A Helper Thread Based Dynamic Cache Partitioning Scheme for Multithreaded Applications[*]

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Taylan Yemliha
Syracuse University
tyemliha@syr.edu

Emre Kultursay
Pennsylvania State University
euk139@cse.psu.edu

ABSTRACT

Focusing on the problem of how to partition the cache space given to a multithreaded application across its threads, we show that different threads of a multithreaded application can have different cache space requirements, propose a fully automated, dynamic, intra-application cache partitioning scheme targeting emerging multicores with multilayer cache hierarchies, present a comprehensive experimental analysis of the proposed scheme, and show average improvements of 17.1% and 18.6% in SPECOMP and PARSEC suites.

Categories and Subject Descriptors: B.3.2 [**Hardware**]: Design Styles - Cache Memories
General Terms: Algorithms, Performance
Keywords: Cache, multi-core, partitioning, helper thread

1. INTRODUCTION

One of the important problems faced by system designers and software writers for multicores is to manage the large variety of resources provided by these machines such as processor cores, on-chip memory space, on-chip network, off-chip memory bandwidth shared across multiple, concurrently executing applications and threads, with potentially different service requirements. Recent work on multicore resource management and partitioning focused on a variety of resources like processor cores [6], shared caches [5, 4, 8, 11, 12, 17, 18, 20], and off-chip bandwidth [3, 10, 13]. A common characteristic of all these studies is that they address resource management and partitioning problem at an application granularity. To be more specific, in the context of shared on-chip caches, prior work considered how to partition the shared on-chip cache space that exist in many commercial multicore architectures among concurrently-executing applications. This problem can be termed as *inter-application cache partitioning.* In contrast, past work mostly ignored *intra-application cache partitioning*, the problem of how to partition the cache space given to a multithreaded application across its threads. The only work on intra-thread cache partitioning is by Muralidhara et al [16]. However, their work can only handle a single layer of shared cache and applying a single layer cache partitioning algorithm to multiple layers in an independent (isolated) fashion does not generate the best results. On the other hand, the approach that is proposed in this paper is designed for multilayer cache partitioning and applying it can enable better results. Focusing on the intra-application multi-layer cache partitioning problem, we make the following contributions:

• We illustrate that different threads of a multithreaded application can have different cache space requirements. As a result, cache space available to the application needs to be distributed nonuniformly across its threads.

• We propose a fully automated, dynamic intra-application cache partitioning scheme targeting emerging multicores with multilayer cache hierarchies. A unique characteristic of our scheme is that, it can partition caches at different layers of a hierarchy at the same time.

• We present a comprehensive experimental analysis of the proposed cache space partitioning scheme.

While this intra-application cache management problem can be attacked in principle using a static (compiler based) strategy, in practice such a strategy may not be very effective due to (i) difficulty a compiler may experience in analyzing source codes of multithreaded applications and (ii) dynamic changes in application behavior (and consequently in resource needs of different threads) during the course of execution. Motivated by these observations, we instead propose a *dynamic scheme*, which is implemented using a *helper thread.* More specifically, our scheme attaches a helper thread to each application, whose main responsibility is to partition the cache space allocated to the application by the operating system (OS) (or a hardware based partitioner) across its threads in the best possible way to maximize overall performance of the application. To achieve this goal, the helper thread maintains a *dynamic performance model* of each thread of the application and, considering the cache space allocated to the application by the OS, determines the ideal partitioning of this space among the threads, with the goal of maximizing application performance. The collected performance numbers from our experiments indicate that, as compared to the best static partitioning scheme, our proposed approach brings an average improvement of 17.1% in the SPECOMP [1] applications and 18.6% in the PARSEC [2] applications.

2. MOTIVATION

In this section, we present experimental data for motivating the *dynamic, intra-application, multi-layer* cache partitioning problem attacked in this paper. For these experiments, we use two multithreaded applications, *facesim* (from PARSEC) and *art* (from SPECOMP), and execute them separately on the multicore architecture shown in Figure 1(d). This architecture has eight cores and three layers of on-chip cache hierarchy. For each application, we performed experiments with six schemes, and recorded average memory access times (AMAT). In all these schemes, we use eight threads per application and each thread is mapped to a separate core. The first of these schemes is *No-Partition*, a scheme that does not partition any cache among the cores that access it. As a result, data accesses from different threads can interleave in random patterns and displace each other's data from the shared cache(s). The second scheme, called *Uniform*, partitions each cache evenly among all cores (threads) that access it. The third scheme, *Nonuniform*, represents an exhaustive search which selects the ideal partition for each cache in the system from a large set of candidate partitions. In the next two schemes, *Nonuniform-L2* and *Nonuniform-L3*, we partition either L2 or L3 caches and the other cache layer is accessed without any restriction (as in *No-Partition*). One can see from the results in Figure 2 that *Nonuniform* performs better than *Uniform*, which indicates that uniform partitioning of cache

