC performance of `barnes` critical thread, the high-priority application in Workload 3, over time in the baseline and ATR schemes. (b) The decay interval variation for `fft` and `radix` in Workload 3.**

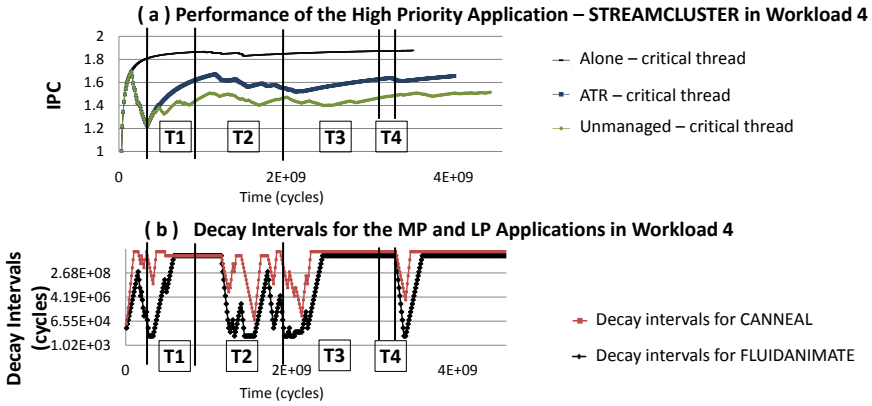

Fig. 17. **(a) The IPC of `streamcluster`'s critical thread over time under the baseline and ATR schemes. During time *T1*, the performance of the critical thread of `streamcluster` is rising, so the ATR logic increases decay intervals for `canneal` and `fluidanimate` as shown in (b). During time *T2*, the performance of the critical thread is declining, so the ATR logic decreases decay intervals of other running applications. As a result, cache lines associated with `canneal` and `fluidanimate` move to the LRU position more quickly, similarly for time *T3* and *T4*.**

`fluidanimate`, so cache lines associated with the two applications move to the LRU position more quickly. As a result, more shared cache space is allocated to the high-priority applications as requested. Consequently, the performance of the high-priority applications in Workloads 3 and 4 is improved by 4% and 7% respectively.

Although the performance of `fft` in Workload 3 and `canneal` in Workload 4 is improved more in the STR scheme, the performance improvement of the high-priority applications is more significant in the ATR scheme. This is because with statically assigned decay intervals in the STR scheme, `fft` and `canneal` are able to use more shared cache space leaving `barnes` and `streamcluster` with insufficient cache share over certain program phases. In contrast, ATR is able to observe these subtle changes

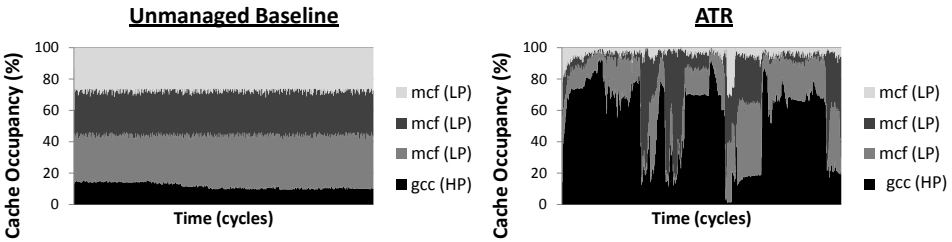

Fig. 18. **Cache space distribution over time for Workload 6: (a) under the baseline scheme and (b) under the ATR scheme.**

in memory needs over time and adjust the decay intervals accordingly. Consequently, the performance of both `barnes` and `streamcluster`, the high-priority applications, is improved by an additional 3% compared to the improvement in the STR scheme.

6.2.3. **High Contention, Sequential Workloads: Workloads 5-7**. Finally we demonstrate that compared to STR, ATR can further improve the performance of Workloads 5-7 in the high contention scenario. ATR works better than (or as well as) STR and improves the performance of Workloads 5-7 by 1.64X, 1.07X, and 1.06X respectively (Figure 7). ATR generally better protects the performance of the high-priority applications in the workloads.

