Intra-Application Cache Partitioning

Sai Prashanth Muralidhara, Mahmut Kandemir, Padma Raghavan
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA 16802, USA
{smuralid, kandemir, raghavan}@cse.psu.edu

***Abstract*—Efficient management of shared on-chip resources such as the shared level 2 (L2) cache has become an important problem with the emergence of chip multiprocessors (CMPs). Partitioning the shared cache in chip multiprocessors (CMPs) among concurrently executing applications can provide important benefits such as throughput improvement, fairness guarantees, and quality of service (QoS) enhancements. In this paper, we pose an interesting related question, which is, if partitioning the shared cache space among concurrently executing threads of the same application can enhance the application performance. We address this problem by identifying and speeding up the slowest thread, also termed as the critical path thread, during each execution interval since the overall performance of a multithreaded application is determined by the critical path thread. To do so, we propose a dynamic, runtime system based, cache partitioning scheme that partitions the shared cache space dynamically among the individual threads of a given application. In a nutshell, we wish to take some cache space away from the faster threads and give it to the critical path thread at each execution interval. We show that speeding up the critical path thread this way, results in overall performance enhancement of the application execution in the long term. Our experimental evaluation indicates that, the proposed dynamic cache partitioning scheme yields benefits up to 15% over a shared cache with no partitions, up to 23% over a statically partitioned cache (private cache) and up to 20% over a throughput-oriented scheme.**

I. Introduction

As chip multiprocessors (CMPs) continue to grow in terms of number of on-chip cores [33], [24], [4], [15], prudent management of shared on-chip resources has become crucial for improved performance. One such on-chip resource common in today's CMPs is the shared level 2 (L2) cache[1]. Shared L2 cache has become the dominant on-chip cache alternative, thanks to its efficiency in utilization and flexibility in dynamic allocation [15], [24]. However, due to its shared nature, more than one thread can contend for the cache space at the same time. This contention for the shared cache from different threads (executing on different cores) can have varying impact on those threads. In such a scenario, performance of one or more of the contending threads can be adversely affected [18], [12]. The commonly used LRU replacement policy can lead to threads with not so good cache behavior occupying most of the shared cache with very little performance gain, while other threads with possibly good cache behavior starve [17], [22]. Therefore, efficient and smart management of an important shared resource such as L2 cache is vital. There have been various techniques proposed to limit such adverse performance impact and improve the overall throughput [29], [30], [17], [22]. Some of these techniques try to partition the shared cache space among different concurrently-executing applications to avoid contention and isolate the individual application performance. There have also been several schemes proposed to look at the fairness and QoS issues that arise in such cache partitioning techniques [18], [10].

In this paper, we study the cache space partitioning problem when the contending threads belong to the same application. That is, as opposed to existing cache partitioning schemes, we investigate *intra-application* cache partitioning. Most parallel applications targeting shared memory systems are programmed to contain one or more parallel sections, which are bound by synchronization constructs such as barriers. Performance of a parallel section is always determined by the slowest thread, also called the *critical path thread*. A thread with excellent cache behavior does little to speed up the application performance if the other application threads have really poor cache behavior. When large multithreaded applications are executed on multicore machines, such differences in thread behavior can severely limit the performance benefits of utilizing large number of cores. Efficacy of previously described cache space partitioning and management techniques is questionable in this scenario, where the contending threads belong to the same application. This is because, those techniques are agnostic to inter-thread relationships and, consequently, try to either optimize throughput or maintain cache fairness. However, as we mentioned above, performance of a multithreaded application is largely determined by the performance of the critical path thread and therefore most of the prior techniques prove ineffective in improving the application performance. We claim that, when partitioning the shared cache among the threads of the same application, the goal should be neither throughput improvement nor fairness, but in fact speeding up the critical path thread.

An important point to consider when contending threads belong to the same application is the net effect of such cache

[1]A number of commercial CMPs such as Intel Dunnington [5] have a shared L3 cache as well. Our work can target any shared cache component in the chip.

contention. For starters, threads from the same application can share data, and consequently, effects of contention need not always be adverse. When threads share data, a simple shared cache without any sort of partitioning may perform well in certain cases, due to possible constructive sharing. In fact, absence of cache data sharing in private caches appears to be a major drawback for single multithreaded application execution on such systems. On the flip-side, in shared cache architectures, inter-thread cache evictions (one thread evicting the data brought in by a different thread) happens to be a major performance concern for most parallel multithreaded applications. Note here that, when threads belong to different applications, there is rarely any inter-thread data sharing that has to be taken into account when deciding the cache space partitioning. We show later that, when the threads belong to the same application, a partitioned shared cache is a much better alternative than either a shared cache or a private cache.

In this paper, we propose a dynamic, runtime system based *cache partitioning* scheme to partition a shared cache among threads of a single application to enhance the overall performance of the multithreaded application. We propose to dynamically determine the cache requirements of individual threads based on their cache performances and partition the cache space such that the slowest thread gets most of the cache space. We discuss two such dynamic, run-time system based cache partitioning techniques. The first technique partitions the cache space at each execution interval based on the cycles-per-instruction (CPI) values of the individual threads, such that the slowest thread (critical path thread), which is the thread with the highest CPI, gets a larger portion of the cache. The second technique dynamically builds a cache model through curve fitting for each thread and finds a cache partition that minimizes the slack time at each interval. In this context, slack time is defined as the difference between thread speeds. The goal of this algorithm is to try to ensure that the threads progress at approximately same speeds by allocating lesser cache to threads with good cache hit rates and more cache to threads with low cache hit rates.

We evaluate our cache partitioning technique against both unpartitioned shared cache and statically partitioned cache for several applications from the NAS parallel and SPEC OMP benchmark suites [1], [2]. We observe that our technique achieves application speedups up to 15% over an unpartitioned shared cache, up to 23% over a statically partitioned cache, and up to 20% over a prior throughput-oriented scheme.

To summarize, in this paper, we make the following contributions:

• We motivate the intra-application cache partitioning problem by presenting the differences in execution behavior of different threads that belong to the same application.

• We argue that prior cache paritioning schemes are not very effective in the intra-application case, i.e., when a cache space needs to be partitioned across the threads of the same application.

• We propose a dynamic cache partitioning scheme to effectively partition the cache when the contending threads belong to the same application and show that the proposed scheme is very effective in practice.

The rest of the paper is organized as follows. Section II presents a summary of the related work. Section III provides a brief background and introduces our experimental setup. We motivate the cache partitioning problem in Section IV. Sections V talks about the cache partitioning alternatives. Section VI presents our dynamic model based cache partitioning scheme. We present the experimental evaluation in Section VII, and conclude the paper in Section VIII.

