International Journal of Parallel Programming (2019) 47:1014–1044
https://doi.org/10.1007/s10766-019-00637-y

Adaptive Thread Scheduling in Chip Multiprocessors

Ismail Akturk[1] · **Ozcan Ozturk**[2]

Received: 27 March 2015 / Accepted: 7 May 2019 / Published online: 14 May 2019

Abstract
The full potential of chip multiprocessors remains unexploited due to architecture oblivious thread schedulers employed in operating systems. We introduce an adaptive cache-hierarchy-aware scheduler that tries to schedule threads in a way that inter-thread contention is minimized. A novel multi-metric scoring scheme is used which specifies L1 cache access characteristics of threads. Scheduling decisions are made based on these multi-metric scores of threads.

Keywords Adaptive scheduling · Chip multiprocessors · Inter-thread contention · Multi-metric scoring

1 Introduction

The number of transistors on a die no longer increases according to Moore's Law [1] due to power constraints and diminishing returns. However, the demand for increased performance and higher throughput is still in place. To provide higher throughput and increased performance without bumping into physical limits of Moore's Law, novel multiprocessor architectures have emerged, including chip multiprocessors that contain multiple cores on a single chip [2]. Another way to provide higher throughput and increased performance is to run more than one thread on each core with multithreading, namely simultaneous multithreading [3]. The choice of threads to be scheduled on the same core has significant impact on overall system performance. Inter-thread contention occurs since coscheduled threads are competing for shared resources. The primary shared resource that influence the ultimate performance is cache. An efficient

✉ Ismail Akturk
akturki@missouri.edu

Ozcan Ozturk
ozturk@cs.bilkent.edu.tr

1 Department of Electrical Engineering and Computer Science, University of Missouri, Columbia, MO, USA

2 Department of Computer Engineering, Bilkent University, Ankara, Turkey

scheduling should minimize the contention on caches to maximize utilization and system performance. Since the execution characteristics of threads varies over time, scheduling decisions should be adapted based on provisioned behaviors of threads for the near future.

In this work, we particularly focus on thread scheduling and introduce an adaptive cache-hierarchy-aware scheduling algorithm for chip multiprocessors. The proposed algorithm uses hardware counters that allow us to identify cache access pattern of each thread. It implements an intelligent scheduling decision mechanism that tries to schedule threads in a way that inter-thread contention is minimized. The originality of this work is the use of *multi-metric scoring* scheme that is calculated based on L1 cache access characteristics of a thread. Then, multi-metric scores of threads are used in scheduling decisions. While previous studies are focused on the performance of last-level cache (LLC) to optimize scheduling decisions, our evaluations indicate that eventual performance of LLC is dependent on how the upper levels of cache hierarchy are utilized. Thus, adaptive cache-hierarchy-aware scheduling effectively increases the utilization of upper levels of cache, and thereby improves the throughput and maximizes the system performance.

The organization of the paper is as follows. In Sect. 2, we provide background on thread scheduling. We discuss multi-metric scoring scheme in Sect. 3 and introduce our proposed adaptive cache-hierarchy-aware scheduling algorithm in Sect. 4. We provide experimental results in Sect. 5. We discuss the related work on thread scheduling in Sect. 6, including cache replacement, partitioning algorithms, and coscheduling methods. Finally, we conclude and provide future work in Sect. 7.

2 Background

Typical workloads running on chip multiprocessors are composed of multiple threads. These threads may exhibit different execution characteristics. In other words, they may run in different phases (e.g., memory phase, compute phase). Besides different threads, even execution characteristics of a particular thread may change over time. When threads are scheduled together that are running in phases that exacerbates contention for shared resources, the system performance decreases and throughput reduces due to conflicts. On the other hand, when threads running in cooperative phases are scheduled together, the contention for shared resources is diminished that yields to better resource utilization, higher throughput, and improved system performance.

The choice of threads to be scheduled on the same core has significant impact on overall system performance. Inter-thread contention occurs since coscheduled threads are competing for shared resources. The primary shared resource that influence the performance is the cache. An efficient scheduling should minimize the contention for shared caches to maximize utilization and system performance. Since the execution characteristics of threads varies over time, the scheduling decision has to be remade based on provisioned behaviors of threads for the near future.

Other shared resources include functional units, instruction queues, memory, interconnections between resources, the translation look-aside buffer (TLB), renaming registers, and branch prediction tables. While threads share these resources to improve

utilization, they also compete for these resources that may reduce efficiency. Our focus in this work is on scheduling of threads based on interactions on shared caches.

From the operating system point of view, scheduling decisions have to be made based on the measures that affect the performance the most. Thus, we make a detailed survey on possible measures and evaluate their effects on performance. We observe that, contrary to the common thought, L1 cache access pattern of threads has a great impact on performance. To elaborate, we focused on L1 cache access patterns of threads and formulate a score for each thread that reflects execution characteristics of threads. The score of a thread specifies the intensity to compete for shared resources, or namely the *friendliness* of the thread. A thread that uses decent shared cache tends to be friendly, namely it causes less degradation to its co-runners, and it suffers less from its co-runners. Although the notion of friendliness is widely used in recent studies; we observed that they consider just a particular metric to determine friendliness, such as IPC of each thread or miss ratio. Such metrics are well indicators for particular cases; however, they become insufficient for general cases where a larger diversity is expected. Due to lack of adequate measure of friendliness, we developed a multi-metric scoring scheme to specify the execution characteristics of threads and make scheduling decisions on this multi-metric score.

2.1 Problem Statement

The conflicts among threads are difficult to predict due to their unrepeatable nature [4]. The behavior of a thread changes over time. For example, a thread may have high memory demands during the initialization and data loading, and following that it may have high CPU demand while processing loaded data. Loading and processing may occur several times that eventually changes behavior of a thread over time. In such cases, static scheduling schemes are likely to fail on minimizing conflicts among threads.

An intuitive scheduling would be to group memory intensive threads with threads that are non-memory intensive. However, it is not always possible to find such pairs (e.g., all threads may be memory intensive in a particular time). Also, threads may be memory intensive; however, their memory access pattern may change drastically that affects the overall performance. For example, streaming threads may generate more memory requests; however, they do not get any benefit from cache hierarchy, since they have limited (or no) locality. Also, streaming behavior of such threads are detrimental to other threads which are memory intensive. They evict the cache lines of other threads without gaining any benefit in return.

Figures 1 and 2 show the variances in L2 miss ratio and instruction per cycle (i.e., IPC) of threads under different scheduling schemes, respectively. Note that, there are four benchmarks running on two cores where two threads share a private L1 cache and all threads share a unified L2 cache as LLC.

Existing schedulers used in operating systems are unaware of multi-level cache hierarchies and access/sharing pattern of threads running on chip multiprocessors. For this reason, traditional schedulers are oblivious to the access patterns of threads and they may schedule threads in a way that their memory accesses contradict with each other. This, in turn, hurts the cache performance leading to high miss ratios, high

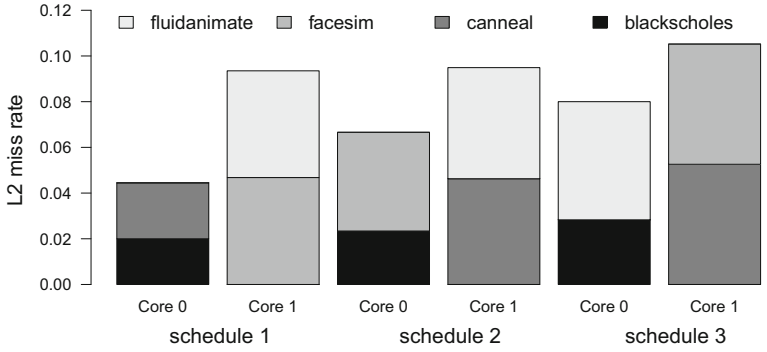

Fig. 1 The L2 miss variation of four threads running on two cores under different scheduling schemes

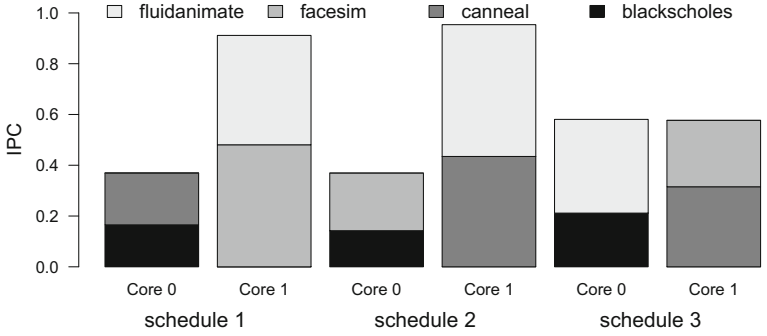

Fig. 2 The IPC variation of four threads running on two cores under different scheduling schemes

number of evictions and longer time to serve memory requests since data has to be brought from lower levels of memory hierarchy.

In addition, the use of profiling provided by compiler-directed approaches can not exploit the full potential of chip multiprocessors, since such profiling may not reflect the dynamically changing inputs and execution characteristics of threads. Similarly, schedulers used in simultaneous multithreaded processors try to schedule threads based on each thread's expected resource utilization to maximize, but do not consider variances and changes in thread execution characteristics over time.

Scheduling of threads has to be made based on the measures that affect the performance the most. As a primary shared resource, caches and their performance measures are critical. There is a large body of work that characterize the last-level cache (LLC) performance. The limitation of these studies are that they are oblivious to performance of higher level caches. Notice that the number of LLC accesses is a function of misses occurred in higher level caches. For this reason, we claim that to minimize LLC accesses and probability of LLC misses, we have to minimize the cache misses in higher levels of cache. In order to do this, the thread scheduling decisions have to be made by considering higher levels of cache performance.

Based on these observations, we conclude that, cache-hierarchy-aware scheduling for chip multiprocessors, which adopts dynamically changing execution characteristics of threads, is inevitable.

2.2 Motivation

Numerous research efforts have been made on minimizing cache conflicts and capacity misses of shared LLC (in most cases L2) for both multiprocessors and chip multiprocessors. Although such efforts are effective (i.e., minimizing cache conflicts and capacity misses of LLC), they ignore the effects of higher levels of cache hierarchy on eventual LLC performance. Typically, each core has L1 that is shared by multiple threads in chip multiprocessors. Being oblivious to L1 cache conflicts and misses eventually creates more pressure on lower level caches (e.g., L2) and results in high latency lower level cache accesses.