The additional performance gain for the high-priority applications comes from ATR's ability to adapt to the memory needs of all applications at runtime. The ATR mechanism first assigns an initial decay interval to each MP and LP application. Then during the workload execution, ATR quickly settles to the decay intervals that can be used to achieve the performance target. When the IPC target is met, ATR increases decay intervals for the medium- and low-priority applications. Figure 18 illustrates that ATR can preferentially allocate more shared cache space to the high-priority application in Workload 6. Consequently, ATR further improves the performance of the high-priority application in the workloads.

6.3. Dynamic Miss Tracking

As discussed in Section 4.3.2, ATR can also be guided by target metrics other than IPC. Here as one example, we consider cache miss counts. Figure 19 illustrates the L2 miss reduction for a workload (Workload 7) which consists of four SPEC2006 applications, `bzip2`, `gcc`, `gobmk`, and `hmmer`. The left bar shows the number of L2 cache misses under no management, the middle bar shows the management scheme under ATR's enforcement with β_M=2 and β_L=4, and the right bar shows the number of L2 misses when each application is running alone. ATR reduces the number of L2 misses of the high-priority application, `hmmer`, by 27%. Furthermore, the overall L2 misses is reduced by 11%. Here ATR is demonstrated as an effective, and yet flexible mechanism for shared cache capacity management using dynamic cache miss tracking.

6.4. Comparison to TADIP-F and TADIP-FP

Another cache capacity management scheme, called Thread-Aware Dynamic Insertion Policy with Feedback (TADIP-F) has been proposed recently [Jaleel et al. 2008]. Both TADIP-F and ATR perform capacity management for shared caches by modifying the cache insertion/replacement policy. The original TADIP-F work does not take into account application priorities in managing the shared cache. Here we take a step further to extend the original TADIP-F work and propose a version of TADIP-F which encodes

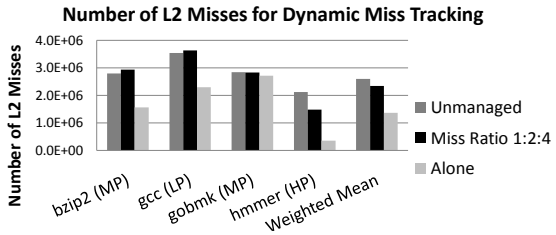

Fig. 19. **Performance comparison for ATR under the guidance of a cache resource metric: L2 miss counts.**

application priorities (TADIP-FP). Then we compare the three schemes to demonstrate ATR's strengths.

6.4.1. **TADIP-F**. TADIP-F observes the memory requirement of all running applications and determines the insertion position of an incoming cache line by set-sampling. It proposes to insert all incoming cache lines to two possible insertion positions: either the LRU or MRU position. After insertion, LRU management occurs. For example, for a cache-thrashing application, the majority of its incoming cache lines are placed in the LRU position and a few in the MRU position. This insertion policy is called bimodal insertion policy (BIP). On the other hand, for a LRU-friendly workload, TADIP-F always inserts its incoming cache lines in the MRU position. This policy is called most-recently-used insertion policy (MIP).

TADIP-F uses a few cache sets to sample workloads running simultaneously and vary a per-core policy selection (PSEL) counter to track misses in each insertion policy. A miss in the MRU-insertion cache sets increments the PSEL counter and a miss in the bimodal-insertion cache sets decrements the PSEL counter. The rest of the cache sets use the most-significant-bit (MSB) of the PSEL counters to determine the insertion policy: MSB=1 uses BIP and MSB = 0 uses MIP. In contrast, the ATR scheme takes access timing into account by keeping track of the time interval since the last access to cached data. We compare TADIP-F with the ATR scheme and demonstrate the importance of this *temporal* aspect of the ATR scheme, particularly for the studied parallel workloads.

6.4.2. **TADIP-FP**. Since the original TADIP-F does not consider process priorities, we also implement a priority-aware version of TADIP-F, called TADIP-FP. TADIP-FP varies how fast PSEL counters are incremented and decremented based on application OS priorities. The PSEL counter for high-priority application is incremented by 1 as it is in TADIP-F, but it is decremented faster by a constant larger than 1. As a result, TADIP-FP uses MIP for the high-priority application more frequently. On the other hand, the PSEL counter for low-priority application is incremented faster by a constant larger than 1 and is decremented as it is in TADIP-F. This means TADIP-FP is more inclined to use BIP for the low-priority application.