II. Related Work

There have been several research efforts to optimize shared cache management in CMPs at the architectural level [35], [9], [17], [22], [8], [16], [26]. Zhang and Asanovic propose to use a small victim cache to improve the performance of the shared L2 cache in CMPs [35]. Chang and Sohi discuss a unified, cooperative caching scheme to combine the advantages of both shared and private caches [9]. Their scheme forms an aggregate shared cache through cooperation among individual private caches. Qureshi et al show that LRU replacement policy can result in thrashing at times, and propose a dynamic insertion policy [17]. They further extend their work with a thread aware dynamic insertion policy for managing shared caches by taking in to account individual application memory requirements [22]. Beckmann et al propose that, by monitoring the workload behavior and by using controlled selective cache block replication, application performance can be significantly improved [8]. Hsu et al discuss various policies that can be employed in CMP shared cache management [16]. They try to optimize different metrics in each of their policies such as throughput, fairness and alike. Zhang et al elaborate on the benefits of OS level page coloring schemes in cache management [36]. Soares et al propose to indentify last level cache pollution using a dynamic operating system scheme and further maintain a small buffer to optimize the cache management [25]. Chen et al propose a sharing aware cache management scheme [20]. Rafique et al discuss a architectural support for OS management of shared caches after recognizing the tradeoff between replacement policy flexibility and performance in shared caches [23]. Chakraborty et al propose a utility based caching scheme for storage caches [27].

Apart from these shared cache management schemes, there have been efforts to partition shared caches in CMPs. Seminal work on cache way partitioning was published by Suh et al [28]. They propose to use a low overhead memory monitoring system to record cache hit rate when the cache size is varied. Suh et al also discuss a dynamic cache partitioning scheme for CMP/SMT, with the aim of reducing the total number of misses and, show that they could enhance the prevalent LRU replacement policy [29], [30]. They use cache performance monitoring to dynamically partition the cache. Chang and Sohi present a cooperative cache partitioning scheme, where, cache contention problem is resolved with multiple time-sharing cache partitions [10] . They guarentee a certain degree of quality of service (QoS) for each executing thread. While most of the previous cache partitioning schemes have taken throughput alone into consideration, Kim et al consider fairness in their cache partitioning scheme [18]. They focus on both static and dynamic cache partitioning algorithms to optimize fairness. They also provide a detailed study of the relationship between fairness and throughput. Lin et al present a OS based software approach to support static and dynamic cache partitioning through memory address mapping [19]. There have also been previous efforts towards developing power saving schemes based on dynamic cache resizing. These schemes dynamically turn off selective cache ways, thereby, saving power [7], [13]. Further, Liu et al exploit slack time and barrier synchronization to implement a power saving scheme [37].

Our cache partitioning scheme differs from these studies in several aspects. Firstly, we consider cache partitioning for a single, multithreaded application. The objective of our scheme is to improve the performance of a single multithreaded application by speeding up the critical path thread during each execution interval. We achieve this by a dynamic cache partitioning scheme, which tries to give more cache ways to the slowest thread (the critical path thread), during each execution interval. Our work is complimentary to existing inter-application cache partitioning schemes. Specifically, we envision a hierarchical system where the OS performs the inter-application cache partitioning and the runtime system implements our intra-application dynamic cache partitioning scheme. We eloborate on implementation details in Section VI-C.

III. Background and Setup

A. Architecture Specification

In this work, we consider a CMP with a shared level 2 (L2) cache. Our target system, unless otherwise mentioned, is a four core CMP, with an L2 cache shared by all cores. This L2 cache is assumed to be highly associative. Each of the cores also maintains private L1 data and instruction caches. We want to make it clear that, whenever we talk about cache partitioning in this paper, we always refer to cache way partitioning [29]. Therefore, assigning more cache resources to a thread is synonymous to assigning more cache ways to the thread in our context. Also, in this paper, we generally use the terms ”thread” and ”core” alternatively since we consider a single thread running on each core unless otherwise mentioned.

B. Parallel Program Structure

Most multithreaded shared-memory parallel programs generally have several parallel code sections interspersed with sequential sections. Sequential sections might perform tasks such as data initialization and data collation. Also, synchronization might be needed between any two sections to maintain data integrity and avoid race conditions. For such synchronization purposes, these parallel sections employ constructs such as *barriers*. Figure 1 shows the structure of a typical parallel program (on the left) and also depicts the execution progress of a sample parallel section (on the right), assuming four threads. Execution proceeds from a parallel section to the next stage (possibly sequential) only after all threads in the parallel section complete, and consequently, reach the barrier. The execution time of such a parallel section is determined by the time taken for the slowest thread to reach the barrier, also termed as the *critical path thread*. The threads that reach the parallel section barrier early stall until the critical path thread also completes and reaches the barrier. As can be seen from Figure 1, thread 1 has reached the barrier, which marks the end of the parallel section, while the other three threads are executing at various points in the parallel section. Now, thread 1 has to stall (wait) until the other three threads proceed at different speeds and reach the barrier.

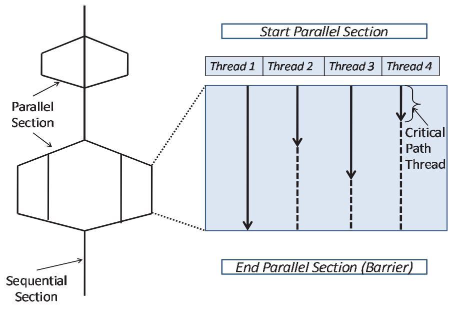


Figure 1. Left: A sample execution for shared-memory multithreaded application. Right: Progress of threads in a parallel section at a particular point during execution.

C. Experimental Setup

Processor	UltraSparc 3
Number of cores	4
Number of threads	4
Core Frequency	1 GHz
Operating System	Sun Solaris 9
L1 cache associativity	4
L1 cache size	8 KB
L2 cache type	Shared
L2 cache associativity	64

Figure 2. Default system configuration used.

We use Simics [21], which is a full system simulator for implementation and performance evaluation purposes. We simulate a four core CMP system, with the cores based on the UltraSparc 3 architecture [15]. The details of the system we simulate are shown in the table in Figure 2. Each of these four cores maintain a private L1 cache of size 8 KB and the cores share the L2 cache, which is 1MB. We run the Solaris operating system on Simics [21]. Also, we bind individual threads to the cores on the CMP using the Solaris system calls.

In this paper, we use nine applications from SPEC OMP [2] and NAS [1] benchmark suites for evaluation purpose. These are parallel multithreaded applications, that use OpenMP parallelization library. We divide the application execution into intervals of 15 million instructions and run the applications for 50 execution intervals.

IV. Motivation

In this section, we present the motivation for our intra-application dynamic cache partitioning scheme. We try to answer three questions in the process of motivating the problem at hand. Firstly, we present an argument in favor of addressing the intra-application cache partitioning problem. Secondly, we argue that prior cache partitioning schemes are not expected to be very effective in handling the intra-application case. Thirdly, we discuss why our dynamic cache partitioning scheme can be beneficial in the intra-application partitioning case.

A. Why Intra-Application Cache Partitioning?

Runtime behavior of parallel multithreaded applications exhibit certain characteristics that make them amenable to intra-application cache partitioning. In this section, we identify those characteristics and describe how each of those characteristics motivates the need for intra-application cache partitioning.