*This work has been supported in part by NSF grants #1017882, #0963839, CNS #0720645, CCF #0811687, CCF #0702519, and a grant from Microsoft Corporation.

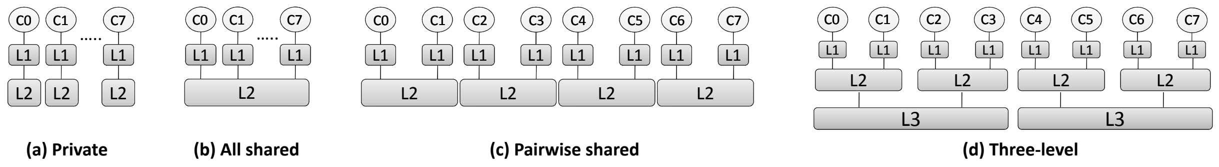


Figure 1: A multicore architecture with multi-layer cache hierarchy. Each C_i refers to a processor core.

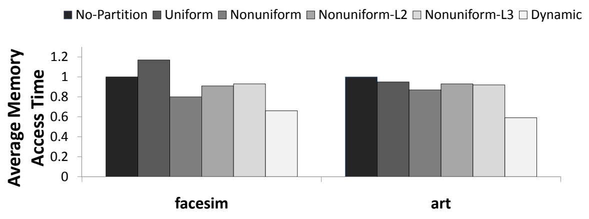


Figure 2: Normalized average memory access times for two multithreaded applications. For both applications, all bars are normalized with respect to the first bar.

space among the threads of an application may not be the best option. *Nonuniform* also performs better than *No-Partition* as it eliminates the destructive interferences (conflict misses) across threads. Further, *Nonunifom* generates better results than both *Nonuniform-L2* and *Nonuniform-L3*, meaning than it may not be sufficient to partition only a single layer in the cache hierarchy.

Nonunifom is a static scheme in which cache space is partitioned across threads nonuniformly and this partitioning is maintained throughout the execution of the application. Considering that an application can exhibit different data access patterns and require different amounts of cache in different phases of its execution, we also performed experiments with a dynamic scheme. In this scheme, marked as *Dynamic* in Figure 2, application execution is divided into fixed epochs and, for each epoch, the partition that performs best is used (identified through exhaustive search). Consequently, as we move from one epoch to another, cache partitioning can change. We observe that *Dynamic* outperforms *Nonuniform*, indicating that dynamically adjusting the partitions based on the needs of different threads can be very effective.

While the results reported in Figure 2 are encouraging, there is an important problem that needs to be addressed. *Dynamic* is not a practical scheme as it employs exhaustive search, which cannot be afforded in a real implementation. Therefore, to mimic its behavior in a realistic implementation, we need a *dynamic model* that can capture the runtime behavior and resource needs of the threads as the application executes.

3. MULTICORE ARCHITECTURES AND PROBLEM DEFINITION

Figure 1 shows four different eight-core multicore architectures, each with a different on-chip cache hierarchy. Keeping the number of cores, the number of layers and the cumulative capacities of caches at each level fixed, we can represent an on-chip cache hierarchy using a set of parameters $\{s_{0,1}, s_{1,2}, s_{2,3}, \cdots, s_{n-1,n}\}$, where n is the number of cache layers in the architecture, $s_{0,1}$ is the number of cores connected to an L1 cache, and $s_{j,j+1}$ (where $1 \leq j < n$) is the number of caches at layer j that are connected to a cache at layer $j+1$. For example, four multicore architectures in Figures 1(a), (b), (c) and (d) can be expressed using $\{1,1\}$, $\{1,8\}$, $\{1,2\}$, and $\{1,2,2\}$. Note that this characterization based on $s_{j,j+1}$ parameters is quite general and can capture many commercially available multicore architectures from different manufacturers. For example, Figures 1(c) and 1(d) correspond to Intel Harpertown and Intel Dunnington machines, respectively. Note also that, we consider only symmetric cache hierarchies, which means, each cache at a particular level is connected to an equal number of lower level caches.