The fundamental motivation behind focusing on LLC in previous research efforts is that a miss on LLC requires high latency main memory access. Although this is a valid argument, it does not justify to underestimate the effect of L1 (or any cache level above LLC) on memory access latency. Contrary, we claim that L1 cache access pattern (i.e., number of accesses, misses, evictions, etc.) has great impact on overall memory access performance. This is our main motivation to implement a cache-hierarchy-aware scheduler.

In recent study, Zhang et al. [5] pointed out that there is very limited sharing of the same cache block among different threads. Modern applications are highly parallelized, where each thread is working on independent cache block. For this reason, it is very unlikely that they will access the same data block, so there is limited or no data sharing. For example, threads of data-parallel programs may process different sections of data. Similarly, threads of pipeline programs may execute different tasks that may not use the same data set. In both cases, there is no concern of shared data among multiple threads; however, the way the threads use shared cache has an influence on performance. This observation is important, since programs show different characteristics in different phases of the execution so that no particular mapping work well for all the phases. With this motivation, we propose *adaptive* cache-hierarchy-aware scheduler. More specifically, adaptive cache-aware-hierarchy scheduling aims to adopt changing execution characteristics of threads and tries to find best scheduling that improves the performance by reducing the cache contention and conflicts among coscheduled threads.

2.3 Contributions

In this work, we conduct a detailed study to show the importance of cache-hierarchy-aware scheduling for applications running on chip multiprocessors. We investigate the impact of scheduling threads with different execution characteristics and observe that the best scheduling for a given thread varies depending on other threads that are scheduled along with it.

We introduce a fine-grained, multi-metric scoring scheme to classify threads with respect to their execution characteristics. We use this fine-grained, multi-metric scoring scheme to predict threads that get along with each other and schedule them on the same core. The metrics used in scoring scheme are gathered from L1 cache, as opposed to LLC as in most of the previous works.

We propose a novel cache-hierarchy-aware scheduler that schedules threads in a way that minimizes the number of accesses to the lower level of cache/memory hierarchy and reduces the number of evictions required on shared caches that eventually limits the interference. Such a strategy leads to higher system throughput and improved performance.

The proposed cache-hierarchy-aware scheduler is adaptive, such that it takes dynamically changing execution characteristics of threads into account. We observe that by employing our adaptive cache-hierarchy-aware scheduling, the performance (i.e., instruction per cycle) of the benchmarks used in this work are improved by up to 12.6% with an average of 7.3% over the static schedules. The improvements are due to reduced interference among coscheduled threads, leading to reduced number of evictions/misses and balanced number of accesses that minimizes capacity conflicts.

2.4 Inter-thread Contention and Slowdown

Threads running on the same core compete for L1 cache, while they compete with all other threads running on chip multiprocessor for shared L2 cache. To assess the interference among threads and make a good scheduling evaluation, it is necessary to formulate the slowdown of a thread when running along with other threads.

To avoid effects of other parameters, such as difference between program execution times and context switches in operating system, we consider the following simplified scenario. We assume that there are N threads with the same number of instructions to be executed. The average slowdown of all threads can be calculated as geometric mean of slowdowns of threads. The scheduler that minimizes the average slowdown as given in Expression 1 is more desirable.

$$min \sqrt[N]{\prod_{i}^{N} \frac{IPC(i)_{stand\_alone}}{IPC(i)_{coscheduled}}} \tag{1}$$

There is a trade-off between minimizing the average slowdown and maximizing the overall system performance (i.e., IPC). It is possible to have schedules that have lower average slowdown, while resulting in lower performance. On the other hand, it is possible to have schedules that provide higher performance, while having higher average slowdown. Therefore, a good scheduler should find a balance between slowdown and performance.

2.5 Performance Counters and Monitoring

The chip multiprocessors have performance monitoring units (PMUs) with integrated hardware performance counters. The statistics that are needed to implement proposed scheduling algorithm can be collected through PMUs. PMUs can provide fine-grained statistics with relatively low overhead [6]. Parekh et al. [7] used hardware performance counters that provide cache miss and related information to schedule threads wisely in simultaneous multithreaded processors. Similarly, Bulpin and Pratt [8] used

performance counters to develop symbiotic coscheduling approach on simultaneous multithreaded processors.

For our proposed cache-hierarchy-aware scheduler, we focus on L1 cache access pattern of threads and classify them based on their propensity to compete for L1 cache and their relative effectiveness of L1 cache usage. Although classification of threads is widely adopted in scheduling research, we observed that they consider a particular metric to classify threads, such as IPC of each thread and miss rate. Such metrics are well indicators for certain cases; however, they do not work well for others. Thus, there is no silver-bullet metric that provides the best for all cases. With this in mind, we developed a multi-metric scoring scheme to specify execution characteristics of threads which is used to make scheduling decisions.

2.6 Phase Detection and Prediction

The prediction of thread's cache access behavior for the next interval is essential to obtain desired performance. Simply, predicting thread's cache access behavior for the next interval will be the same as the previous interval provides reasonable accuracy (e.g., between 84 and 95% [9]). Although the accuracy of the prediction can be increased by using more complex prediction methods, we believe that using last interval behavior as a prediction model for the next interval is sufficient for our purpose. It is a fair trade-off to have decent prediction accuracy with less complexity, compared to marginal gain in accuracy with high complexity.

3 Multi-metric Scoring Scheme

The behavior of a thread can be generalized by expressing three attributes for a given interval. These attributes are aggressiveness, density, and inefficacy. These attributes are represented in a binary vector, called *Attribute Vector* (AV). The illustration of AV is given in Fig. 3.

Each attribute corresponds to different characteristics of a thread. These characteristics have impact on the overall performance, eventually. The description of attributes are as follows.

Aggressiveness determines the degree of acrimony of a thread, specifying how much a thread interfere with other threads running concurrently on the same core. Aggressiveness of a thread is related to its propensity of evicting cache blocks of other threads. A thread that has higher eviction rate is considered as aggressive, while the one with lower eviction rate is considered as complaisant.

Density determines the relative intensity of cache accesses of a thread with respect to the sum of cache accesses of all threads. If a thread has higher number of cache accesses, then it is considered as dense. On the other hand, a thread is considered as sparse if a it has lower number of cache accesses relative to the number of overall cache accesses made during the given interval.

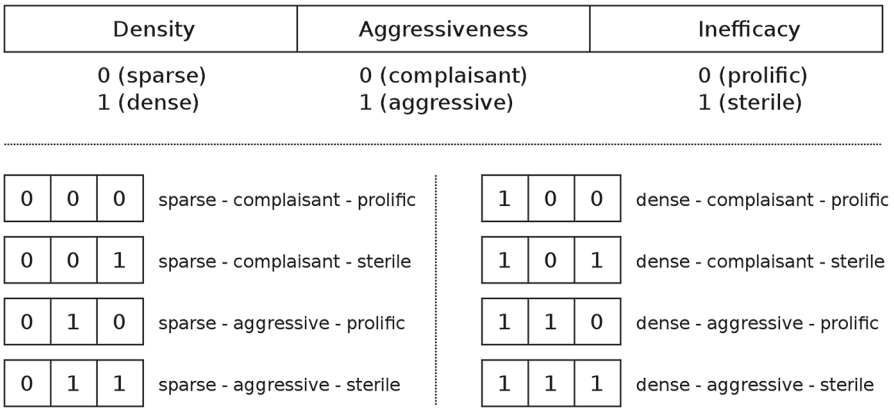

Fig. 3 Attribute vector (AV) expresses execution characteristics of a thread

Inefficacy determines the degree of efforts of a thread that goes unrewarded. If a thread has high cache miss ratio, then it is considered as sterile. On the other hand, it is considered as prolific if a thread has high cache hit ratio.

Although the attributes are related, they are considered as orthogonal to each other. Note that, a thread may be sterile, but not aggressive if its misses do not cause evictions.

These attributes are represented as bits in the attribute vector. The following formulas are used to determine whether a thread has certain attribute or not.

$$Aggressiveness(T_i) = \begin{cases} 1 \;\; if \;\; \frac{\#\ of\ L1\ evictions(T_i)}{\#\ of\ L1\ accesses(T_i)} \geq \tau_a, \\ 0 \; else \end{cases}$$

$$Density(T_i) = \begin{cases} 1 \;\; if \;\; \frac{\#\ of\ L1\ accesses(T_i)}{\sum_{j=0}^{N} L1\ accesses(T_j)} \geq \tau_d, \\ 0 \; else \end{cases}$$

$$Inefficacy(T_i) = \begin{cases} 1 \;\; if \;\; \frac{\#\ of\ L1\ misses(T_i)}{\#\ of\ L1\ accesses(T_i)} \geq \tau_i, \\ 0 \; else \end{cases}$$

where τ_a, τ_d and τ_i are thresholds for aggressiveness, density and inefficacy, respectively. They are determined empirically. Note that, N is the number of threads running on the chip multiprocessor.

Each attribute vector corresponds to a decimal value that specifies a multi-metric score of a thread. This value is calculated as:

$$Score = \sum_{i=0}^{2} 2^i \times AV_i \tag{2}$$

where AV_i represents ith bit of *attribute vector* of a thread (AV_i represents the least significant bit when i = 0, and AV_i represents the most significant bit when i = 2).

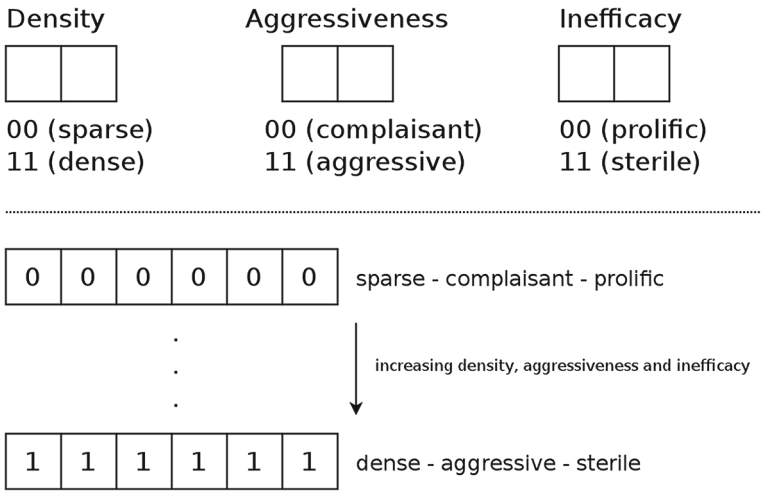

Fig. 4 Attribute vector that expresses execution characteristics of a thread in higher resolution

3.1 Scalability of Mutli-metric Scoring Scheme

In case of having large number of threads running on chip multiprocessors with extensive number of cores, the 3-bit attribute vector and scoring scheme may not differentiate execution characteristics of threads in a desired resolution. This may yield to have coarse-grained schedules. To have higher resolution of execution characteristics of threads with fine-grained schedules, it is better to expand the attribute vector. For each attribute, more bits can be used to specify the attribute in higher resolution. An example of attribute vector with a higher resolution is shown in Fig. 4.