1) Performance Variability: In Section III-B, we discussed the structure of a typical parallel program and the possible variability in execution speeds of the different threads. In this section, we analyze with empirical support, if this is indeed the case. Figure 3 shows the overall performance (we consider the inverse of execution time as performance). of each of the four threads of nine parallel benchmarks over 50 execution intervals. The performance of the application threads are *normalized* to the fastest thread in Figure 3. As we can see, the applications exhibit a wide variability in the performance of the threads. More importantly, in every application, the critical path thread (represented by the lowest bar) is considerably slower than the other threads and hence determines the performance of the overall application. In general, variation in performance (slack time) among the application threads is very high. For instance, in MGRID, although thread 3 performs exceedingly well with a CPI of 7.1, the application performance is held back by thread 2, which has a comparably poor CPI value of 11.5.

In order to explore the reasons for this variability among the application threads, we also collected cache performance statistics for all the above applications during their runs. Figure 4 plots the cache performance of the four application threads in terms of the L2 misses incurred by each of the threads. The values are *normalized* to the thread with the highest number of L2 misses. As we can clearly see from Figures 3 and 4, the variability in the overall performance of these threads correlates very closely with the variability in their cache performances. More specifically, if the performance of a thread is low in Figure 3, its cache miss count is high in Figure 4, and vice versa. This correlation between the CPI values and the number of cache misses can be clearly seen in Figure 5. This figure plots the correlation coefficient between the number of L2 cache misses and the CPI value for the applications in our experimental suite. We can infer, from this plot, a strong linear dependence between number of L2 cache misses and CPI for these applications, with an average correlation coefficient value of 0.97.

Further, we plot the performance of the individual threads of SWIM application over 50 contiguous execution intervals in Figure 6. It can be observed from this graph that, in addition to the variability across the performances of individual threads, there is also a variation across execution intervals. This is due to the fact that a multithreaded application, typically goes through different phases during its execution [11]. There can be various reasons for this phase behavior. During one stage, a thread can be memory bound with poor cache performance; during another, it can have really good cache performance; some parts may not be memory intensive at all; some other parts can have poor branch predictor performance [11]. In order to identify the main factor, we plot the L2 cache misses corresponding to thread 2 in Figure 6(b) during the same 50 contiguous execution intervals in Figure 7. As before, we observe a clear correlation between the CPI across time and the corresponding cache misses during the same time interval. Therefore, due to this variability across time, the critical path thread may change from one execution phase to another.

To summarize, different threads belonging to the same application have very different cache requirements from one another and further, these cache requirements also vary over time.

2) Cache Interaction Across Threads: Application threads executing on different cores sharing a cache can interact in different ways. In this section, we describe and quantify both the amount and the kind of cache interactions exhibited by the application threads. Firstly, we ran our nine applications on the target CMP and collected the inter-thread cache interaction statistics. By *inter-thread cache interaction*, we mean the percentage of cache accesses that are inter-thread accesses. In this context, we specify a cache access to be an inter-thread cache interaction if a previous access to the same cache line was from a different thread. On the other hand, if two contiguous accesses to a cache line are from the same thread, then it is an intra-thread interaction. Recall that we use the terms ”thread” and ”core” interchangeably since we consider a single thread

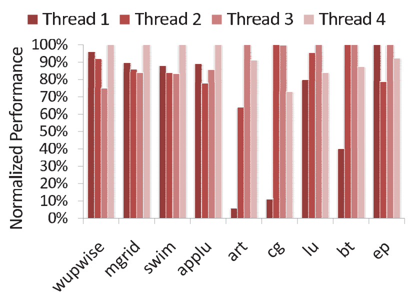


Figure 3. Performance of individual threads of the application normalized to the fastest thread.

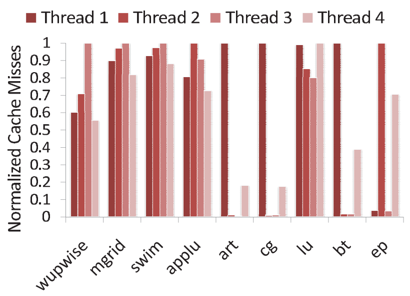


Figure 4. Number of L2 misses incurred by each individual thread normalized to the thread with highest misses.

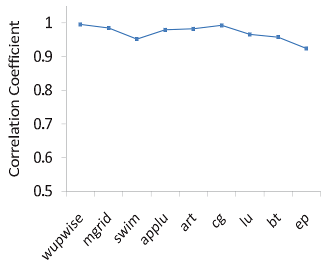
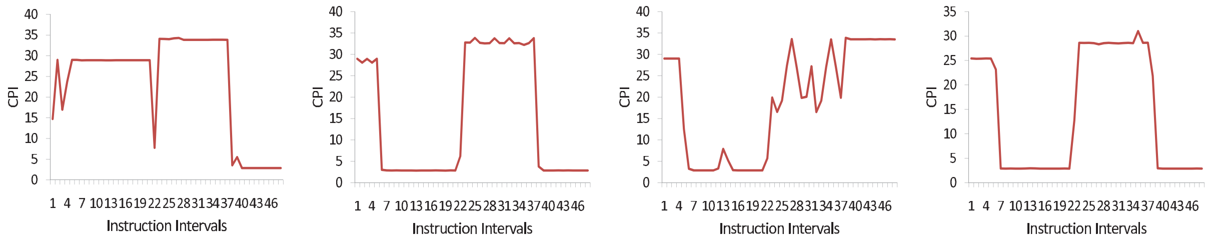


Figure 5. Correlation coefficient between the number of L2 cache misses and the corresponding CPI values.



(a) Thread 1. (b) Thread 2. (c) Thread 3. (d) Thread 4.

Figure 6. CPI values of the four SWIM application threads during 50 consecutive execution intervals.

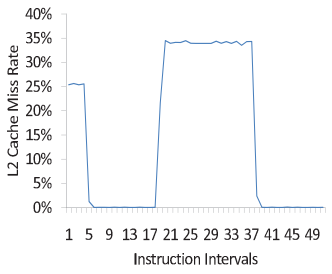


Figure 7. L2 misses during same 50 execution intervals of thread 2 of SWIM benchmark as in Figure 6(b) .

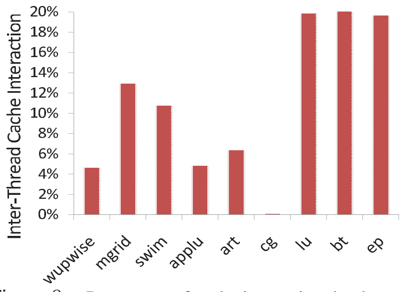


Figure 8. Percentage of cache interaction that happens to be inter-thread.

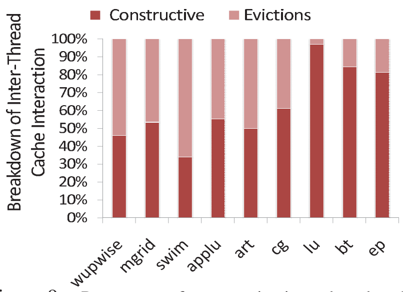
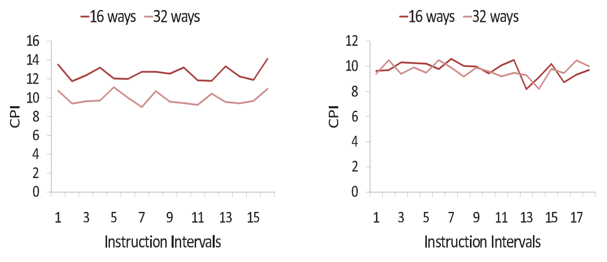


Figure 9. Percentage of constructive inter-thread cache interactions.

executing on each core. Another important point to note here is that cache interaction covers all accesses and not just misses. Figure 8 shows the contribution of inter-thread cache interactions (when considering all interactions). As we can clearly see, there is a considerable amount of inter-thread interaction in these multithreaded applications, averaging about 11.5% of all cache interactions.