Our proposed cache partitioning scheme takes this characterization (along with the number of cores and cache capacities) as input and partitions all shared caches among all cores (threads) that access them. For example, when targeting the 8-core architecture represented by $\{1,2,2\}$ (see Figure 1(d)), our scheme partitions an L2 cache between the threads that execute on the two cores connected to that L2 cache. Similarly, the first L3 cache in the system will be partitioned among the first four cores and the second L3 cache will be partitioned among the second group of four cores. Throughout our discussion, we assume that a given thread is mapped to and executed on a single core.

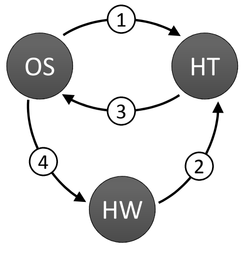
The main differences between inter-application and intra-application cache partitioning are that, their objectives and the way they are implemented are different. The intra-application cache partitioning tries to minimize the latency of the slowest thread, whereas inter-application cache partitioning tries to optimize an inter-application (workload wide) metric such as workload throughput or weighted speedups [19]. This difference in objectives also require adoption of different strategies in implementing them. While inter-application partitioning is typically an OS problem, for intra-application partitioning, different implementation strategies (e.g., a runtime system or a dynamic compiler based one) are possible.

In this paper, our main goal in partitioning the shared caches in a multicore system across threads of a multithreaded application is to reduce the latency of its slowest thread. Since the workload-wide performance metrics used by the current state-of-the-art inter-application partitioning schemes does not have the same goal, such existing schemes cannot be used for partitioning the cache space allocated to an application across its threads. In addition, current partitioning schemes consider partitioning at a single layer (usually the last layer in the on-chip hierarchy). Our proposed scheme, in contrast, *partitions all shared caches simultaneously* in the same optimization step. As we demonstrate later in the paper, the savings coming from our multi-layer partitioning *cannot* be obtained from independent partitioning of each layer of caches. That is, applying any known cache partitioning scheme to each layer in isolation may not generate very good results. In this work, we use way partitioning [18, 11, 5, 17, 20] to partition a given cache space across multiple threads.

4. DYNAMIC PARTITIONING SYSTEM

4.1 Interactions among the Players

There are three main players in our proposed partitioning system: (1) operating system (OS), (2) helper threads (HT), and (3) hardware (HW). The role of each player and the flow of information between them are depicted in Figure 3. Each application is attached a "helper thread" whose main responsibility is to partition the cache space allocated to the application to maximize its performance. In this system, the OS



(1) Cache space (the number of ways) allocated to the application by the OS.
(2) Per-thread statistics from hardware performance counters.
(3) Request for redistribution of cache ways among the threads of the application.
(4) Enforcing way-partitioning.

Figure 3: System components and their interactions. The main players are the operating system (OS), helper thread (HT), and hardware (HW).

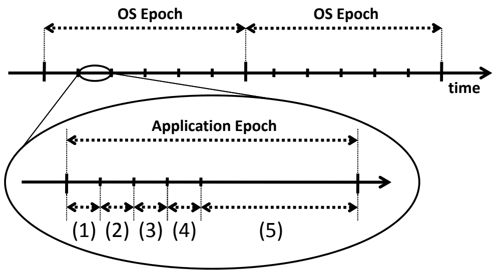


Figure 4: Each OS epoch (the inter-application partitioning interval) is composed of many application epochs which are further divided into five intervals: (1) performance monitoring, (2) performance modeling, (3) resource partitioning, (4) system interfacing, and (5) application execution using the new partitioning.

loads an application with a helper thread while informing the helper thread on the resources allocated to the application (step 1). Through the course of execution of the application, the helper thread *periodically* collects statistics regarding the cache performances of the individual threads using hardware performance counters (step 2). Then, it decides on a new partitioning of the shared cache components in the system and informs the OS with its decision (step 3). The OS, in turn, instructs the hardware to take necessary actions through an ISA extension (step 4). The big picture here is that, while the OS manages cache partitioning across applications, our proposed helper thread, which is the focus of this paper, is responsible for partitioning the cache space allocated to an application across its threads. Note that our helper thread can be a standalone runtime component as well as an OS-provided component attached to each application at the time the application is initiated in our multicore system.