4 Adaptive Cache-Hierarchy-Aware Thread Scheduling

After collecting information regarding L1 cache performance and updating attribute vectors of threads, the scheduling decision can be made. The scheduling decision is made based on the multi-metric scores of threads.

Each candidate schedule has a score expressed as coscheduling score, in short *CoScore*. The aim of scheduling is to find a schedule that minimizes CoScore calculated as:

$$CoScore(T_i, T_j) = AV(T_i) \quad \wedge \quad AV(T_j) \tag{3}$$

where T_i and T_j are candidate threads to be coscheduled (i.e., $T_i \neq T_j$); and $AV(T_i)$ and $AV(T_j)$ are attribute vectors of T_i and T_j, respectively. Note that T_i and T_j are candidate threads, so they are not scheduled yet.

The *CoScore* is simply a logical bitwise AND operation between multi-metric scores of candidate threads. It is simple, yet an effective way to find desired schedules that improve the performance. This, simple AND operation favors scheduling threads

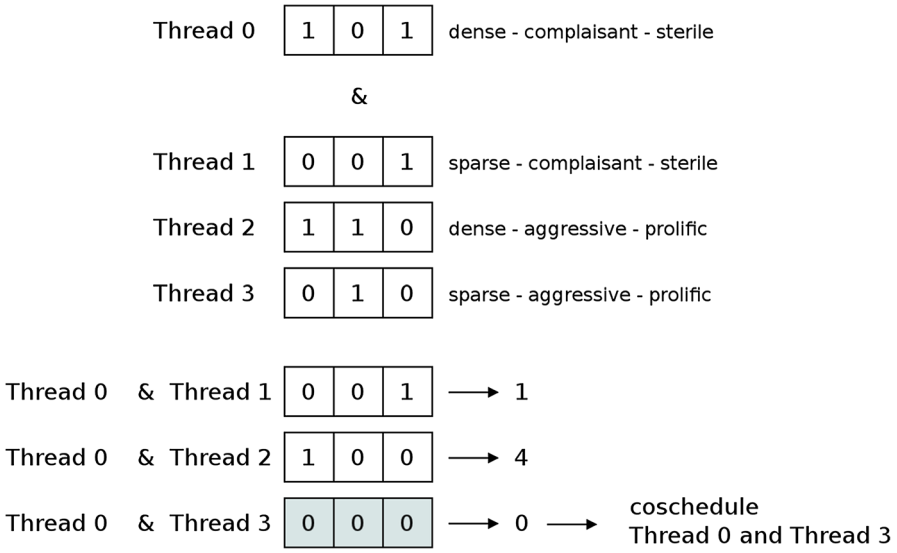

Fig. 5 Coscheduling score (CoScore) calculation

which can get along with each other. Figure 5 shows the illustration of calculating CoScore and finding the schedule that minimizes CoScore.

CoScore's most significant bits are dominant in selecting the schedule. Our approach tends to prefer a CoScore 011 over 100. Hence, metrics can be prioritized according to their positions in the CoScore.

A schedule that has the lowest CoScore is selected as a candidate. If there are more than one candidate, then the one that preserves locality is selected (i.e., no thread migration will be required).

The calculation of CoScore starts with a thread that has the lowest multi-metric score. Then, a thread that will minimize the coscheduling score when scheduled with the current thread is found. If there are multiple threads with lowest multi-metric score, a preference is given to the thread with higher IPC. If there are multiple threads with the same multi-metric score and the same IPC, then a thread is chosen randomly.

When all threads are scheduled to appropriate cores, the performance counters are reset. With the new scheduling period, attribute vectors of threads are reconstructed, multi-metric scores of threads are reevaluated and scheduling is re-executed as discussed. The details of this adaptive cache-hierarchy-aware thread scheduling algorithm is given in Algorithm 1.

When a thread is scheduled to execute on a different core than it was running on before, the cache blocks required by this thread have to be reloaded from L2 or lower level of the cache hierarchy. While this comes with an overhead, we determined that it is amortized over long execution intervals. This is due to the fact that the number of cache misses will be reduced as a result of reduced interventions in the scheduled thread.

Algorithm 1: Cache-hierarchy-aware thread scheduling algorithm.

```
UnSched_T → unscheduled threads;
UnMapped_T → matched but not mapped threads;
S, C → set of threads to be scheduled;
T_s, T_c → threads to be scheduled;
P_s, P_c → cores on which T_s and T_c run during the last interval, respectively;

while UnSched_T ≠ empty do
    T_s ← a thread that has the lowest score;
    if S has multiple threads then                          /* with the lowest score */
        T_s ← select thread ∈ S that has the highest IPC;
        if S has multiple threads then                      /* with the highest IPC */
            T_s ← select a thread ∈ S randomly;
        end
    end

    // Find the best candidate to be coscheduled with T_s
    C ← a thread ∈ UnSched_T that minimizes CoScore;
    if C has multiple threads then                          /* with the lowest CoScore */
        T_c ← select thread ∈ C that run on P_s recently;
    else
        T_c ← select a thread ∈ C with the highest IPC;
        if C has multiple threads then                      /* with the highest IPC */
            T_c ← select a thread ∈ C randomly;
        end
    end

    // Map matched threads to the core
    if P_s is available then
        map T_s and T_c to P_s;
    end
    else if P_c is available then
        map T_s and T_c to P_c;
    end
    else
        UnMapped_T ← T_s and T_c
    end
end

while UnMapped_T ≠ empty do
    T_s and T_c ← select matched threads from UnMapped_T;
    map T_s and T_c to the available(free) cores;
end
```

Figure 6 illustrates that how cache-hierarchy-unaware scheduling can penalize the threads that could perform much better. Notice the lower L1 hit ratio and higher L2 miss ratio. The number of L2 hits is four, while the number of L2 misses is 17.

Figure 7 illustrates that how cache-hierarchy-aware scheduling actually reduces the number of L2 misses and increases the L1 hit ratio. The same example is used with Fig. 6. The pressure due to the L1 misses is reduced that results in less number of L2 accesses and L2 misses. While the number of L1 hits is increased from 0 to 4, the number of L2 misses reduced from 17 to 14. Since this is just an illustration, we do not

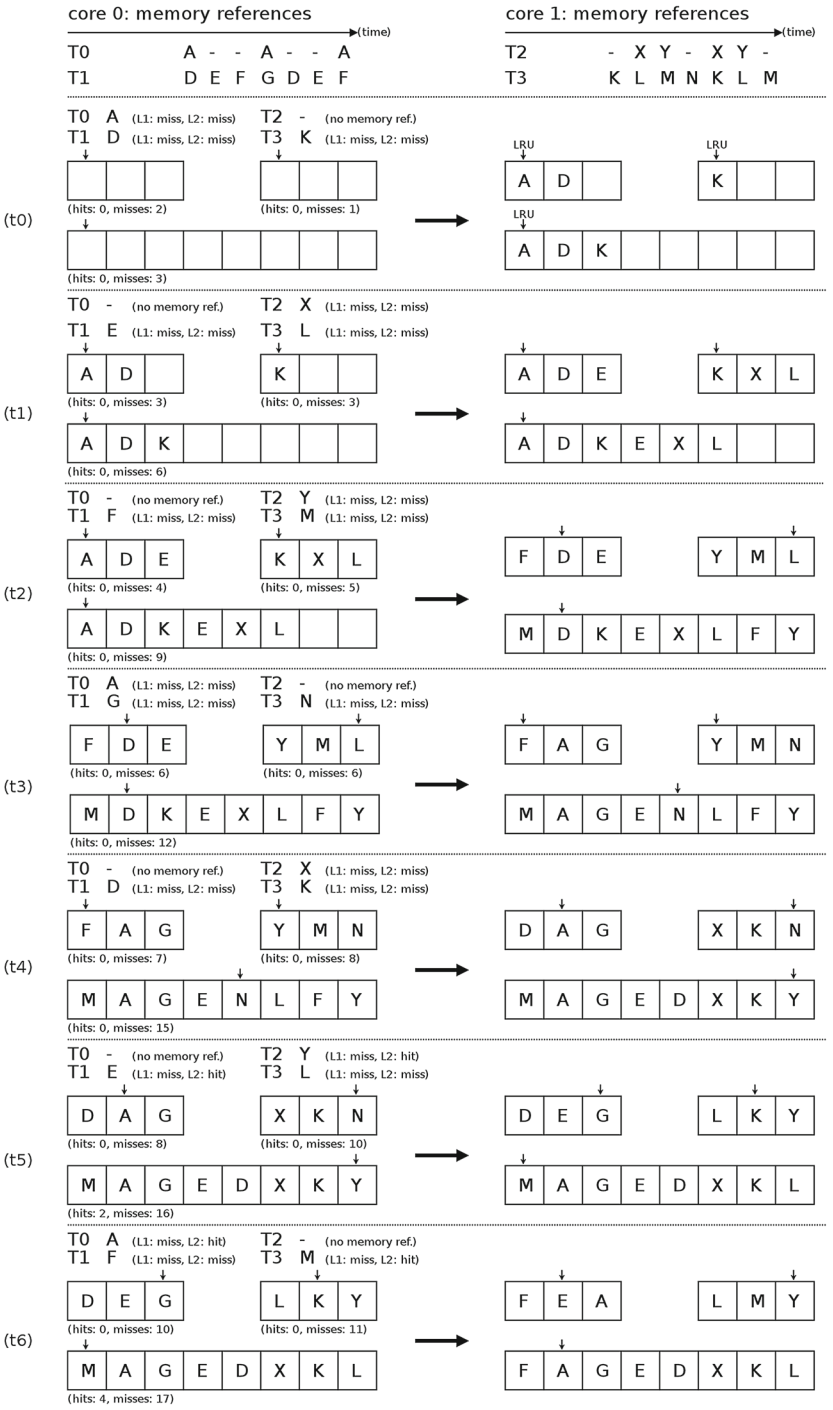

Fig. 6 Cache-hierarchy-unaware scheduling. The number of L2 hits is four, while the number of L2 misses is 17