We also studied the nature of these inter-thread interactions. We describe *constructive* inter-thread cache interaction to be the percentage of inter-thread interaction that happen to be cache hits. Therefore, a set of two contiguous accesses to a cache line is considered to be constructive interaction if the second access is a hit. To rephrase, constructive interaction happens when a data element brought into the cache by a thread is also used by another thread before it gets displaced, thereby helping the latter thread to improve its performance. It is to be noted that, constructive inter-thread cache interaction is a result of data sharing between the interacting threads. We plot the breakdown of constructive and destructive (evictions) inter-thread interactions in Figure 9. We can infer from this graph that, not all inter-thread interactions are constructive. A significant amount of destructive inter-thread interactions in the form of evictions can be seen in Figure 9. An inter-thread eviction happens when a thread evicts a cache line which was previously accessed by a different thread. Clearly, inter-thread eviction is an adverse consequence of cache line contention which results in destructive cache interaction, thereby, hampering performance. A simple shared cache can take advantage of constructive cache sharing. However, a shared cache is also susceptible to destructive interactions. A private cache, on the other hand, prevents destructive interactions but it fails to exploit constructive interactions. More specifically, in a private cache organization, data sharing results in data



(a) Thread 1 with 16 and 32 ways. (b) Thread 2 with 16 and 32 ways.

Figure 10. CPI curves for two threads of SWIM when executed with 16 and 32 ways. Clearly, thread 1 shows considerably more improvement when the number of ways is increased from 16 to 32, when compared to very little improvement exhibited by thread 2

replication across caches. Therefore, a partitioned shared cache is the solution to exploit data sharing and at the same time inhibit destructive inter-thread interactions. In such a partitioned shared cache, a thread can access a cache line present in a cache partition belonging to another thread, thereby, enabling constructive inter-thread cache sharing. However, a thread cannot evict a cache line belonging to another thread's cache partition, thereby, preventing destructive inter-thread cache interference. In other words, cache space partition is in terms of eviction control. Implementation of the above mentioned partitioned shared cache will be explained in detail in Section V.

3) Cache Sensitivity Variability: Another facet of a parallel program's behavior is the cache sensitivity of individual threads of the program. We ran a four-threaded SWIM application multiple times, but each time using different cache sizes. We increased the cache size from 32 KB progressively until 1MB. We want to reiterate that, to increase cache size, we simply add more ways. Therefore, the increase in the total cache size is accompanied by a corresponding increase in associativity. For instance, a 32KB cache is 2 way associative, but a larger 64KB cache will be 4 way associative. We show the CPI for two of the threads of this application for 16 and 32 ways in Figure 10. An important observation that can be made is that, the individual threads have variable sensitivity to cache capacity. While a cache size increase may benefit one thread, it might not improve another thread's performance. In Figure 10, when the cache size is increased from 16 to 32 ways, thread 1 exhibits a higher CPI reduction, as compared to almost no CPI reduction in the case of thread 2. Therefore, thread 1 is more sensitive to cache than thread 2 since thread 1 benefits a lot more by cache size increase than thread 2. This heterogeneity in cache sensitivity among threads of the same application is an important observation because taking cache resources away from a cache insensitive thread may not be detrimental, as far as overall performance is considered. Therefore, cache resources can possibly be taken away from a cache insensitive thread without much affect on its performance. On the flip side, if the critical path thread is cache insensitive, then giving more cache ways to it might not be too beneficial in practice. Therefore, cache sensitivities of threads dictates how effective a cache partitioning scheme can be.

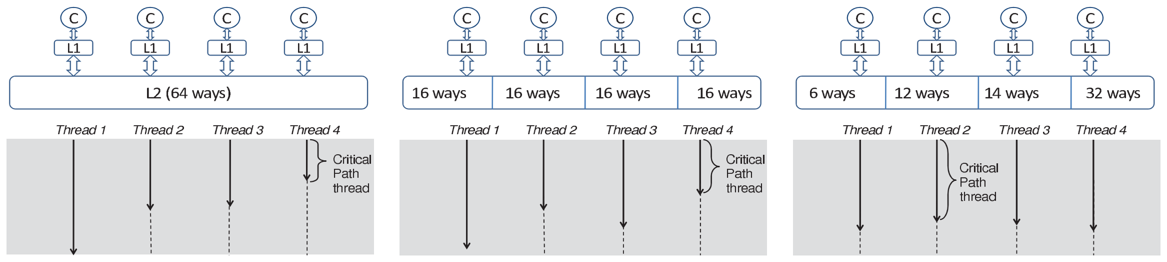
B. Why Not Prior Cache Partitioning Schemes?

All of the prior cache partitioning schemes can be classified into two categories. The throughput oriented schemes try to improve throughput by partitioning the cache among different application threads. The second category tries to maintain fairness. Most existing throughput oriented schemes [29], [30], [35], [9], [17], [22], [8], [16] assign more cache space to the thread that best utilizes it, thereby, improving the overall throughput. Such a scheme can be very beneficial in the multiprogramming based environment. However, when the threads belong to the same application, in the process of improving throughput, these schemes can end up improving the performance of non-critical threads, which may not have much impact on application performance. In other words, most of the existing schemes try to improve a global metric such as throughput without caring about the thread relationships. In the process, there is no targeted effort to improve the performance of an individual application locally, although overall processor metrics such as throughput can be significantly improved.

On the other hand, fairness oriented schemes [10], [18] try to ensure that the impact of cache sharing is uniform for all the threads in a shared cache environment with LRU. This is akin to mimicking a private cache configuration. These schemes ensure all threads including the critical path thread make balanced progress. In [10], Chang and Sohi allocate a bigger partition of the cache to one of the threads. However, each of the threads gets that bigger partition for a fixed quantum in a round robin fashion, thereby improving fairness. In comparison, speeding up the critical path thread can be more beneficial in the intra-application case. We compare our scheme to a throughput oriented scheme later.

C. Why Dynamic Critical Path Cache Partitioning?

In this section, we use an illustrative example to explain why a dynamic cache partitioning scheme targeted at speeding up the critical path thread (reducing the slack time) can bring in benefits in the intra-application case. Figure 11 shows execution progress at a particular time point of a sample four threaded application and the cache allocated to each of the four threads. In the case where threads share an unpartitioned cache as in Figure 11(a), thread 4 is the slowest progressing thread and hence is the critical path thread, which determines the application performance. In comparison, when the same sample application is executed with an equally-partitioned cache as illustrated in Figure 11(b), thread 3 and thread 4 benefit a little bit at the expense of thread 1. This could be because of the



(a) Shared (unpartitioned) cache. (b) Private (static, equal partition) cache. (c) Dynamic unequal partitions.