4.2 Details of the Helper Thread

The proposed helper thread is responsible for the following tasks: (i) Performance Monitoring: Reading the performance counters and calculating the performance of the threads under the current configuration (current cache partitioning), (ii) Performance Modeling: Constructing per-thread performance models using the current and past performance statistics. Note that this is a *dynamic model* being built during the course of execution, (iii) Resource Partitioning: Partitioning the space allocated to the application by the OS in each of the shared caches in the target architecture using the constructed performance models, and (iv) System Interfacing: Communicating with the OS, receiving the total resources allocated to its application and sending the partitioning/allocation requirements of the individual threads in the application.

When an application starts execution, the helper thread attached to it also starts. This helper thread communicates with the OS at the end of two intervals, namely, the *OS epoch* and the *application epoch*. The OS epoch defines the granularity of OS-initiated inter-application cache partitioning. At the beginning of every OS epoch, the OS can partition the cache space among simultaneously-executing applications and informs their helper threads[1]. Then, until the next OS epoch, at every application epoch, the helper thread of each application distributes the application's resources (cache ways) among its threads. Figure 4 shows that within an OS epoch, there are many application epochs and, at every application epoch, the helper thread performs four main tasks, namely, performance monitoring, performance modeling, resource partitioning, and system interfacing, which are explained next.

4.2.1 *Performance Monitoring*

While the application threads are executing, the helper thread monitors these threads and measures their cache performance. If a single cache level partitioning is targeted, in order to quantify a thread's cache performance, the miss rate of the thread on that cache level could be used. However, when there are multiple levels that can be partitioned (e.g., see Figure 1(d)), the effect of partitioning decisions at multiple levels must be considered collectively. This is because the miss rates of the caches are not uncorrelated, and more importantly, misses at different layers incur different costs. Therefore, cache misses or miss rates cannot be used as our performance measure.

In this work, we use the *average memory access time* (AMAT) [9] as a measure of the cache performance of a thread. The AMAT of a thread depends on the sizes of the cache partitions allocated to threads, and it also takes into account different costs of accessing different levels of cache hierarchy. Therefore, AMAT can accurately represent the effects of varying cache allocations to threads. In order to calculate AMAT, at every epoch, the helper thread accesses hardware performance counters to obtain the following statistics for each thread: (i) total cycles spent on memory instructions and (ii) total number of memory instructions. The ratio of these two values gives the AMAT of a thread in that epoch. As explained below, this AMAT value, along with the current allocations of this thread in L2 and L3 caches, are used to update that thread's dynamic performance model. Note that, while we use AMAT values to balance latencies of different threads of an application, in our experimental evaluation, we also present execution time results.

4.2.2 *Performance Modeling*

In order to perform dynamic cache partitioning, we need to predict the impact of increasing and decreasing the cache space available to a thread on its performance (data access latency). To achieve this, we propose a *dynamic performance model*, which is parameterized using cache space allocations at different layers in the on-chip cache hierarchy. Specifically, our runtime model for a given thread (that belongs to a multithreaded application) can be expressed using a *three-dimensional plot* where the x-axis and y-axis denote, respectively, the cache space allocations from the L2 and L3 layers (note that a core has access to only one cache from each layer). We build a separate model (three-dimensional plot) for each thread of the application. In the model of thread-i, point $d(s_{L2}, s_{L3})$ indicates the observed AMAT value for this thread when allocated s_{L2} ways from L2 and s_{L3} ways from L3. When this thread executes with allocation (s_{L2}, s_{L3}) for one epoch, we record the observed d value in our three-dimensional plot. As a result, as a thread is allocated different (s_{L2}, s_{L3}) values during the course of execution, we can build a *dynamic model* for that thread.

[1]Any inter-application cache partitioning strategy can be used for this purpose.