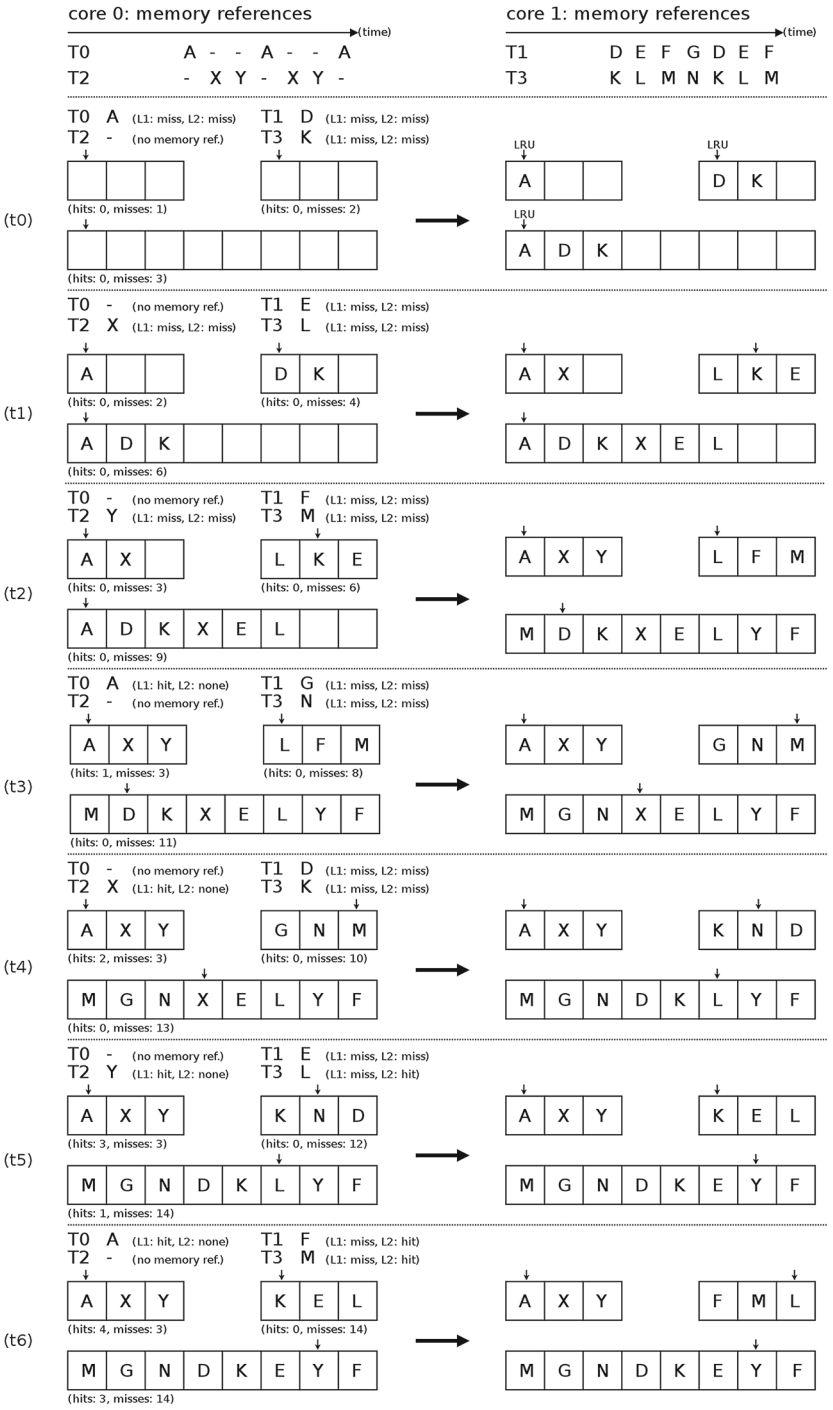

Fig. 7 Cache-hierarchy-aware scheduling. The number of L2 misses reduced to 14

Table 1 Chip multiprocessor and memory configuration for evaluations

# Cores: 4	2 threads per core
Architectural parameters	
Private L1 cache (combined)	16 sets, 2-way assoc.
Replacement Policy = LRU	block size = 64, latency = 1 cycle
Shared L2 cache	64 sets, 4-way assoc.
Replacement Policy = LRU	block size = 64, latency = 10 cycles
Main Memory	128 sets, 8-way assoc.
Replacement Policy = LRU	block size = 64, latency = 100 cycles

Table 2 PARSEC benchmarks used in evaluations

Blackscholes (black)	Stock-option pricing
Canneal	Heuristic for routing cost minimization
dedup	Compression with data deduplication
facesim (face)	Simulation of a human face motion
fluidanimate (fluid)	Fluid dynamics animation
freqmine (freq)	Frequent itemset mining
vips	Image processing
x264	H.264 video encoding

consider the effects of L1 hits on core 0. In reality, core 0 is most likely generate more memory requests compared to core 1, since core 0 can continue issuing instructions in a higher rate due to higher L1 hit ratio.

5 Evaluation

5.1 Simulation Environment

We performed our experiments on multi2sim simulation framework that is developed to evaluate multicore-multithreaded processors [10]. Otherwise specified, we used the configuration given in Table 1 for chip multiprocessor and main memory.

We used PARSEC benchmarks to evaluate our proposed adaptive cache-hierarchy-aware thread scheduling algorithm. PARSEC is a set of multithreaded programs focusing on emerging workloads and was designed to be a representative set of next-generation shared-memory programs for chip multiprocessors [11]. In our experiments, we used eight benchmarks from PARSEC suite. Details of the benchmarks are given in Table 2.

dedup uses the pipeline parallelization model with a dedicated pool of threads for each pipeline stage. *facesim* and *fluidanimate* are streaming programs. *blackscholes*, *canneal*, *freqmine*, *vips*, and *x264* are data-level parallel programs with different amount and patterns of synchronizations and inter-thread communications.

Table 3 Static schedules used in evaluations

Schedule	Core 0	Core 1	Core 2	Core 3
S1	black–vips	canneal–dedup	face–x264	fluid–freq
S2	black–canneal	vips–dedup	face–fluid	x264–freq
S3	black–dedup	vips–canneal	face–freq	x264–fluid
S4	black–face	vips–fluid	canneal–x264	dedup–freq
S5	black–x264	vips–freq	canneal–face	dedup–fluid
S6	black–fluid	vips–x264	canneal–freq	dedup–face
S7	black–freq	vips–face	canneal–fluid	dedup–x264

At the very beginning of evaluations, we collected profiling regarding all the benchmarks. We run each benchmark along with other benchmarks one by one on the same core and observed their respective performances. Then, we select the best schedules that maximize the performance (i.e., IPC) by using this profiling. At each interval, we scheduled threads in a way that the overall performance of the IPC is maximized. We referred these schedules as *dynamic-offline* and we used them as a baseline to compare against the proposed adaptive cache-hierarchy-aware thread scheduler. We also compared our adaptive cache-hierarchy-aware thread scheduler with possible static schedules.

Although IPC of threads obtained during offline profiling do not match the one obtained on the fly due to interactions of other scheduled threads, it provides a very good estimate of the highest IPC that can be achieved. Throughout the experiments, we observed that adaptive cache-hierarchy-aware thread scheduling outperforms static schedules and it is very close to the IPC achieved by dynamic-offline schedule.

We have generated seven different static schedules. Since there are eight benchmarks, we allowed a benchmark to run with a different one on the same core in each schedule. By doing so, we aimed to cover all possible schedules for eight benchmarks (running on a four-core chip multiprocessor). We permuted the scheduled threads and generate distinct thread combinations. Since there are eight benchmarks, each benchmark can be scheduled with the remaining seven benchmarks at most. Note that, it does not matter on which core the two threads are scheduled; however, it matters which threads are scheduled together. The static schedules generated and corresponding threads running on cores are given in Table 3.

5.2 The Effect of Scheduling on System Performance

In this set of experiments, we compared the effect of different scheduling schemes on performance for each benchmark (i.e., IPC). Figure 8 shows the IPC of each benchmark under different schedules.

As it can be seen from Fig. 8, different schedules increase the performance for different benchmarks. There is no single schedule that outperforms the others for all

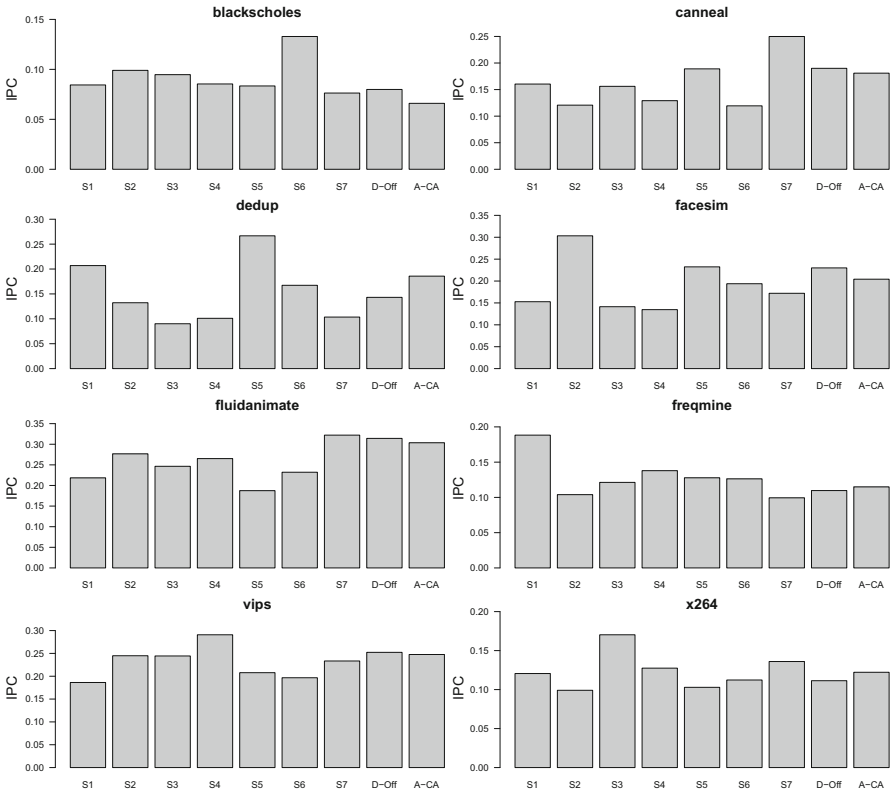

Fig. 8 Performance of benchmarks under different scheduling schemes

benchmarks. This is also true for our proposed adaptive cache-hierarchy-aware scheduler. An important observation from this figure is that it is necessary to understand the main dynamics of the overall performance. Instead of increasing the performance of a particular benchmark, it is more desirable to find a balance among the performance of all threads. Our adaptive cache-hierarchy-aware scheduler works towards this goal. It tries to maximize the performance of overall system, not the performance of a particular thread. So, adaptive cache-hierarchy-aware scheduler does not favor (unfairly) a particular thread that may contribute to the overall performance the most (i.e., a thread that has highest potential to increase IPC in case of more resources are given to it). Instead, it tries to find a balance among threads where they contribute to the overall system performance.