In our experiments, we use the weights for high-, medium-, and low-priority applications to determine the rate of change for the per-core PSEL counters. For high-priority applications, we increment the PSEL counters by 1 when a miss occurs in the cache set sampling for MIP and decrement the PSEL counters by 3 when a miss occurs in the cache set sampling for BIP. As a result, TADIP-FP is more likely to select MIP for the high-priority applications. For medium-priority applications, TADIP-FP increments the PSEL counters by 2 and decrements the PSEL counters by 2 also. Finally, for low-priority applications, TADIP-FP increments the PSEL counters by 3 when a miss occurs in its cache sets sampling for MIP and decrements the PSEL counters by

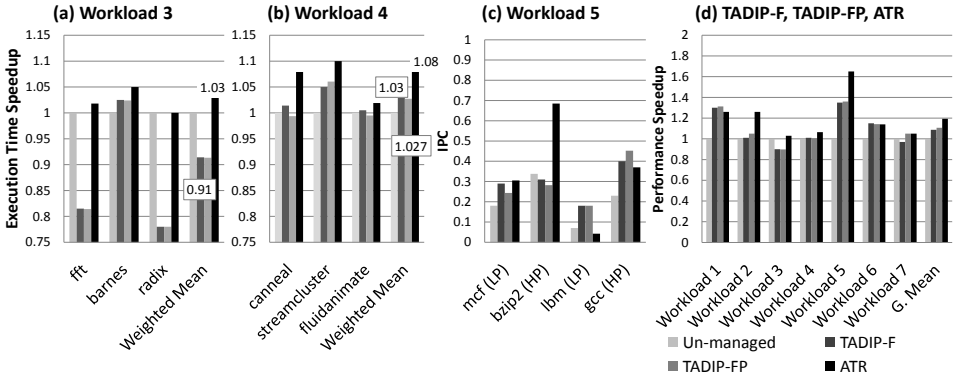

Fig. 20. **Comparison of TADIP-F and the ATR scheme. (a) and (b) The ATR scheme outperforms TADIP-F for parallel applications in Workloads 3 and 4. (c) The performance of `bzip2`, the high-priority application in Workload 5, is doubled in the ATR scheme than that in TADIP-F. (d) The ATR scheme performs better than TADIP-F on average for the studied workloads.**

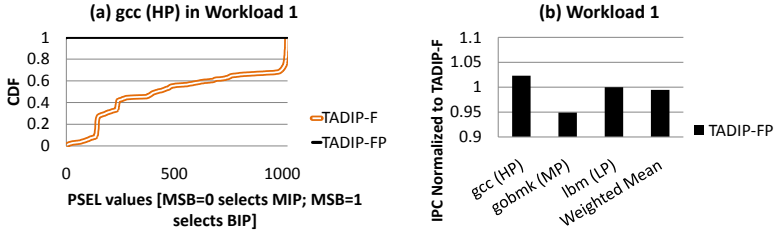

Fig. 21. **(a) Cumulative density function for the PSEL counter of `gcc`, the high-priority application, in Workload 1. TADIP-FP selects to use MIP for the high-priority application more frequently than TADIP-F. (b) Performance comparison for TADIP-F and TADIP-FP. TADIP-FP improves the performance of the high-priority application by 2.5% more than TADIP-F.**

1 when a miss occurs in its cache sets sampling for BIP. As a result, TADIP-FP selects BIP for the low-priority applications more frequently than TADIP-F.

6.4.3. **Comparative Performance Results**. Figure 20 compares the overall performance for all workloads under TADIP-F, TADIP-FP, and ATR. ATR performs better than TADIP-F by 9% on average. More importantly, TADIP-F does not account for OS-imposed priorities as ATR does. Figures 20(a) and (b) show that ATR outperforms TADIP-F for parallel applications in Workloads 3 and 4 by 15% and 4% respectively. Furthermore, the performance of two parallel applications in Workload 3, `fft` and `radix`, is degraded severely under TADIP-F. This is because (1) not all cache sets in TADIP-F use the optimal insertion policy, and (2) threads in the same parallel application also have to compete for shared cache in TADIP-F. Figure 20(c) illustrates that the performance of `bzip2`, the high-priority application in Workload 5, is 21% better in the ATR scheme than in TADIP-F. This is because ATR preferentially allocates more shared cache space to the high-priority application(s) whereas TADIP-F only observes memory footprint characteristics and optimizes for the overall system throughput, but does not aim to provide preferential cache allocation based on process priority.