The most important use of this dynamic model in our framework is to *predict* the performance of a thread if allocated certain number of L2 and L3 ways. That is, using the observed (s_{L2}, s_{L3}) values, we can fit a *surface* that represent the d values under various cache (way) allocations and use this surface to predict the performance of the thread if allocated (s_{L2}, s_{L3}), i.e., s_{L2} L2 ways and s_{L3} L3 ways. This surface is dynamically updated with the newly-observed (s_{L2}, s_{L3}) values and the corresponding d values to adapt to the dynamic modulations in thread behavior. Consequently, if a thread is allocated the same (s_{L2}, s_{L3}) ways at two different points during its execution, we update the surface with the most recently-observed d value (for that allocation pair). In this way, our approach continuously adapts to the phase changes the application experiences during execution. It is to be noted that, while this model captures the behavior of a single thread under different cache (way) allocations it experiences during execution, one needs a higher level approach to decide the cache space allocations across *all* the threads of the multithreaded application. This approach needs to consider the dynamic models built for all threads and make globally-acceptable (application-wide) cache space (way) allocations. The next section gives the details of such an approach.

4.2.3 Cache Space Partitioning

This step solves an optimization problem, whose goal is to maximize application performance by carefully distributing cache ways (allocated to the application) across its threads. In order to formulate the optimization problem, we first make the following definitions. Considering the ith L2 cache in the target architecture, $q_{L2,i}$ denotes the total cache ways allocated to this application by the OS. These cache ways are shared by $m_{L2,i}$ threads (numbered from 0 to $m_{L2,i} - 1$), and the number of ways allocated (from this cache) to the kth thread is denoted as $s_{L2,i}(k)$. The definitions for the L3 cache are analogous. Using these definitions, the objective function and constraints of our optimization problem when targeting a multicore system with $N(L2)$ and $N(L3)$ number of L2 and L3 caches can be stated as follows:

$$minimize[max_i\{AMAT(thread_i)\}], \text{under}$$
$$\forall\ i \in [0, N(L2)-1] \sum_{k\in[0,m_{L2,i}-1]} s_{L2,i}(k) \leq q_{L2,i}$$
$$\forall\ i \in [0, N(L3)-1] \sum_{k\in[0,m_{L3,i}-1]} s_{L3,i}(k) \leq q_{L3,i}.$$

The goal is to minimize the average memory access time of the thread with the slowest memory performance. The two conditions above ensure that the sum of the cache ways allocated to the threads (that belong to the same application) at a cache level cannot be more than the ways allocated to the application in that cache component.

There are various possible methods to solve this optimization problem. However, since this problem needs to be solved at runtime and at every application epoch, costly techniques such as exhaustive search or integer linear programming are not desirable. Instead, we employ a heuristic technique that identifies the current slowest and fastest threads of the application (in terms of average memory access times in the current epoch) and transfers the resources (cache ways) from the fastest thread to the slowest one in an attempt to balance their AMAT values. Since this transfer is performed at every epoch of the application, this approach approximates the ideal resource reallocation closely. The pseudo-code in Figure 5 shows our algorithm for the helper thread. *Threads* represents the set of threads in the associated application and, for the t^{th} thread, $P[t]$ denotes cache resources (number of ways in the L2 and L3 caches) allocated to it. The performance statistics collected in *data* are used to construct a performance model given by $surf$. Then, the algorithm first finds the slowest (t_{max} with the maximum $AMAT$ value) and fastest (t_{min} with the minimum $AMAT$ value) threads in the system. If the variance in performance is less than a threshold, the helper thread decides to continue with the existing partition. Otherwise, it searches for a better partition than the current one by redistributing the resources allocated to these two threads. The new partition obtained after the redistribution differs from the current partition in only two entries which correspond to t_{min} and t_{max}. Note that, an ideal solution to the given optimization problem would redistribute the resources of all threads in the system. On the other hand, since we are solving this problem at runtime, our approach approximates the ideal solution by redistributing only the resources of the slowest and the fastest threads in the system at every application epoch.

```
HelperThread() :
  while ( true )
    for each $t \in Threads$
      $c \leftarrow memcycles(t)$
      $n \leftarrow meminstrs(t)$
      $AMAT[t] \leftarrow c/n$
      $data[t].insert(P[t], AMAT[i])$
      $surf[t] \leftarrow$ SurfaceFit($data[t]$)
    end for
    $(t_{min}, AMAT_{min}) \leftarrow$ Min($AMAT[0\ to\ n_{threads} - 1]$)
    $(t_{max}, AMAT_{max}) \leftarrow$ Max($AMAT[0\ to\ n_{threads} - 1]$)
    $L_{avg} \leftarrow$ FindAvgAMAT()
    $variance \leftarrow (AMAT_{max} - AMAT_{min})/AMAT_{avg}$
    if $variance > threshold$
      $(R[t_{min}], R[t_{max}]) \leftarrow$ Redistribute($t_{max}, t_{max}, surf$)
      $P[t_{min}] \leftarrow R[t_{min}]$
      $P[t_{max}] \leftarrow R[t_{max}]$
      Partition($P$) // system call
    end if
  end while
```

Figure 5: The pseudocode for the helper thread.