In the next set of experiments, we compared the overall system performance provided by different scheduling schemes. The results are given in Fig. 9. As it can be seen, our adaptive cache-hierarchy-aware scheduler outperforms all the static schedules, and barely left behind the dynamic-offline scheduling. Note that, dynamic-offline provides the highest IPC that can be achieved; however, it requires profiling in advance. For this reason, dynamic-offline is not a practical scheduler, but it helps us to evaluate our approach against. Figure 9 also shows the effectiveness of adaptive cache-hierarchy-

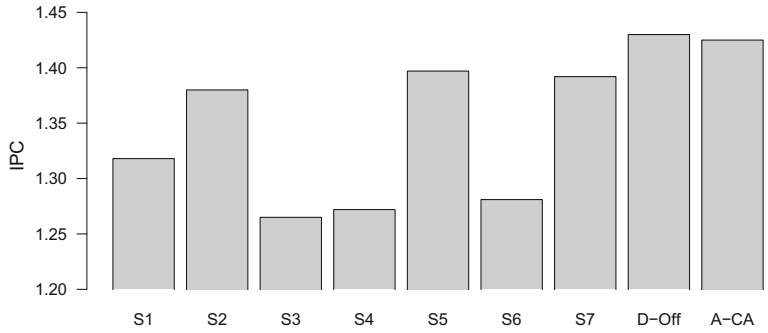

Fig. 9 Comparison of overall system performance under different scheduling schemes

aware scheduler on maximizing overall system performance without unfairly favoring certain benchmarks.

5.3 Slowdown of Benchmarks

It may be misleading to focus solely on IPC when evaluating scheduling schemes. As we discussed in Sect. 5.2, the overall system performance can be maximized by unfairly favoring threads that have higher potential to contribute to the overall system IPC. However, such an approach is not desirable in most cases. Instead, the system performance has to be maximized such that each thread contributes to the overall system performance as much as possible while interference with other threads is minimized. In other words, fairness should not be traded for performance. The schedulers proposed for chip multiprocessors should also take slowdown of threads into account while trying to maximize system performance.

In this set of experiments, we analyzed the behavior of a particular benchmark when it is scheduled with a different one. Table 4 shows the slowdown of benchmarks when they run concurrently with another benchmark. Slowdown specifies the degree of vulnerability of a benchmark to the interference of the other thread.

The slowdowns given in Table 4 are observed on a single core that can run two threads concurrently. There is only one schedule possible, since two threads exist in this set of evaluations.

As noted earlier, a desired scheduler should maximize the system performance while preserving fairness. Since slowdowns of benchmarks provide a notion of fairness, we can evaluate the effectiveness of schedulers on maximizing system performance fairly. Table 5 shows the slowdown of benchmarks when scheduled with another thread on the same core, while the rest of the benchmarks are running on other cores under the static scheduling. The slowdowns presented in the rest of the section are observed on a quad-core chip multiprocessor (i.e., the configuration given in Table 1).

Compared to static schedules, threads can be scheduled with different threads throughout the execution in dynamic-offline scheduling. For this reason, we represented overall slowdown for each benchmark and difference between the minimum, maximum, and average slowdown observed under the static schedules. Table 6 shows

Table 4 Slowdown of a thread when scheduled with another thread on the same core

	running with							
	black	canneal	dedup	face	fluid	freq	vips	x264
black	–	2.1	3.4	2.5	1.7	2.6	2.9	3.5
canneal	2.8	–	4.3	2.5	1.5	2.8	2.7	3.0
dedup	2.1	1.6	–	1.7	1.3	1.9	1.9	2.1
face	3.2	2.6	4.4	–	1.7	2.8	3.2	3.3
fluid	2.9	2.3	6.2	2.4	–	2.7	2.6	2.9
freq	3.3	2.8	4.3	2.7	1.9	–	3.3	3.4
vips	1.8	1.4	2.2	1.5	1.2	1.6	–	1.9
x264	3.0	2.0	2.9	2.1	1.5	2.2	2.5	–

Table 5 Slowdown of a thread when scheduled with another thread under the static scheduling scheme

	running with							
	black	canneal	dedup	face	fluid	freq	vips	x264
black	–	4.8	5.0	5.6	3.6	6.3	5.7	5.7
canneal	6.1	–	4.6	3.9	2.9	6.2	4.7	5.7
dedup	5.5	2.4	–	3.0	1.9	4.9	3.8	4.8
face	6.7	3.9	4.6	–	3.0	6.3	5.2	5.9
fluid	5.6	4.0	6.9	4.6	–	5.9	4.9	5.2
freq	8.0	6.3	5.8	6.6	3.6	–	6.2	7.7
vips	3.0	2.3	2.3	2.4	1.9	2.7	–	2.9
x264	4.7	3.8	3.6	4.0	2.8	4.9	4.3	–

the slowdown of benchmarks under dynamic-offline scheduling. The first column of the table specifies the overall slowdown of benchmarks. The second, third, and fourth columns of the table specify how much overall slowdown of a thread deviates from minimum, maximum, and average slowdown observed under the static schedules, respectively. In other words, the second column of the table is calculated as the subtraction of the minimum slowdown for a thread under the static scheduling from the slowdown of a thread under the dynamic-offline scheduling. The third and fourth columns are calculated similarly.

Similar to the dynamic-offline scheduling, adaptive cache-hierarchy-aware scheduling allows threads to be scheduled with different threads throughout the execution. Table 7 shows the slowdown of benchmarks under the adaptive cache-hierarchy-aware scheduling. Likewise, the first column of the table specifies the overall slowdown of benchmarks. The second, third, and fourth columns of the table specify how much overall slowdown of a thread deviates from minimum, maximum, and average slowdown observed under the static schedules, respectively.

As indicated earlier, a desired scheduler should also try to minimize average slowdown while trying to increase system performance. To this end, our proposed adaptive

Table 6 Slowdown of a thread when scheduled with another thread under the dynamic-offline scheduling scheme

	Slowdown	δ from		
		Min. (+/−)	Max. (+/−)	Avg. (+/−)
black	6.0	2.4	−0.3	0.7
canneal	3.9	0.9	−2.3	−1.0
dedup	3.5	1.6	−2.0	−0.3
face	3.9	0.9	−2.8	−1.2
fluid	4.1	0.1	−2.8	−1.2
freq	7.3	3.6	−0.8	0.9
vips	2.2	0.3	−0.8	−0.3
x264	4.3	1.5	−0.5	0.3

Table 7 Slowdown of a thread when scheduled with another thread under the cache-hierarchy-aware scheduling

	Slowdown	δ from		
		Min. (+/−)	Max. (+/−)	Avg. (+/−)
black	7.3	3.7	1.0	2.0
canneal	4.1	1.1	−2.1	−0.8
dedup	2.7	0.8	−2.8	−1.1
face	4.4	1.4	−2.3	−0.7
fluid	4.2	0.2	−2.6	−1.1
freq	6.9	3.3	−1.1	0.6
vips	2.3	0.3	−0.7	−0.2
x264	4.0	1.1	−0.9	−0.1

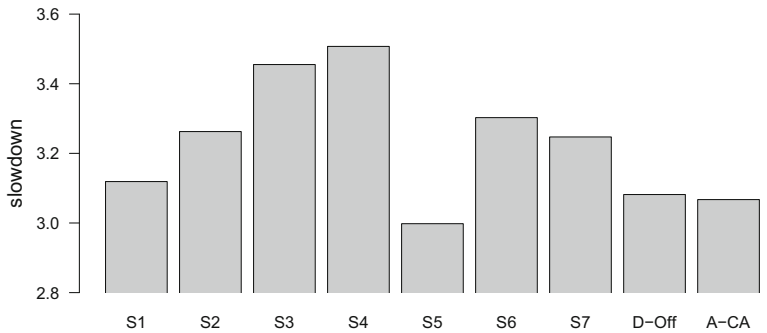

Fig. 10 Slowdowns of benchmarks under different scheduling schemes

cache-hierarchy-aware scheduler obtains decent slowdown and provides higher system performance. Figure 10 shows the comparison of slowdown of all the benchmarks running under different scheduling schemes.

Proposed adaptive cache-hierarchy-aware scheduler outperforms all other scheduling schemes, including dynamic-offline, except the fifth static schedule (i.e., S5). However, slowdown observed in S5 and adaptive cache-hierarchy-aware scheduler are very close. In addition, although S5 has lower slowdown, it does not improve

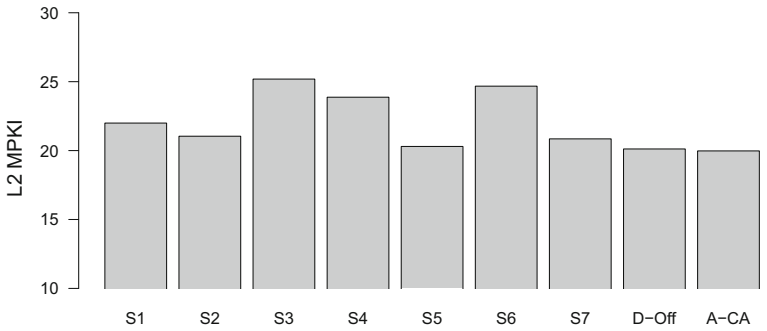

Fig. 11 L2 misses per kilo-instructions of benchmarks under different scheduling schemes

the system performance as much as adaptive cache-hierarchy-aware scheduler. If we consider both slowdown and system performance, we can conclude that the adaptive cache-hierarchy-aware achieves better results compared to S5.