Figure 11. Execution progress of four threads at a particular execution time point within a parallel section.

reduced contention due to the partitioned cache. Thread 4 still remains the critical path thread. Thread 2 shows no sign of change and this could be because it does not suffer from an adverse impact of contention. Lastly, when the shared cache is partitioned based on a performance measure such as thread CPI as shown in Figure 11(c), thread 4, which was previously in the critical path, is speeded up and is no longer the slowest thread of the application. Thread 2 appears to show no change which may be because it is not very sensitive to cache size. After this CPI based partition, thread 2 becomes the critical path thread. As can be clearly seen, the new critical path thread is much faster than the critical path thread in both the shared unpartitioned and the equal partition cases. Therefore, we can expect an improvement in the overall performance of this parallel section. As stated earlier, an important limiting factor for this scheme could be if the critical path thread is not very cache sensitive, in which case, it does not benefit from additional cache ways and as a result, there may not be much performance benefit from dynamic cache partitioning. If, on the other hand, threads other than the critical path thread have a low cache sensitivity, then some cache resources can be taken away from these threads and can be given to the critical path thread. After discussing the mechanics of cache partitioning in Section V, in Section VI, we present two schemes that can be used to reach the desired situation depicted in Figure 11(c).

V. Required Hardware Support

In this section, we discuss the requisite underlying hardware enhancements required to implement our technique. There are at least two options in partitioning a shared cache. The first approach is to use reconfigurable caches where the cache hardware structures are modified at runtime [6], [14]. This approach may lead to considerable loss of data during the reconfiguration. Also, the cache remains unavailable during the reconfiguration process and hardware complexity increases. The second approach is to implicitly partition the cache by modifying the cache replacement algorithm used by the shared cache [28]. In this case, there is no sudden reconfiguration but a gradual move towards the intended partition. This approach also does away with problems of cache unavailability during reconfiguration and heavy hardware complexity. When a thread suffers a cache miss and the number of cache ways that belong to it is less than the thread's assigned cache partition ways, a cache line belonging to some other thread is chosen for replacement. If, on the other hand, the number of cache ways belonging to the thread is greater than or equal to the assigned number of ways, a cache line belonging to the same thread is chosen for replacement. This way, the cache is incrementally partitioned via the replacement policy. To implement this strategy, each set of the cache is assigned four counters to record the current assignment of cache ways among the four threads. There are four other counters which contain the current target assignment of ways for each thread. Upon a miss, if the current assignment counter for the thread is less than its target assignment, a cache way from some other thread is replaced. If the counter is not less than the target, one of its own is replaced. Note that the least recently used policy (LRU) is still used for replacement but in a sense it is now the thread-wise LRU. Essentially, cache partitions are maintained by controlling which thread can evict which cache line. Therefore, in this approach, although a thread/core can access a cache line present in another thread's cache partition, it cannot evict a cache line belonging to another thread's partition.

VI. Dynamic Cache Partitioning

Our dynamic cache partitioning scheme is applied at the granularity of *execution intervals* of around 15 million instructions. That is, at the end of each interval, we optimize for the next interval[2]. The dynamic cache partitioning scheme gathers execution counter values such as cache hits/misses, cycle counts and instruction counts for each thread during each execution interval. At the end of each such interval, the cache space is partitioned based on individual thread performances, so as to speed up the critical

[2]Note that an (execution) interval can contain multiple parallel sections, and similarly, a parallel section can span multiple execution intervals.

path thread, at the cost of other threads. We present two variants of our dynamic cache partitioning approach in this section. We found CPI value to be more helpful in reflecting a thread's performance than miss rate. This is because, miss rate does not account for the number of memory accesses made by the thread. The basic structure of the scheme remains the same in both the cases. We envision a dynamic runtime system which gathers thread wise cache metrics of a multithreaded program execution at each execution interval. Although we consider an execution interval of 15 million instructions in this paper, this is actually a tunable parameter. The schemes we present vary in the way the cache partitioning decision is made.

A. CPI Based Partitioning

```
Initialization:
  •Start out with equal partitions in the first interval.
    $\forall\ t,\ partition_i = \frac{TotalCacheWays}{NumberofCores}$
At the end of each interval:
  •Record CPI for each thread t, $CPI_i$.
  •Assign cache partitions to threads, based on their CPIs.
    Partition for thread t,
    $partition_t = \frac{CPI_t}{\sum CPI_i} \times TotalCacheWays$
```

Figure 12. CPI based dynamic cache partitioning scheme.

CPI based cache partitioning is a scheme which collects execution characteristics during each interval and partitions the cache space based on the thread cycles-per-instruction (CPI) values. The thread with a high CPI receives a larger portion of the cache and those with lower CPIs receive lesser portions accordingly. This is done with a hope that a thread with a high CPI (critical path thread) can improve its performance with a larger cache share, thereby, improving the overall application performance. Figure 12 outlines the CPI based dynamic cache partitioning algorithm. In this scheme, we start out with equal cache partitions during the first interval. At the end of each interval, the execution characteristics are collected and the CPI values for each of the threads are computed. The cache space is then partitioned based on these computed CPI values. The formulation used to decide the cache partitions is:

$$partition_t = \frac{CPI_t}{\sum CPI_i} \times TotalCacheWays$$

That is, the number of cache ways assigned to each thread is proportional to the CPI value of the thread.

B. Dynamic Model Based Partitioning

The main drawback of the previous scheme is its naivete in assuming a particular cache sensitivity metric. It lacks the knowledge of how the CPI value of the thread may change when a cache way is given to it or when a cache way is taken away from it. Therefore, a simple CPI based cache partitioning may not do very well, and in fact, may harm performance in certain cases. Therefore, we propose a *dynamic learning based* algorithm that considers not just the thread CPIs during the current interval but also the individual thread CPI curves so that the cache space can be partitioned more accurately. The core of this scheme is a *runtime thread performance modeling*, which is explained below.

In the first execution interval, we start out with equal partitions for all the threads. At the end of the first interval, the previously described CPI based cache partitioning is used to partition the cache for the second interval. We do the same for the second interval, in the process collecting two data points for the CPI models. Later, at the end of each interval, we build a runtime CPI model for each thread. Specifically, we model the dependency of CPI on the number of cache ways. These CPI models are built at runtime, for each of the threads using the available data points. By data points, we mean the assigned number of cache ways and the corresponding CPI figures observed under these cache ways. Using these available data points for each thread, we use a simple cubic spline interpolation [34] to fit a curve for each of the threads. The choice of the curve fitting algorithm used is independent of the partitioning scheme, and therefore, any other algorithm could also be used. Figure 15 shows the dynamic CPI models of a sample four threaded application execution. The CPI curves of the threads can be of any form. Now, the goal is to allocate ways so as to minimize the CPI of the highest CPI thread, which essentially minimizes the overall CPI of the application. This problem can be posed in the following form:

```
•For thread $t$, cache model can be of this form,
  $CPI_t = f_t^n(Ways_t)$
•Goal is to find each thread partition $Ways_t$,
  such that the overall CPI is minimized,
    •Find $Ways_t\ \forall t$
    •such that, $Minimize(CPI_{overall})$
      where, $CPI_{overall} = max(CPI_t)$
•Constraint for the optimization is, $Ways_{total} = \sum Ways_t$
```

Since the search space for this problem is very large, we employ a heuristic strategy with minimal runtime overhead in our scheme. Our curve fitting algorithm tries to find the best possible cache way partition that minimizes the CPI of the highest CPI thread. Minimizing the CPI of the highest CPI thread is synonymous with speeding up the slowest thread (critical path thread), thereby speeding up the entire application execution.