Field	Number of Bits	Description
PRT	32	op-code
COID	8	core ID
CLVL	1	cache level
CAID	2	cache ID
W	64	way allocation bit-vector

Table 1: Fields of the cache partitioning instruction.

4.2.4 System Interfacing

When a new partitioning is performed, the helper thread passes the new allocations for each thread to the OS. Note, however, that the OS is free in honoring these new allocations or simply dropping them, depending on the current status of the system and co-runner applications. The new partition information is delivered to the OS using a system call with the following signature:

$$void\ partition(uint\ L2_{T1}, uint\ L3_{T1}, uint\ L2_{T2}, uint\ L3_{T2}, ...).$$

Note that, the OS already knows which application process made this system call and how many threads there are within that process. When the OS receives this system call, it starts allocating the L2/L3 cache ways to the threads of the application while verifying that the sum of resources allocated the application threads does not exceed the total resources allocated to the application. If the application does not use some of the ways it is allocated, the OS records this information and can use it at its next inter-application partitioning decision, which is performed at the end of the current OS-epoch.

Although the way-partitioning decision is performed in software, it is actually administered in hardware. The OS informs the hardware about the cache ways allocated to each thread by using a new instruction added to the ISA (recall that we have one thread per core). The format of this privileged in-

Parameter	Value
Number of Cores	8
ROB/Core	128 entry
Bandwidth/Core	4-fetch, 4-issue, 4-commit
Branch Pred./Core	hybrid 8192-entry shared / 2048-entry bimod / 8192-entry meta table
L1 Cache	8×(16KB; 4 ways; 32B line; 2 cycles)
L2 Cache	4×(512KB; 16 ways; 128B line; 10 cycles)
L3 Cache	1×(6MB; 64 ways; 256B line; 20 cycles)
Off-Chip Latency	200 cycles

Table 2: Default multicore configuration.

struction is as follows:

```
PRT COID, CLVL, CAID, W.
```

The fields of the instruction and their bit-widths (for the system described in Table 2 that has 4 L2 caches, 2 L3 caches, and a maximum of 16 ways of L2 and 64 ways of L3 that can be allocated to a single thread) are listed in Table 1. Note that, as the hardware is unaware of the thread IDs, the instruction uses core IDs and the OS performs the conversion between thread IDs and core IDs during the system call. The core ID (*COID*) field is used to indicate the core that the partitioning decision will be performed for. The cache level (*CLVL*) and cache ID (*CAID*) fields indicate the target cache level (we are concentrating on L2 and L3 caches) and the sequence number of the target cache component[2]. Each bit in the 64-bit wide way allocation (W) field indicates whether the cache way that corresponds to that bit position is allocated to that core or not. For example, the following two instructions allocate the last 7 ways of the first L2 cache ($CAID = 1$) and the last 10 ways of the zeroth L3 cache ($CAID = 0$) to a core with $coreID = 2$:

```
PRT 2, L2, 1, 0x007F
PRT 2, L3, 0, 0x000003FF 00000000
```

In the first instruction, the least significant 7 bits (bits 0 to 6) of the W field are set, which allocates ways 9 through 16 of cache $L2_1$ to the indicated core. Similarly, in the second instruction, the bits 32 to 41 are set, which implies that ways 0 to 9 of the $L3_0$ cache must be allocated to the same core.

5. EXPERIMENTS

5.1 Setup

The default configuration simulated in this study models the 8 core multicore system depicted in Figure 1(d). The major experimental parameters and their default values are given in Table 2. We used SIMICS full-system simulator [14] and GEMS [15] to model this multicore architecture. We performed full system simulation, and the implemented helper thread is triggered by the OS (Solaris in our case) along with the corresponding application. We used 120 million instructions as our application epoch size (a value determined after some initial experiments). To fit surface in our 3D plots (performance models), we use *statistical regression* [7].