5.4 The Effect of Scheduling on Cache Performance

An important metric to evaluate the effectiveness of a scheduler is LLC (i.e., L2 in our case). In this section, we justify why considering higher level of cache hierarchy in scheduling decisions will eventually affect the performance of LLC.

We used L2 misses per kilo-instructions (MPKI) as a metric to evaluate the effectiveness of the scheduling schemes. The scheduling scheme that minimizes the L2 MPKI is more desirable than the others. Figure 11 shows the L2 MPKI for different scheduling schemes. As it can be seen, our proposed adaptive cache-hierarchy-aware scheduler has the lowest L2 MPKI, thereby, justifying our claim on the importance of higher level caches on LLC performance.

Figure 12 shows the L2 miss rates of the benchmarks for different scheduling schemes. Although adaptive cache-hierarchy-aware scheduler does not have the minimum L2 miss rate, we believe that it is reasonable. The results of L2 miss rate might be misleading if it is considered without taking corresponding system performance

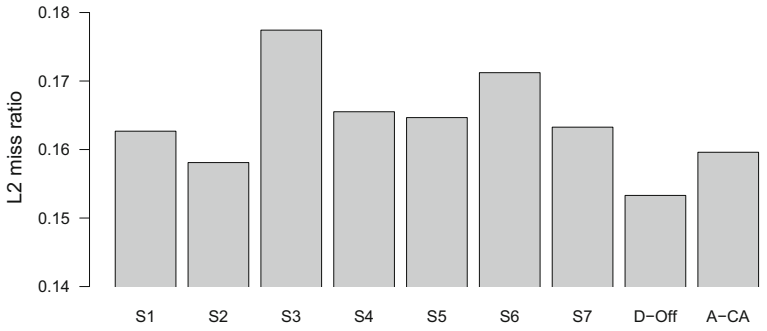

Fig. 12 L2 miss rates of benchmarks under different scheduling schemes

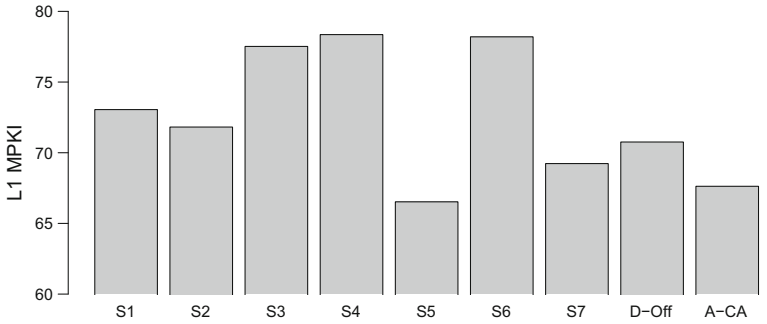

Fig. 13 L1 misses per kilo-instructions under different scheduling schemes

(i.e., IPC) into account. There might be cases where threads make slow progress due to the contention on shared resources, thus generating less number of cache accesses. These cache accesses might have higher hit rate. On the other hand, there might be cases where threads make faster progress, thanks to wise scheduler that reduces the contention on shared resources, thus generating more cache accesses. These accesses might have lower hit rate compared to the first case. However, we can not conclude that the scheduling in the first case is better than the second one, solely based on it has the lower miss rate. In fact, we need to look at what would be the miss rate when the threads in the first case would also make the same progress as the threads in the second case. For this reason, we used L2 misses per kilo-instructions as a metric for performance of cache, instead of L2 miss rate.

As it can be seen from Fig. 12, adaptive cache-hierarchy-aware scheduling has higher L2 miss rate compared to the second static schedule (i.e. S2). However, the overall system performance of adaptive cache-hierarchy-aware scheduling is comparably higher than the performance of S2.

Likewise, the adaptive cache-hierarchy-aware scheduling utilizes the L1 cache much better compared to other scheduling schemes. Figure 13 shows the L1 MPKI for different schedules. Adaptive cache-hierarchy-aware scheduler outperforms other scheduling schemes except the fifth static scheduling (i.e. S5). Although S5 has lower MPKI, its overall system performance is lower than the adaptive cache-hierarchy-aware scheduler. Despite it seems awkward, there is a logical reason behind it. The accesses to L1 that are misses go to L2 cache. Some of these misses are also misses in L2 cache. Thus, these misses require high latency main memory accesses. Compared to adaptive cache-hierarchy-aware scheduler, the L1 misses of S5 are not found in L2, so they have to be fetched from main memory. That is why S5 has lower IPC than adaptive cache-hierarchy-aware scheduler, although it has lower L1 MPKI.

Figure 14 shows the L1 miss rate of benchmarks under different scheduling schemes. Similar to the L2 miss rate, adaptive cache-hierarchy-aware scheduler does not have the minimum L1 miss rate. The same argument is also valid in this case. The results of L1 miss rate might be misleading if it is considered without taking corresponding system performance (i.e., IPC) and L2 MPKI into account. Note that, misses on L1 might be misses on L2, as well. In such cases, high latency memory access reduces the system performance. This is why the seventh static scheduling (i.e., S7)

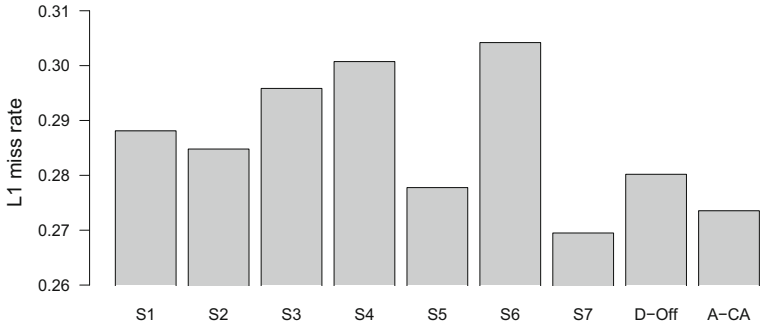

Fig. 14 L1 miss rates of benchmarks under different scheduling schemes

has lower IPC compared to adaptive cache-hierarchy-aware scheduler, although it has lower L1 miss rate as shown in Fig. 14. The same observation is valid for dynamic-offline. Although adaptive cache-hierarchy-aware scheduler has lower L1 miss rate, dynamic-offline has higher IPC compared to adaptive cache-hierarchy-aware scheduler.

5.5 Sensitivity of Performance to the Thread Quantum

Threads have a time quantum that is specified as the number of cycles to be executed. When threads exceed this quantum, the adaptive cache-hierarchy-aware scheduler updates the scheduling decisions as explained in Sect. 4. After this update, the quanta of threads are reset and they run on specified cores until the time quantum is exceeded again.

The number of cycles specified for quanta of threads has an influence on the performance. When the length of quantum is short (i.e., small number of cycles), the scheduling decision has to be made more often. The drawback of short quantum is that the decision of scheduling becomes vulnerable to short bursts and fluctuations on thread behaviors. In addition, the length of quantum may not be sufficient to compensate the overhead due to thread migration (in case a thread is scheduled on a different core).

On the other hand, if the length of quantum is too long, then the scheduling decision has to be made less often. The drawback of long quantum is that the execution characteristics of threads may change which may result in with inappropriate scheduling. For this reason, the length of quantum has an impact on the overall system performance.

Figure 15 shows the effect of quantum length on overall system performance. As it can be seen, the quantum of 100,000 cycles maximizes the system performance. The results reported in this chapter are gathered by using a quantum with 100,000 cycles.

5.6 Sensitivity of Performance to the Weights of Thread Attributes

As explained in Sect. 3, the position of a bit (that represents a particular attribute of a thread) in an attribute vector determines the relative weight of the attribute of interest.

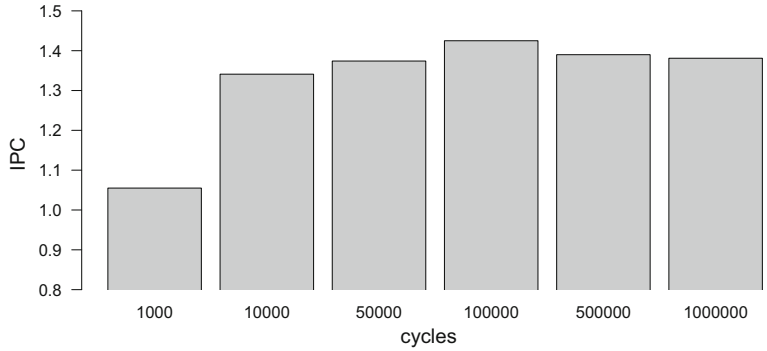

Fig. 15 System performance for different thread quantum lengths

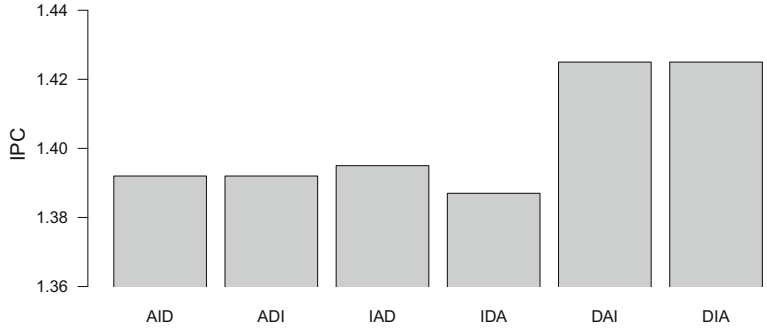

Fig. 16 System performance changes with respect to relative weights of thread attributes

The bit for an attribute with the most significant bit in the attribute vector naturally obtains the highest weight, while the bit for an attribute with the least significant bit in the attribute vector obtains the lowest weight. The weights of attributes (i.e., position of corresponding bits in attribute vector) have an impact on the overall system performance.