As described earlier, at the end of each execution interval, the execution characteristics such as instruction counts and cycle counts are recorded for each thread along with the current cache partition. Using the current data point (cache ways, CPI) and the previously recored data points, a CPI model is built for each thread of the application being executed. Then, our cache partitioning algorithm is invoked. At each iteration of our partitioning algorithm, we take away a cache way from the thread with the lowest CPI (fastest thread) and assign it to the thread with the highest CPI (slowest thread). After this repartitioning, the CPI values are recalculated for all the threads based on the thread

```
Initialization:

• Start out with equal partitions in the first interval.
    $\forall\ t,\ partition_i = \frac{TotalCacheWays}{NumberofCores}$

At the end of first two intervals:

• Use the previous CPI based cache partitioning.
    • Record CPI for each thread $t$, $CPI_i$.
    • Assign cache partitions to threads, based on their CPIs.
        Partition for thread t,
            $partition_t = \frac{CPI_t}{\sum CPI_i} \times TotalCacheWays$

At the end of each interval:

• Record the CPI value for each thread t, $CPI_i$.
• Build performance models for each of
the threads using cubic spline
    •Model CPI to cache way dependency
• Determine individual cache partitions by redistributing
cache ways based on the thread performance models
    •Step1: Reassign cache partitions
        $threadMaxCPI = HighestCPI$
        $threadMax$ = Thread with highest CPI
        $threadMinCPI = LowestCPI$
        $threadMin$ = Thread with lowest CPI
        $waysMaxCPI = waysMaxCPI + 1$
        $waysMinCPI = waysMinCPI - 1$
    •Step2: Recalculate the thread CPIs after
     reassignment (step 1) based on the individual
     thread performance models
        $newThreadMax$ = Thread with highest CPI
        IF($threadMaxCPI \neq tewThreadMaxCPI$)
            $waysMaxCPI = waysMaxCPI - 1$
            $waysMinCPI = waysMinCPI + 1$
            EXIT
        ELSE GOTO Step1
• Assign the newly calculated cache partitions to threads
```

Figure 13. Dynamic curve fitting based cache partitioning scheme.

CPI models built earlier. Notice that, by recalculating the CPI values this way, whether the repartitioning has actually helped or not is taken into account. It is important to note here that, determining CPI values using the CPI models yields predicted CPI values for each of the threads and not the actual CPI values. After recalculating the thread level CPI values this way, repartitioning step is performed again, and so on. Thread CPI recalculation based on the thread CPI models and the cache way repartitioning steps are iteratively repeated this way. The termination point is when some other thread becomes the highest CPI thread. When that happens, we revert back the assignment by one step and terminate. Finally, we apply the calculated cache way assignment for the threads. Figure 13 shows a detailed sketch of the workings of our curve fitting based cache partitioning heuristic, and Figure 14 summarizes the entire procedure in a pictorial form.

We now provide a sample illustration of this algorithm. Figure 15 shows the thread based CPI curves for each of the four threads of a sample execution. We can clearly see that, these CPI curves vary and the cache way sensitivity varies. The total number of cache ways in the entire cache is assumed to be 32. Now, the current partition of cache ways is assumed to be an equal one with each thread allocated 8 cache ways. Also, thread 2 is the critical path thread, with a CPI value of nearly 8. The best possible cache

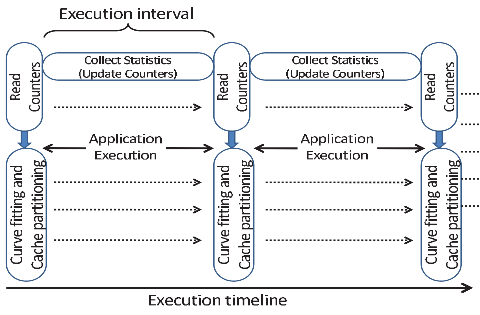

Execution interval
Read Counters
Collect Statistics (Update Counters)
Application Execution
Curve fitting and Cache partitioning
Execution timeline

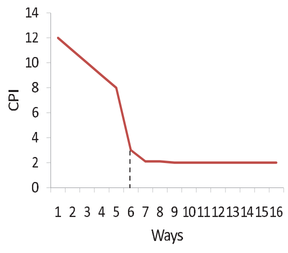
Figure 14. Dynamic cache partitioning scheme in action during application execution.

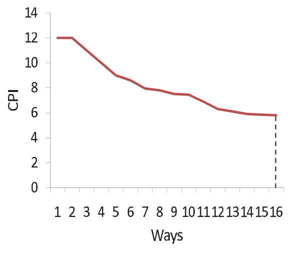
partition, which minimizes the predicted CPI of the critical path thread, and therefore the overall CPI, is indicated in Figure 15 by dotted lines. This new partition is obtained by our cache partitioning algorithm described in Figure 13, using the runtime cache models shown in Figure 15. As can be seen, thread 2, which is the critical path thread gets the biggest cache partition of 16 ways and the other three threads get 5, 5 and 6 ways. As a result, this runtime curve fitting based partition yields an overall predicted CPI of around 6 instead of nearly 8 in case of equal partitions. Note also that these models (Figure 15) are updated after each execution interval (15 million instructions), and consequently, the next time our scheme makes a better decision. In this way, dynamic variations in thread behavior are taken into account.

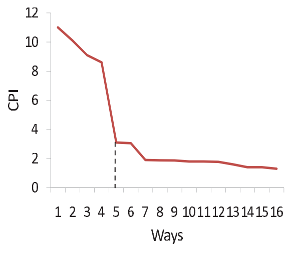
C. Implementation Details

In our implementation, we consider an operating system (OS) cache allocator which allocates a certain amount of cache to each of the applications in a workload [29]. The application now executes under the control of our proposed runtime system which implements the cache partitioning algorithm. Implementing the cache partitioning this way (i.e., within a runtime system) gives us more flexibility. This is because it is not always easy to extract thread-level information in commercially available OSs. We envision a hierarchical system (shown in Figure 16), where OS manages the cache-partitioning among applications and the runtime-system manages the cache-partitioning among the threads of an application. Note that, in this setting, our intra-application scheme can be applied to each application simultaneously. Details of the thread-level partitioner (runtime system) in Figure 16 are shown in Figure 17.

At each execution interval, our runtime system reads the execution characteristics (instruction and cycles) from performance monitors [31] and uses them to calculate the cache partitions. This runtime system comprises of three main components (see Figure 17). The cache/CPI monitor collects performance counter values from the hardware and feeds them to the Partition Engine, which calculates the cache partitions based on our algorithm. Finally, the Configuration Unit applies the calculated cache partitions to the hardware.