In this paper, we use all the applications in the SPECOMP [1] and PARSEC [2] application suites. For each application in our experimental suite, we tested the following schemes: (i) *No-Partition.* This scheme corresponds to unrestricted sharing, as a result, applications that access a cache component can displace each other's data. (ii) *Uniform.* Under this scheme, each cache in the multicore system is divided (way-wise) as evenly as possible among all threads that share it. (iii) *Static Best.* In this scheme, for each cache, we select a static partitioning that generates the best result (when the entire on-chip cache hierarchy considered) through exhaustive search of all possible static partitions. Consequently, this scheme represents the best partitioning that can be achieved by any static scheme. It needs to be noted that, this scheme is not implementable in practice. (iv) *Dynamic.* This is our proposed intra-application cache partitioning scheme described in detail in Section 4. (v) *Dynamic-L2.* In this scheme, we partition only the L2 caches, and the L3 space is shared by all cores that access it. (vi) *Dynamic-L3.* In this scheme, we partition only the L3 caches, and each L2 cache is shared by all cores that access it. (vii) *L2+L3.* In this scheme, we partition L2 and L3 in an *uncoordinated fashion* using a separate performance model for each one of them. More specifically, under this scheme, performance models for L2 layer and L3 layer observe the data access latency in respective layers and adapt partitioning based on these dynamically updated performance models independently. (viii) *Ideal.* This scheme represents an *optimal strategy* for partitioning the shared on-chip cache space. Under this scheme, application execution is divided into epochs and, for each epoch, we run *Static Best* to determine the best partition of cache components for that epoch. Clearly, this scheme is not implementable in practice due to its extremely high computational cost.

[2]The cache level (CLVL) and the sequence number at that cache level (CAID) are necessary and sufficient parameters to uniquely identify a cache component in a hierarchy.

5.2 Results

In all experiments, we execute the application using 8 threads, and there is a one-to-one mapping between threads and cores. Since we use full-system simulation, all the overheads incurred by the tested schemes are included in the results presented in this section. Figure 6 presents average memory access times (AMAT) for our applications under the schemes described above. For each application, all bars are *normalized* with respect to the data access latency value achieved by the *No-Partition* scheme. In all applications, our proposed scheme (*Dynamic*) generates better results than all other schemes except *Ideal.* More specifically, average improvements achieved by *Uniform*, *Static Best*, *Dynamic*, *Dynamic-L2*, *Dynamic-L3*, *L2+L3*, and *Ideal* over *No-Partition* are 1.5%, 11.6%, 26.8%, 14.0%, 13.1%, 17.6%, and 30.6%, respectively, for the SPECOMP benchmarks. The corresponding savings for the PARSEC benchmarks are -3.6% (performance degradation), 7.5%, 24.7%, 8.9%, 7.0%, 10.8%, and 28.9% in the same order. When we compare the results generated by *Dynamic-L2* and *Dynamic-L3* with those obtained through *Dynamic*, we see that, for the maximum improvements in performance, both L2 and L3 layers should be partitioned. Further, *L2+L3* performs worse than our scheme. The main reason for this is the lack of coordination (under this scheme) between the L2 and L3 partitionings. More specifically, each performance model tries to adapt based on the observed performance without considering the potential partitioning that may be adopted by the other performance model, and this in turn results in suboptimal behavior. Finally, the difference between our scheme and *Ideal* are 5.4% and 5.6%, on average, for the SPECOMP and PARSEC benchmarks, respectively. This shows that our proposed intra-application cache partitioning strategy comes very close to the optimal partitioning, under the same epoch length.

Figure 7 shows the *normalized execution cycle results* for the different schemes. The trends observed in this plot are similar to those in Figure 6. The average execution time improvement our scheme brings over *L2+L3* (the next-best performing scheme excluding *Ideal*) are 7.2% in the case of SPECOMP and 10.6% in the case of PARSEC. To give a visual perspective on the distribution of cache ways by our scheme among different threads, we present in Figure 8 the distribution of cache ways in one of our L2 caches and the L3 cache in the target multicore architecture over the course of execution of application *apsi.* It can be easily observed from these plots that our scheme dynamically distributes the cache ways among involved threads.

Figure 9 gives a snapshot from execution (again, of *apsi*)

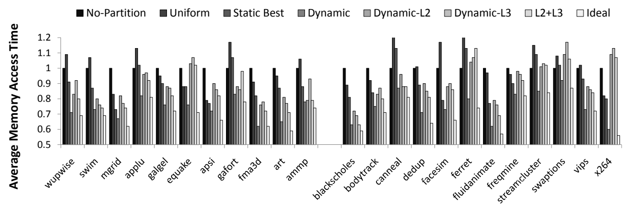


Figure 6: Average memory access times for different applications. For each application, each bar is normalized with respect to the first bar (*No-Partition*).