Figure 16 shows the effect of changing the position of bits for attributes in attribute vector. Each column represents different ordering of bits for attributes in an attribute vector. For example, ADI means that the bit for aggressiveness is the most significant bit in attribute vector, thus it has the highest weight. On the other hand, the bit for inefficacy is the least significant bit in the attribute vector, thus it has the lowest weight.

We used DAI (i.e., density being the most important attribute and inefficacy being the least important attribute) for the results reported in this chapter. The relative importance of density is comprehensible, since L1 cache is limited in size and the contention for cache blocks is severe. Thus, giving more weight to density allows adaptive cache-hierarchy-aware scheduler to have tendency to schedule threads that reduce such contention.

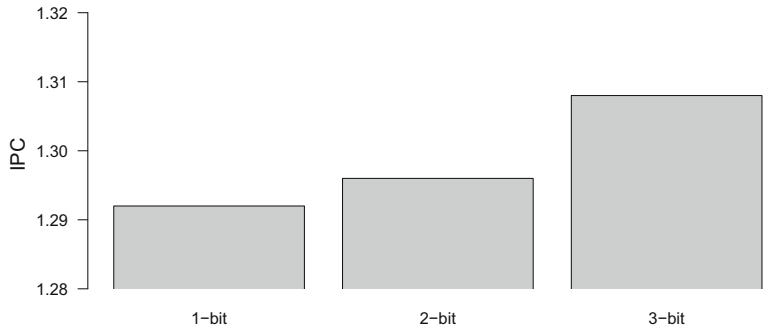

Fig. 17 System performance with respect to the number of bits used to represent each attribute in an attribute vector

5.7 Sensitivity of Performance to the Resolution of Thread Attributes

Figure 17 shows the effect of the number of bits used for each attribute of a thread in an attribute vector. Fine-grained scores are possible when higher number of bits is used. Although one bit for each attribute is fairly enough in case of a small number of threads, it becomes harder to differentiate candidate schedules (i.e., deciding which one is better) when the number of threads increases. This is the case, because there is less number of distinct CoScores possible with less number of bits. For this reason, increasing the number of bits per attribute in attribute vector enables fine-grained scores. Figure 17 shows the overall system performance for 16 threads running on 8 cores. Notice that, increasing the number of bits results with a higher system performance.

It provides marginal benefit to use more bits for each attribute for our base simulation environment where 8 threads are running on four cores. For this reason, we use a single bit per attribute for the results reported.

5.8 Sensitivity of Performance to Scoring Thresholds

As discussed, each thread has a multi-metric score based on its attributes: aggressiveness, density and inefficacy. The decision of a thread being aggressive/complaisant, dense/sparse and sterile/prolific is given through respective thresholds. Each attribute has its own threshold, where these thresholds are determined empirically. Table 8 shows the overall system performance with respect to different thresholds for the attributes.

The aggressiveness, density and inefficacy of a thread are determined as specified in Sect. 3. The overall system performance is maximized when thresholds are $\tau_a = 1.6$, $\tau_d = 0.3$ and $\tau_i = 0.4$ where τ_a, τ_d and τ_i are thresholds for aggressiveness, density, and inefficacy, respectively. The maximum performance obtained is specified as bold in the table. We used $\tau_a = 1.6$, $\tau_d = 0.3$ and $\tau_i = 0.4$ for the results collected in our experiments.

Table 8 System performance with respect to the attribute thresholds for threads

D	I	A			
		0.3	0.4	0.5	0.6
1.4	0.2	1.394	1.402	1.392	1.4
	0.3	1.374	1.402	1.394	1.381
	0.4	1.388	1.386	1.381	1.394
1.5	0.2	1.409	1.412	1.389	1.389
	0.3	1.393	1.393	1.398	1.405
	0.4	1.416	1.393	1.391	1.391
1.6	0.2	1.422	1.402	1.411	1.416
	0.3	1.407	**1.425**	1.409	1.409
	0.4	1.414	1.395	1.409	1.416
1.7	0.2	1.403	1.378	1.382	1.419
	0.3	1.405	1.341	1.393	1.403
	0.4	1.404	1.39	1.393	1.393

Bold number represents the maximum system performance is achieved and corresponding configuration (i.e., D = 1.6, I = 0.3, A = 0.4)

6 Related Work

The shift from uniprocessor to chip multiprocessor made scheduling a much more interesting and compelling problem. Jiang et al. [12] proved that scheduling in chip multiprocessors with more than two cores is an NP-complete problem. For this reason, there are numerous heuristics developed for scheduling in chip multiprocessors.

There are three main concerns regarding scheduling. The first one is to improve the computing efficiency (e.g., [13,14]). The second concern is fairness (e.g., [15]) and the last one is performance isolation (e.g., [16]). There are vast amount of studies targeted these concerns.

Deciding threads to be coscheduled is one part of the story. In addition to that, there is also a greater need to decide the amount of resource to be allocated to each thread. To this end, various replacement and cache partitioning strategies have been proposed. Notice that, scheduling algorithms are not alternatives to replacement and cache partitioning strategies; however, they all have impact on each other and, thereby in the overall system performance.

6.1 Replacement and Partitioning

The threads scheduled on the same core compete for shared cache resources. A request from a thread can conflict with a request from another one. A thread may need to evict data that belongs to a different thread in order to bring its own data into shared cache without considering whether the evicted data will be used by other threads, or not. Likewise, the benefits obtained through cache usage may differ among threads. Thus, allowing a thread to use more cache resources although it does not obtain much benefit from it, may prohibit the possible benefit that could be obtained by other threads. Such interference and evictions reduce the performance of multiple threads.

If they are not coordinated appropriately, such evictions can be destructive for the overall system performance. There are various eviction and replacement strategies such as Least-Recently-Used (LRU) [17–19] and sampling-based adaptive replacement (SBAR) [20]. In addition to replacement policies, there are various partitioning strategies such as way-partitioning [21] and cache partitioning [15,19,20].

It is challenging for an operating system scheduler to ensure a faster progress for a high-priority thread on a chip multiprocessor, because the performance of a thread could be arbitrarily decreased by a high-miss-rate thread that is running concurrently with this high-priority thread. Fedorova et al. [16] proposed an operating system scheduler to ensure performance isolation in such cases. In their proposal, threads running concurrently with similar cache miss rates get equal cache allocations. The shared cache is allocated based on demand; so, if the threads have similar demands they will have similar cache allocations.

To improve the cache access efficiency and system performance both replacement and scheduling strategies should be in place. Therefore, LRU or way-partitioning schemes are orthogonal to our cache-hierarchy-aware scheduling. Any replacement policy can be used along with cache-hierarchy-aware scheduler. It is beyond the scope of this work to tune replacement policy that would work best with the proposed cache-hierarchy-aware scheduler. Rather, we focus on the cache access characteristics of threads and try to come up with the best scheduling in which scheduled threads have the least interference and the number of evictions is minimized.

6.2 Cache-Sharing-Aware Scheduling

Cache-sharing-aware scheduling in operating systems can mitigate the cache contention among scheduled threads by assigning threads that can benefit from running on the same core by sharing data. Such cache-sharing-aware scheduling schemes can improve cache usage efficiency and program performance considerably in an environment where data sharing among threads is considerable. However, Zhang et al. [5] showed that cache sharing has insignificant impact on performance of modern applications. This is due to the fact that there is very limited sharing of the same cache block among different threads in such applications. These applications are highly parallelized, where each thread is working on a different block that are independent from each other. For this reason, it is very unlikely that they will access the same data block, so cache-sharing-aware schedulers have limited applicability.

Tam et al. [6] proposed a scheduling scheme to schedule threads based on data sharing patterns that are detected online through hardware performance counters. The proposed scheme detects data sharing patterns and clusters threads based on the data sharing patterns. Then, the scheduler tries to map threads that belong to the same cluster onto the same processor, or as close as possible to reduce the number of remote cache accesses for shared data.

Settle et al. [9] developed a memory monitoring framework providing statistics in simultaneous multithreaded processors. Statistics regarding memory accesses of threads gathered from the proposed framework can be used to build a scheduler that minimizes capacity and conflict misses. For each thread, L2 cache accesses are moni-

tored on a set basis to generate per-thread cache activity vectors. These vectors indicate the sets that are accessed most of the time. The intersection of these vectors specifies the sets that are likely to be conflicting. This information is then used in scheduling decisions.

As noticed earlier, there is limited sharing of the same cache block among different threads in chip multiprocessor workloads that makes cache-sharing-aware scheduling inefficient. In contrast, our cache-hierarchy-aware scheduler considers more generic interactions (not just sharing) among threads that makes it applicable and efficient for diverge set of chip multiprocessor workloads.

6.3 Phase Prediction and Thread Classification

Sherwood et al. [22] introduced phase prediction method based on basic block vectors. Basic block vector represents the code blocks executed during a given interval of execution.

Chandra et al. [23] focused on L2 cache contention on dual-core chip multiprocessors. They proposed analytical model to predict number of L2 cache misses due to contention of threads on L2 cache.

Cazorla et al. [24] introduced a dynamic resource control mechanism and allocation policy in simultaneous multithreaded processors. The policy monitors the usage of resources by each thread and tries to fairly allocate resources to each thread to avoid monopolization. It classifies threads into groups based on cache access patterns as *fast* and *slow*. Then, it allocates the resources to these groups accordingly. Threads with pending L1 data misses are classified as members of the *slow* group and the ones without any pending L1 data misses are classified as members of the *fast* group. Another classification is made as *active* and *inactive*, based on the usage of certain resources. This classification allows borrowing resources from an inactive thread for the sake of an active one. Our approach is similar to theirs since they also used pending L1 data misses as a classification method; however, our approach differs in variety of ways. First, we use multiple L1 access characteristics such as number of accesses, miss ratio and number of evictions that provides better representation of execution characteristics of threads. Second, they do not rely on L1 access statistics for scheduling, instead they use it for clustering threads. Third, the goal of this work is not to develop a scheduler, but it is to develop a dynamic allocation policy for shared resources.

El-Moursy et al. [13] introduced a scheduling algorithm in which threads are assigned to processors based on the number of ready and in-flight instructions. The number of ready and in-flight instructions are strong indicators of different execution phases. The algorithm tries to schedule threads that are in compatible phases. They also used hardware performance counters to gather information required to assess the compatibility of thread phases.