(a) Thread 1


(b) Thread 2


(c) Thread 3

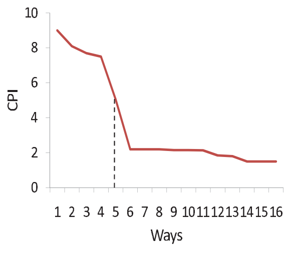

(d) Thread 4

Figure 15. Cache models for individual threads (in a four thread execution) and the best possible cache way partition for a 32-way set-associative cache. The goal of the partitioning step is to minimize the CPI for the slowest thread. The models shown in this figure are maintained for each thread of the application.

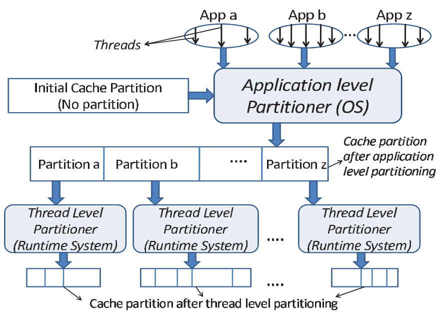

App a
App b
App z
Threads
Initial Cache Partition (No partition)
Application level Partitioner (OS)
Partition a
Partition b
....
Partition z
Cache partition after application level partitioning
Thread Level Partitioner (Runtime System)
Thread Level Partitioner (Runtime System)
Thread Level Partitioner (Runtime System)
Cache partition after thread level partitioning

Figure 16. Hierarchical cache partitioning system.

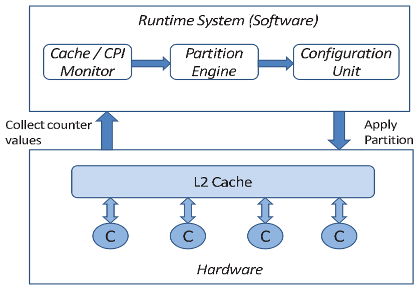


Figure 17. Runtime system based cache partitioning system.

VII. Experimental Evaluation

We use the experimental setup described in Section III-C to evaluate our dynamic partitioning scheme. As mentioned before, we use Simics [21], which is a full system simulator to implement our scheme. We implement our cache partitioning scheme as a module in Simics. The cache partitioning scheme acts as a dynamic runtime system which implements the cache partitioning as described in Section VI-C. The runtime system decides the individual thread cache partitions and then issues the cache partition commands to Simics which actually partitions the shared cache. Runtime system collects the execution characteristics of the threads by reading the performance counters. The results presented below include all the runtime overheads incurred by our implementation. Also, for our scheme to work, we would need to pin threads to cores. We bind individual threads to the cores on the CMP using the Solaris system calls. We also made experiments where threads were not pinned, which showed similar results. When we explored further, we realized that this is because the OS (Solaris) exploits thread-to-core locality, i.e., threads are not migrated frequently. In rare cases where migration occurred, our predictions were not optimal (during that period), but our approach quickly adapted to the new thread-mapping. Therefore, we believe our approach is quite resistant to thread migrations.

We now provide a brief snapshot of our dynamic cache partitioning scheme in action and then, make three kinds of comparisons. Note that, we evaluate only the curve fitting based dynamic cache partitioning scheme described in Section VI-B, since it outperforms the simple CPI based scheme in all of the cases we tested. We used execution intervals of 15 million instructions for our dynamic cache partitioning scheme. We performed experiments with other execution intervals as well, and there appeared to be little variation across the results when the execution interval was either increased or decreased. We evaluated our approach with 9 representative applications from the SPEC OMP [2] and NAS [1] benchmark suites. After warming up the caches, we ran each of these benchmark applications for 5 billion instructions to collect the relevant statistics. An important concern when evaluating the effectiveness of any kind of dynamic runtime system based scheme is the performance overheads incurred by the dynamic scheme. Since we consider execution intervals higher than that of most operating system scheduling quanta, overheads are incurred very infrequently, as infrequently as once in 15 to 20 million instructions or higher. Therefore, overheads turned out to be very small (less than 1.5%) when weighed against the overall execution time and are already included in the results presented below.

A. Dynamic Cache Partitioning Snapshot

We now provide a brief snapshot of our dynamic cache partitioning scheme in action. Figure 18 shows the working of our dynamic cache partitioning scheme across a small set of execution intervals of NAS CG application [1]. During the first execution interval, the cache partition is equal. During the next intervals, cache is partitioned based on their run time cache models in order to speed up the critical path thread. In this example, since thread 3 is the slowest thread

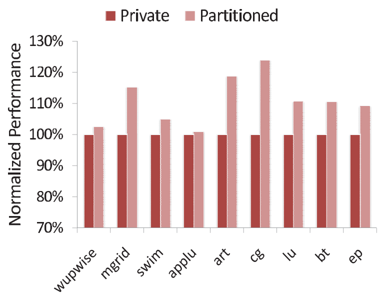


Figure 19. Performance improvement of our dynamically partitioned cache over the equally partitioned cache (private cache).

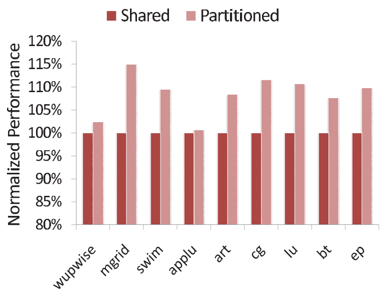


Figure 20. Performance improvement of our dynamically partitioned cache over the shared unpartitioned cache.

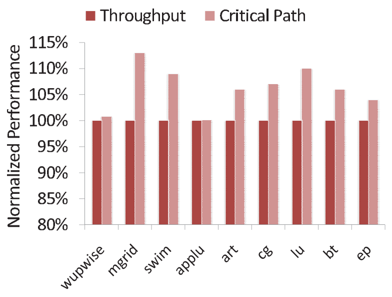


Figure 21. Performance improvement of our dynamically partitioned cache over a throughput-oriented cache partitioning scheme.

	Cache Way Partitioning				Overall
	Thread 1	Thread 2	Thread 3	Thread 4	CPI
Interval 1	16	16	16	16	7.72
Interval 2	10	8	32	6	6.35
Interval 3	10	6	34	6	6.21
Interval 4	10	6	34	6	6.22

Figure 18. A snapshot of our dynamic cache partitioning scheme in action across four consecutive execution intervals of the NAS CG application. This figure shows the cache ways allocated to each of the threads during the execution intervals and the resulting CPI values.

(the CPI values for threads 1, 2, 3, 4 were 3.06, 2.96, 6.35, 2.95, after the first interval), it is given the largest cache partition and consequently, as can be seen, the overall CPI of the application is reduced, thereby, improving the overall performance. A similar situation results as we move from interval 2 to interval 3. That is, our approach successfully modulates the cache space allocation dynamically during execution.