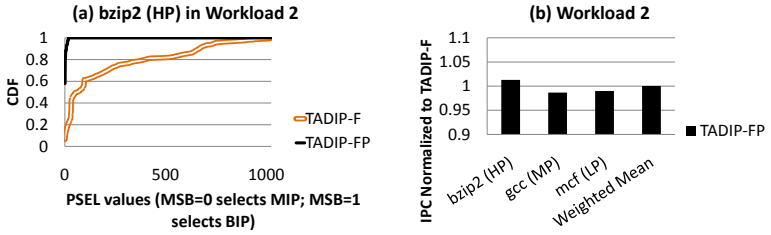

Fig. 22. **(a) Cumulative density function for the PSEL counter of `bzip2`, the high-priority application, in Workload 2. TADIP-FP selects to use MIP for the high-priority application more frequently than TADIP-F. (b) Performance comparison for TADIP-F and TADIP-FP. TADIP-FP improves the performance of the high-priority application by 1.5% more than TADIP-F.**

Next, we investigate the performance results in the priority-aware TADIP-FP technique. For Workloads 1 and 6, TADIP-FP improves the aggregate performance the most, by 28% and 14%. For all other workloads, ATR performs better. Although TADIP-FP prioritizes high-priority cache lines, for a low-priority cache line which is inserted to the LRU position followed by a cache hit, TADIP-FP promotes this low-priority cache line to the MRU position. From here on, such low-priority cache lines are treated no differently from a high-priority cache lines. As a result, the performance benefit gained by high-priority applications is still limited in the TADIP-FP scheme.

Figure 21(a) shows the cumulative density function (CDF) for the PSEL counter values of the high-priority application, `gcc`, in Workload 1. TADIP-F selects to use BIP for `gcc`'s cache lines for 55% of the program run while TADIP-FP always selects to use MIP for `gcc`'s cache lines. Figure 21(b) illustrates that, as a result, TADIP-FP improves the performance of the high-priority application, `gcc`, by 2.5% over TADIP-F. This is because more high-priority cache lines are inserted in the MRU position.

Similarly, Figure 22(a) shows the CDF for the PSEL counter values of the high-priority application, `bzip2`, in Workload 2. TADIP-F selects to use BIP for `bzip2`'s cache lines for 19% of the program run while TADIP-FP again always selects to use MIP for `bzip2`'s cache lines. As a result, TADIP-FP improves the performance of the high-priority application by 1.5% over TADIP-F as illustrated in Figure 22(b). Although TADIP-FP can help improve the performance of the high-priority applications in both Workloads 1 and 2, the overall (weighted) performance does not improve much compared to TADIP-F. On average, TADIP-FP improves workload performance by 1% more than TADIP-F.

In summary, although TADIP-F can be extended for priority level consideration by weighting misses of high-priority applications more heavily (for example, TADIP-FP), there are only two insertion position for incoming cache lines: MRU or LRU. If a low-priority application's cache line is first inserted to the LRU position and then re-referenced, TADIP-F promotes it to the MRU position. It then must step back through the position to LRU before eviction. On the other hand, with ATR, low-priority cache lines become immediate candidates for eviction based on their decay counters even when they are in MRU position. Therefore, ATR can better manage the shared cache capacity taking into account the temporal behavior of data and process priority while also improving the system throughput.

6.5. Sensitivity to Decay Intervals

Although temporal adaptivity is clearly important, we observe that decay intervals used in most of the workloads settle to a few possible values in various program phases.

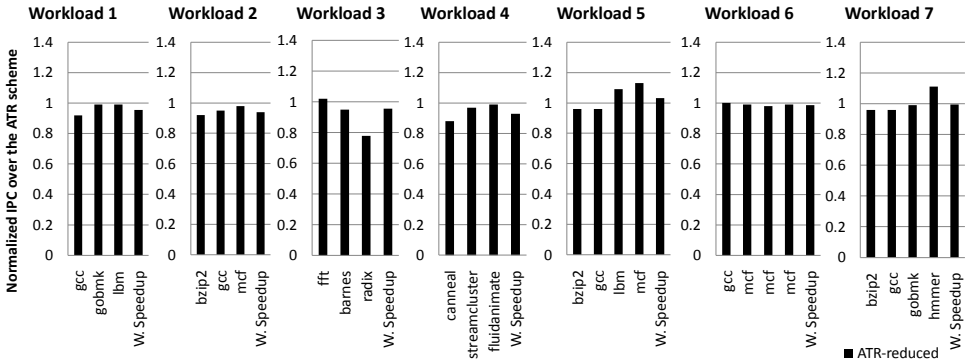

Fig. 23. **Comparison of the ATR and ATR-reduced schemes. Performance here is normalized to the ATR scheme, and the performance of ATR already represents an improvement over the baseline in all cases.**