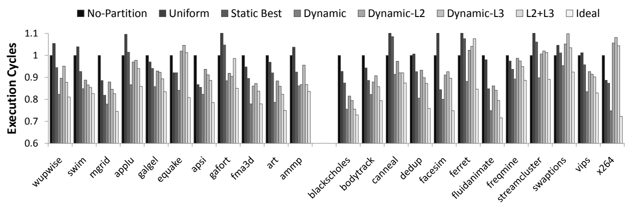


Figure 7: Execution cycles for different applications. For each application, each bar is normalized with respect to the first bar (*No-Partition*).

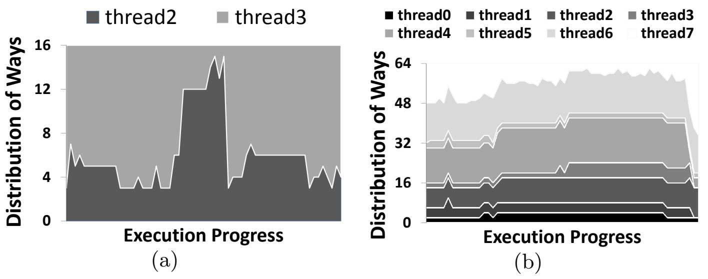


Figure 8: Distribution of the cache ways in threads of *apsi* under our scheme: (a) partitioning of one of the L2 caches and (b) L3 cache partitioning.

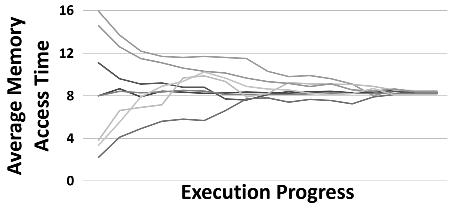


Figure 9: A snapshot of *apsi*. Average memory access times of threads converge as the execution progresses when our partitioning scheme is used.

that illustrates how our scheme improves performance by balancing the data access latencies of different threads. The x-axis corresponds to execution time, and the y-axis captures the average access latency. The important observation from this graph is that, as the execution progresses, the data access latencies experienced by different threads of this application get closer to one another, and eventually, they all end up in the range of 8.12 cycles and 8.47 cycles, illustrating how our scheme balances data access latencies of the different threads. Although not presented here due to space constraints, we observed similar patterns with our remaining applications as well.

6. CONCLUSIONS

We proposed an intra-application cache partitioning strategy that divides the shared cache space available to a multithreaded application across its threads. Using a dynamic performance model, this technique is able to partition shared caches in multiple layers of the on-chip cache hierarchy across threads. We implemented this scheme and tested it using multithreaded benchmarks. The collected performance numbers indicate that, as compared to the best static partitioning scheme, our proposed approach brings an average improvement of 17.1% in the SPECOMP applications and 18.6% in the PARSEC applications.

7. REFERENCES

[1] V. Aslot et al. SPECOMP: A new benchmark suite for measuring parallel computer performance. In *International Workshop on OpenMP Applications and Tools*, 2001.

[2] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[3] D. Burger et al. Memory bandwidth limitations of future microprocessors. In *ISCA*, 1996.

[4] D. Chandra et al. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, 2005.

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, 2007.

[6] Y. Ding et al. A helper thread based edp reduction scheme for adapting application execution in cmps. In *IPDPS*, 2008.

[7] P. Duchesne and B. Remillard. *Statistical Modeling and Analysis for Complex Data Problems*. Springer, 2005.

[8] F. Guo et al. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2007.

[10] D. Kaseridis et al. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. In *HPCA*, 2010.

[11] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[12] B. Ko et al. Scalable service differentiation in a shared storage cache. In *ICDCS*, 2003.

[13] F. Liu et al. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *HPCA*, 2010.

[14] P. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2), Feb 2002.

[15] M. M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.

[16] S. P. Muralidhara et al. Intra-application shared cache partitioning for multithreaded applications. In *PPoPP*, 2010.

[17] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[18] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *PACT*, 2006.

[19] A. Snavely and D. M. Tullsen. Symbiotic job scheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[20] G. E. Suh et al. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1), 2004.