B. *Comparison with Alternate Schemes*

We start by comparing our dynamic cache partitioning scheme with a statically partitioned cache with equal partitions, which is the same as a private L2 cache. A private cache also yields the optimal fairness results. Therefore, comparison with private cache is the same as the comparison with fairness oriented schemes (such as those presented in [10] [18]). Figure 19 shows the performance improvement achieved by our dynamic cache partitioning scheme over a private, equally partitioned cache. Dynamically partitioned cache results in performance improvement of up to 23% over the private cache case. This is because our algorithm adapts dynamically to build performance models and allocates available cache space specifically to speed up the critical path thread, as opposed to a simple static partition. On average, our scheme outperforms the private cache configuration by about 11%.

Next, we compare our proposed dynamic cache partitioning scheme to an unpartitioned shared cache (i.e., a fully-shared cache). Figure 20 shows the performance improvement achieved by our dynamic cache partitioning scheme over a shared, unpartitioned cache. As can be observed from this plot, dynamic cache partitioning scheme outperforms the shared cache by up to 15%. The average performance improvement achieved is about 9%. Although shared cache is considered the most efficient in terms of utilization and data sharing, the dynamic cache partitioning scheme outperforms the shared cache. In three of the benchmarks we tested, dynamic partitioning scheme yields only a small benefit. This is because of the very small working set size of those benchmarks.

We also compare our scheme to a throughput-oriented scheme (similar in objective to prior schemes mentioned in Section IV-B). Here, we use the throughput oriented strategy employed by these prior schemes in the intra-application case for comparison with our scheme. When four threads belonging to the same application are executing on a four core CMP, a throughput-oriented scheme tries to improve the overall throughput of the four core CMP. Therefore, such a scheme does not care about which thread is being speeded up in the process. As we can see from the plot in Figure 21, our proposed cache partitioning scheme outperforms the throughput-oriented scheme for all the applications we tested.

C. *Sensitivity Study*

To study the sensitivity of our dynamic cache partitioning scheme to the number of cores in the CMP, we ran the same nine applications but with 8 threads each on a 8 core CMP with a cache size of 1MB, which is larger than the working size set. Figure 22 shows the performance improvement of our dynamic partitioning scheme over both the private and shared caches. As we can clearly see, our scheme shows performance gains similar to those observed in the 4 core case.

VIII. Conclusions

In this paper, we have shown the threads that belong to a given (multithreaded) application can exhibit considerably different on-chip cache behaviors and consequently, considerably different execution characteristics. We have

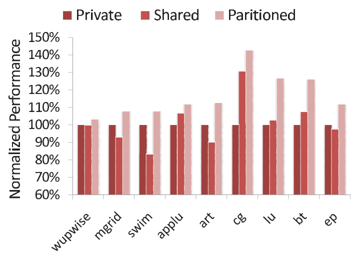


Figure 22. Performance comparison on an 8 core CMP.

also made an argument that many of the prior shared cache management and cache partitioning schemes proposed in the context of emerging chip multiprocessors (CMPs), target throughput, fairness and QoS as primary objectives. While these techniques are generally very effective, they are agnostic to whether these threads belong to the same application or to different applications. When these threads belong to the same application, the parameter to optimize should be the critical path execution time rather than overall throughput. We proposed a dynamic cache partitioning scheme which tries to optimize the critical path execution time at each execution interval. We have also further shown that such a scheme can enhance the application performance significantly. Finally, we show that our scheme outperforms shared unpartitioned cache and statically (equal) partitioned cache by up to 15% and 23%,respectively, and also brings over 20% improvement over a prior throughput oriented scheme.

Acknowledgment

This research is supported in part by NSF grants CNS 0720645, CCF 0811687, CCF 0702519, CNS 0202007, CNS 0509251, and a grant from Microsoft Corporation. The authors also acknowledge the support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRCs Focus Center Research Program.

References

[1] http://www.nas.nasa.gov/resources/software/npb.html.
[2] http://www.spec.org/omp.
[3] http://openmp.org.
[4] Cell broadband engine - white paper. IBM, 2006.
[5] http://www.intel.com/p/en_US/products/server/processor/xeon7000?iid=servproc+body_xeon7400subtitle
[6] S. Adve. Reconfigurable caches and their application to media processing. In *Proc. ISCA*, 2000.
[7] D. H. Albonesi et al. Selective cache ways: on-demand cache resource allocation. In *Proc. MICRO*, 1999.
[8] B. M. Beckmann et al. Asr: Adaptive selective replication for cmp caches. In *Proc. MICRO*, 2006.
[9] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proc. ISCA*, 2006.
[10] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. ICS*, 2007.
[11] T. Sherwood et al. Discovering and exploiting program phases. In *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, Dec. 2003.
[12] S. Chen et al. Scheduling threads for constructive cache sharing on cmps. In *Proc. SPAA*, 2007.
[13] K. Flautner et al. Drowsy caches: simple techniques for reducing leakage power. In *Proc. ISCA*, 2002.
[14] A. Gordon-Ross et al. Fast configurable-cache tuning with a unified second-level cache. In *Proc. ISLPED*, 2005.
[15] R. Hetherington. In *The UltraSparc T1 processor*. SUN, 2005.
[16] L. R. Hsu et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *Proc. PACT*, 2006.
[17] A. Jaleel et al. Adaptive insertion policies for managing shared caches. In *Proc. PACT*, 2008.
[18] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. PACT*, 2004.
[19] J. Lin et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proc. HPCA*, 2008.
[20] Y. Chen et al. Efficient Shared Cache Management through Sharing-Aware Replacement and Streaming-Aware Insertion Policy. In *Proc. IPDPS*, 2009.
[21] P. S. Magnusson and et al. Simics : A full system simulation platform. In *Computer, 35(2):50-58*, 2002.
[22] M. K. Qureshi et al. Adaptive insertion policies for high performance caching. In *Proc. ISCA*, 2007.
[23] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *Proc. PACT*, 2006.
[24] R. Ramanathan. Intel multi-core processors : Making the move to quad-core and beyond. In *Intel White paper, Intel Corporation*.
[25] L. Soares et al. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *Proc. MICRO*, 2008.
[26] S. Srikantaiah et al. Adaptive set pinning: managing shared caches in chip multiprocessors. *SIGARCH Comput. Archit. News*, 2008.
[27] L. Chakraborty et al. A Utility-based Approach to Cost-Aware Caching in Heterogeneous Storage Systems.. In *Proc. IPDPS*, 2007.
[28] G. E. Suh et al. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. HPCA*, 2002.
[29] G. E. Suh et al. Dynamic cache partitioning for simultaneous multithreading systems. In *Proc. PDCS*, 2001.
[30] G. E. Suh et al. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.
[31] P. F. Sweeney et al. Using hardware performance monitors to understand the behavior of java applications. In *Proc. VM*, 2004.
[32] D. Tam et al. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proc. EuroSys*, 2007.
[33] G. H. Timothy G. Mattson. In *An overview of the Intel TFLOPS Supercomputer*. Intel.
[34] D. F. Watson. Contouring: A guide to the analysis and display of spatial data. 1994.
[35] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proc. ISCA*, 2005.
[36] X. Zhang et al. Towards practical page coloring-based multicore cache management. In *Proc. EuroSys*, 2009.
[37] C. Liu et al. Exploiting Barriers to Optimize Power Consumption of CMPs. In *Proc. IPDPS*, 2005.