Thus, it may be sufficient for the ATR scheme to support just these representative decay intervals which can ease ATR's temporal control for cached data. This $ATR_{reduced}$ scheme implements four possible decay interval options for all cache lines: 4K cycles, 32K cycles, 1M cycles, and never decay. Figure 23 shows the performance comparison of the ATR and $ATR_{reduced}$ schemes. On average the $ATR_{reduced}$ scheme performs similarly as the ATR scheme for all sequential workloads. We note, however, that the coarser-grained control does not improve the performance of the high-priority application(s) as much as the ATR scheme.

ATR also performs better than the $ATR_{reduced}$ scheme (7% and 6% respectively) for the parallel workloads, Workloads 3 and 4. This is because the memory needs in parallel applications change more frequently and more diversely as illustrated previously in Figures 16(b) and 17(b). Nonetheless, overall $ATR_{reduced}$ offers competitive performance improvement as ATR. For all workloads, the performance gain under the $ATR_{reduced}$ mechanism is only 3% worse than ATR's. In general, $ATR_{reduced}$ represents a simpler control implementation and similar performance to ATR.

6.6. Summary

This work proposes a priority-aware capacity management approach, ATR. We demonstrate that ATR's fine-grained cache capacity management in space and time is effective and important in improving the performance of both sequential and parallel workloads. ATR outperforms an unmanaged baseline by as much as 1.63X and by an average of 1.19X. We take a step further to implement a priority-aware version of TADIP-F, called TADIP-FP. Our results show that ATR performs better than both TADIP-F and TADIP-FP by an average of 9% and 8% respectively. Finally, we show, with a simpler ATR implementation featuring four decay intervals, $ATR_{reduced}$ can sustain ATR's significant performance improvement with only 3% performance tradeoff on average.

7. RELATED WORK

There has been a significant amount of research in capacity management for shared CMP caches [Bitirgen et al. 2008; Hsu et al. 2006; Iyer 2004; Jaleel et al. 2008; Kim et al. 2004; Nesbit et al. 2007; Petoumenos et al. 2006; Qureshi and Patt 2006; Rafique et al. 2006; Suh et al. 2001; Suh et al. 2002; Zhao et al. 2007]. However, the underlying capacity control mechanisms used in these proposals are often based solely on spatial partitioning of shared caches. To the best of our knowledge, our proposed ATR scheme

is the first shared cache capacity management that takes into account not only spatial allocation of shared caches and temporal characteristics of cached data but also application priorities. Furthermore, this is the first detailed study of shared cache capacity management considering thread behaviors in parallel applications.

Qureshi et al. [2006] proposed Utility-based Cache Partitioning which monitors resource utilization among all running processes in flight and distributes the shared cache accordingly using way-partitioning. CacheScouts [Zhao et al. 2007] offered a finer-grained monitoring for shared caches but it also uses spatial partitioning for its shared caches. Similarly, Nesbit et al. [2007] proposed virtual private caches which consist of a bandwidth manager and a capacity manager implementing way-partitioning. Furthermore, Bitirgen et al. [2008] also uses spatial partitioning as its underlying mechanism for the shared cache while coordinating other on-chip shared resources: shared caches, cache bandwidth, and power budgets. To investigate the effectiveness of spatial partitioning, Iyer [2004] demonstrates that spatial partitioning of shared caches in various granularity can help achieve QoS goals more effectively for CMP platforms. Moreover, Kim et al. [2004] discussed five performance metrics for CMP cache sharing which help the operating system determine shared cache allocation better and offered two algorithms that also suggest to spatially partition the shared cache. We argue that the proposed ATR scheme can provide a more effective shared cache capacity allocation because not only is it capable of managing shared caches spatially in the granularity of cache lines, but it also takes the timing behavior of cached data into account. It is promising both as a stand alone capacity management scheme, as well as in concert with other resource management scheme discussed above.

Srikantaiah et al. [2008] proposed adaptive set pinning to identify and eliminate inter-process misses in CMP systems. This work is complementary to the proposed ATR scheme and can be used to further improve system throughput. Petoumenos et al. [2006] offered a statistical model to predict thread behaviors in a shared cache based on cache decay while no direct performance studies nor implementation details are given for managing the shared cache. Jaleel et al. [2008] proposed TADIP-F, as discussed in Section 6.4. Xie and Loh [2009] extended the idea in TADIP-F for shared cache capacity management and showed that the proposed pseudo-partitioning scheme allows better utilization of the shared cache. Finally, Jaleel et al. [2010] proposed RRIP which modifies cache replacement policies based on cache line reuse interval prediction. While this work is similar to ours, ATR is distinct in its application priority handling. Furthermore, this work is the first quantifying the need for fine-grained capacity management for parallel workloads.

8. DISCUSSION AND FUTURE WORK

In this section, we describe the role of the operating system in general for capacity management schemes and specific to the proposed ATR scheme. In addition, we discuss other design choice for cache memory organizations and the impact for cache capacity management studies. Finally we list the future direction for this work.

8.1. Operating Systems

Operating systems apportion system resources to running processes based on the assigned priorities. Traditionally that has included CPU time slices, but our work shows that shared CMP caches must similarly be managed. While the operating system offers more flexibility in defining quality of service goals, it plays a significant role in annotating its policies or specific goals to the underlying hardware mechanisms. The underlying hardware mechanisms then interpret the policies defined by the operating

system and guarantee some level of quality of service by adjusting shared resource allocation.

In the case of ATR, the operating system can specify the α and β values based on its QoS goals. The underlying hardware can vary decay intervals for each process on the fly to satisfy OS policies. It is interesting to consider whether there is benefit to having more priority levels than what is currently supported by ATR. On the other hand, for processes with equal priorities, ATR does not treat them differently. A potential future work is to explicitly assign the α and β values based on application cache utilization for processes in the same priority level.

8.2. Non-Uniform Cache Access Design

Non-uniform cache access (NUCA) design is becoming more prevalent for today's memory systems and poses an interesting challenge in cache capacity management techniques. The traditional monolithic last-level cache (LLC) is separated to a few cache banks enabling more parallelism accessing the cache bank modules. However, this results in non-uniform cache access latencies for the shared LLC. Accessing cache banks located closer to a particular processor takes less time than other banks located further on the chip.

This NUCA design certainly influences the cache capacity management problem and motivates new approaches to cache partitioning. Cache capacity management schemes should take into account cache bank access time when apportioning the shared LLC. This can further optimize for application performance and system throughput. However, NUCA or the traditional uniform cache access (UCA) design is orthogonal to both ATR and TADIP-F, which focus on improving application performance via modifying cache insertion and replacement policies. As a result, we do not further investigate this cache design choice in this work, but leave it as potential future work.

8.3. ATR for Shared versus Private Data in Parallel Applications

The proposed ATR scheme is the first work quantifying an approach for cache capacity management for parallel applications. Currently, ATR, without special treatment, can effectively improve parallel thread performance with its fine-grained capacity allocation. However, a more complicated ATR scheme can be designed to take into account the data sharing characteristics: *shared* versus *private* data in a parallel application. For example, assigning a longer decay interval to shared data can help reduce the eviction rate of *shared* L2 cache blocks and their corresponding L1 cache blocks. As a result, the amount of coherence messages in the interconnection network between L1 and L2 caches can be significantly reduced. Furthermore, this can also reduce the overall application cache miss rate. Although the current ATR scheme does not take advantage of the data sharing property in parallel applications, this opportunity remains for future work.

9. CONCLUSION

In this work, we propose ATR, a fine-grained capacity management for shared caches, based on timekeeping techniques. ATR monitors the performance of high-priority application(s) and dynamically apportions shared cache space among running processes based on memory footprints and process priorities. We evaluate the proposed approach including operating system effects. We demonstrate that ATR outperforms a baseline system by as much as 1.63X and by an average of 1.19X for all studied workloads. Furthermore, we study ATR's effectiveness on workloads consisting of parallel applications. ATR can improve application performance over unmanaged scenarios by accelerating critical threads; overall application performance is improved by as much as 7%. ATR's fine-grained temporal control is particularly important for parallel applica-

tions. We take a step further to implement a priority-aware version of TADIP-F and show that ATR performs better than both TADIP-F and TADIP-FP by 9% and 8% respectively. Overall, ATR is an effective mechanism for capacity management in shared caches, because of its fine temporal and spatial control of the shared cache space. While we have demonstrated ATR's effectiveness in isolation, it can also be viewed as a building block to be used along with other prior work on cache and network bandwidth management.

ACKNOWLEDGMENTS

We thank Konstantinos Aisopos, Yu-Yuan Chen, Daniel Lustig, Jakub Szefer, and the anonymous reviewers for their feedback. We also thank Aamer Jaleel and Joel Emer for their useful feedback and insights related to this work. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0627650 and CNS-07205661. The authors also acknowledge the support of the Gigascale System Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

REFERENCES

BHATTACHARJEE, A. AND MARTONOSI, M. 2009. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*.

BIENIA, C., KUMAR, S., AND LI, K. 2008. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*.

BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

BITIRGEN, R., IPEK, E., AND MARTINEZ, J. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO '08: Proceedings of the 41st Annual International Symposium on Microarchitecture.*

CHANG, J. AND SOHI, G. S. 2007. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st Annual International Conference on Supercomputing*.

HSU, L. R., REINHARDT, S. K., IYER, R., AND MAKINENI, S. 2006. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*.

HU, Z., KAXIRAS, S., AND MARTONOSI, M. 2002. Let caches decay: Reducing leakage energy via exploitation of cache generational behavior. *ACM Transactions on Computer Systems*.

INTEL CORP. 2010. Intel Core2 Extreme Processor QX9000 Series and Intel Core2 Quad Processor Q9000, Q9000S, Q8000 and Q8000S Series Datasheet.

INTEL CORP. 2011. Intel 64 and IA-32 Architectures Optimization Reference Manual.

ISCI, C., CONTRERAS, G., AND MARTONOSI, M. 2006. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO '06: Proceedings of the 39th Annual International Symposium on Microarchitecture*. 359–370.

IYER, R. 2004. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*.

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. *ACM SIGMETRICS Performance Evaluation Review*.

JALEEL, A., HASENPLAUGH, W., QURESHI, M., SEBOT, J., STEELY, JR., S., AND EMER, J. 2008. Adaptive insertion policies for managing shared caches. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*.

JALEEL, A., THEOBALD, K. B., STEELY, JR., S. C., AND EMER, J. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*.

KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*.

KIM, S., CHANDRA, D., AND SOLIHIN, Y. 2004. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*.

MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*.

NESBIT, K. J., LAUDON, J., AND SMITH, J. E. 2007. Virtual private caches. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*.

PETOUMENOS, P., KERAMIDAS, G., ZEFFER, H., KAXIRAS, S., AND HAGERSTEN, E. 2006. Modeling cache sharing on chip multiprocessor architectures. In *IISWC '06: Proceedings of the IEEE International Symposium on Workload Characterization*.

QURESHI, M. K. AND PATT, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO '06: Proceedings of the 39th Annual International Symposium on Microarchitecture*.

RAFIQUE, N., LIM, W. T., AND THOTTETHODI, M. 2006. Architectural support for operating system-driven CMP cache management. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*.

SRIKANTAIAH, S., KANDEMIR, M., AND IRWIN, M. J. 2008. Adaptive set pinning: Managing shared caches in CMPs. In *ASPLOS '08: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operation Systems*.

SUH, G., DEVADAS, S., AND RUDOLPH, L. 2002. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*.

SUH, G. E., RUDOLPH, L., AND DEVADAS, S. 2001. Dynamic cache partitioning for simultaneous multithreading systems. In *IASTED '01: Proceedings of the International Conference on Parallel and Distributed Computing and Systems*.

TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*.

VIRTUTECH SIMICS. 2010. http://www.virtutech.com/.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*.

XIE, Y. AND LOH, G. H. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*.

ZHAO, L., IYER, R., ILLIKKAL, R., MOSES, J., MAKINENI, S., AND NEWELL, D. 2007. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*.

Received September 2009; revised September 2010; accepted December 2010