Kihm et al. [25] proposed a memory monitoring framework that makes use of activity vectors that allow scheduler to estimate and predict cache utilization and inter-thread contention dynamically. However, they do not propose any scheduling algorithm that actually employs activity vectors.

6.4 Coscheduling

Tian et al. [26] proposed an A*-search-based algorithm to accelerate the search for optimal schedules. They formulated optimal co-scheduling as a tree-search problem and developed A*-based algorithm to find optimal schedule. The authors reduced constraints on finding optimal scheduling such that they allowed threads of different lengths. Further, they developed and evaluated two approximation algorithms, namely A*-cluster and local-matching. A*-cluster algorithm is a derivative of A*-search-based algorithm that employs online adaptive clustering. It trades accuracy for scalability. The local-matching algorithm, on the other hand, applies graph theory to find the best schedule at a given time without any provision for the upcoming schedules. Although optimal scheduling algorithms are costly and inefficient for practical purposes, they can provide insights to enhance the practical scheduling algorithms and associated complexities with them.

Jiang et al. [27] proposed a reuse-distance based [28] locality model that provides proactive prediction of the performance of scheduled processes. The prediction is used in run-time scheduling decisions. They employed the proposed locality model in designing cache-contention-aware proactive scheduling that assigns processes to the cores according to the predicted cache-contention sensitivities. However, predictive model has to be constructed for each application through an offline profiling and learning process.

Snavely et al. [14] introduced a symbiotic scheduler, called SOS (Sample, Optimize, Symbios) simultaneous multithreaded processor. It identifies the characteristics of threads that are scheduled through sampling. SOS runs in two distinct phases: sample phase and symbiosis phase. It gathers information about threads running together in different schedule permutations during the sample phase. After this sample phase, SOS picks the schedule that is predicted to be the optimal and proceeds to run this schedule in the symbiosis phase. The performance metrics of a schedule are gathered through hardware counters. SOS employs many predictors to identify the best schedule. One interesting result provided by Snavely et al. is that IPC alone is not a good predictor. It may happen that threads with higher IPCs monopolize system resources and can be detrimental to threads with lower IPCs. The limitation of this work is that it tries many schedules during sample phase to predict the best schedule to be executed in symbiosis phase. For workloads that are composed of many threads exceeding the available hardware resources, the sample phase would be much longer. In such a scenario, threads can change their characteristics that would not be reflected during the symbiosis phase. Therefore, symbiosis phase would be inaccurate due to the change in execution characteristics of threads during sample phase. Limited number of samples can be used to avoid longer sample phase; however, the probability of missing better schedules is increased in this case.

Suh et al. [29] proposed an online memory monitoring scheme that uses hardware counters to provide estimates for isolated cache hits/misses with respect to the cache size. The estimation does not require a change in the cache configuration. This is achieved by employing single pass simulation method introduced by Sugumar and Abraham [30]. The provided estimation is used in designing memory-aware scheduling

that schedules processes based on the cache capacity requirements. The marginal gains in cache hits for different sizes of cache for each process are monitored. Then a process that has low cache capacity requirement is scheduled with a process that has high cache capacity requirement to minimize the overall miss ratio.

DeVuyst et al. [31] proposed a scheduling policy for chip multiprocessors that allows unbalanced schedules (i.e., uneven distribution of threads among the available cores) if they provide higher performance and energy efficiency. The main challenge of allowing unbalanced schedules is to have an increased search space with a great extent.

Our cache-hierarchy-aware scheduler has similarities in goals; however, it is based on more general, representative, yet easy to determine and process attributes of threads. As opposed to previous coscheduling efforts, our scheduler makes use of hardware counters not to decide schedules directly, instead, they are used to infer the thread attributes (i.e. aggressiveness, density, and inefficacy) that guide the scheduling and help to provision how would threads get along with each other if scheduled together.

7 Conclusion and Future Work

In a chip multiprocessor, the choice of threads to be scheduled on the same core has significant impact on overall system performance. Inter-thread contention occurs since coscheduled threads are competing for shared resources. The primary shared resource that influence the performance is the cache. An efficient scheduling should minimize the contention for shared caches to maximize utilization and system performance. Since the execution characteristics of threads varies over time, the scheduling decision has to be remade based on provisioned behaviors of threads for the near future.

To address this, we propose a novel adaptive cache-hierarchy-aware thread scheduling algorithm that minimizes the number of accesses to the lower levels of cache/memory hierarchy and reduces the number of evictions due to contention. We use a fine-grained, multi-metric scoring scheme to classify threads with respect to their execution characteristics in the proposed scheduling algorithm. The metrics used in scoring are obtained from L1 cache, as opposed to LLC as has been done in most of the previous studies.

We observe that our adaptive cache-hierarchy-aware scheduler improves the performance (i.e., instruction per cycle) of the benchmarks used in this work by up to 12.6% with an average of 7.3% over the static schedules.

The cache partitioning techniques and replacement policies to improve LLC performance are orthogonal to our approach, so they can be used along with our adaptive cache-hierarchy-aware scheduling scheme. We believe that integration of partitioning techniques with our adaptive cache-hierarchy-aware scheduler will provide even higher performance. Similarly, employing efficient replacement policies will result in with reduced number of evictions and misses, thus will improve the performance even further.

As a future work, we will integrate cache partitioning and replacement policies with our adaptive cache-hierarchy-aware scheduler and evaluate the impact on system performance.

In addition to the multi-metric scoring scheme, the ability to predict/detect the regions of the cache that are used by threads can be helpful to minimize inter-thread conflicts. Such an ability will improve the performance even further. We left these enhancements as future work.

References

1. Moore, G.E.: Cramming more components onto integrated circuits. Proc. IEEE **86**(1), 82–85 (1998). https://doi.org/10.1109/JPROC.1998.658762
2. Olukotun, K., Nayfeh, B.A., Hammond, L., Wilson, K., Chang, K.: The case for a single-chip multiprocessor. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, New York, NY, USA, pp. 2–11 (1996). https://doi.org/10.1145/237090.237140
3. Tullsen, D.M., Eggers, S.J., Levy, H.M.: Simultaneous multithreading: maximizing on-chip parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 392–403. ACM, New York, NY, USA (1995). https://doi.org/10.1145/223982.224449
4. Kumar, R., Tullsen, D.M.: Compiling for instruction cache performance on a multithreaded architecture. In: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 419–429. IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
5. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 203–212. ACM, New York, NY, USA (2010). https://doi.org/10.1145/1693453.1693482
6. Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 47–58. ACM, New York, NY, USA (2007). https://doi.org/10.1145/1272996.1273004
7. Parekh, S.S., Eggers, S.J., Levy, H.M.: Thread-Sensitive Scheduling for SMT Processors. Technical report, University of Washington (2001)
8. Bulpin, J.R., Pratt, I.A.: Hyper-threading aware process scheduling heuristics. In: Proceedings of USENIX Annual Technical Conference, p. 27. USENIX Association, Berkeley, CA, USA (2005)
9. Settle, A., Kihm, J., Janiszewski, A., Connors, D.: Architectural support for enhanced SMT job scheduling. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 63–73. IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/PACT.2004.7
10. Ubal, R., Sahuquillo, J., Petit, S., López, P.: Multi2Sim: a simulation framework for CPU-GPU computing. In: Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (2007)
11. Bienia, C.: Benchmarking modern multiprocessors. Ph.D. thesis, Princeton University (2011)
12. Jiang, Y., Shen, X., Chen, J., Tripathi, R.: Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 220–229. ACM, New York, NY, USA (2008). https://doi.org/10.1145/1454115.1454146
13. El-Moursy, A., Garg, R., Albonesi, D.H., Dwarkadas, S.: Compatible phase co-scheduling on a CMP of multi-threaded processors. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, p. 141. IEEE Computer Society, Washington, DC, USA (2006)
14. Snavely, A., Tullsen, D.M.: Symbiotic jobscheduling for a simultaneous multithreaded processor. In: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 234–244. ACM, New York, NY, USA (2000). https://doi.org/10.1145/378993.379244
15. Kim, S., Chandra, D., Solihin, Y.: Fair cache sharing and partitioning in a chip multiprocessor architecture. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, pp. 111–122. IEEE Computer Society, Washington, DC, USA (2004) . https://doi.org/10.1109/PACT.2004.15
16. Fedorova, A., Seltzer, M., Smith, M.D.: Improving performance isolation on chip multiprocessors via an operating system scheduler. In: Proceedings of the 16th International Conference on Parallel

Architecture and Compilation Techniques, pp. 25–38. IEEE Computer Society, Washington, DC, USA (2007). https://doi.org/10.1109/PACT.2007.40
17. Denning, P.J.: The working set model for program behavior. Commun. ACM **11**(5), 323–333 (1968). https://doi.org/10.1145/363095.363141
18. Wong, W., Baer, J.L.: Modified LRU policies for improving second-level cache behavior. In: Proceedings of the 6th International Symposium on High Performance Computer Architecture, pp. 49–60 (2000). https://doi.org/10.1109/HPCA.2000.824338
19. Stone, H.S., Turek, J., Wolf, J.L.: Optimal partitioning of cache memory. IEEE Trans. Comput. **41**(9), 1054–1068 (1992). https://doi.org/10.1109/12.165388
20. Qureshi, M.K., Lynch, D.N., Mutlu, O., Patt, Y.N.: A case for MLP-aware cache replacement. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, USA, pp. 167–178 (2006). https://doi.org/10.1109/ISCA.2006.5
21. Chiou, D., Devadas, S., Rudolph, L., Ang, B.S., Chiouy, D., Chiouy, D., Rudolphy, L., Rudolphy, L., Devadasy, S., Devadasy, S., Angz, B.S., Angz, B.S.: Dynamic cache partitioning via columnization. In: Proceedings of Design Automation Conference (2000)
22. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 45–57. ACM, New York, NY, USA (2002). https://doi.org/10.1145/605397.605403
23. Chandra, D., Guo, F., Kim, S., Solihin, Y.: Predicting inter-thread cache contention on a chip multiprocessor architecture. In: Proceedings of the 11th International Symposium on High Performance Computer Architecture, pp. 340–351. IEEE Computer Society, Washington, DC, USA (2005). https://doi.org/10.1109/HPCA.2005.27
24. Cazorla, F.J., Ramirez, A., Valero, M., Fernandez, E.: Dynamically controlled resource allocation in SMT processors. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 171–182. IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/MICRO.2004.17
25. Kihm, J.L., Janiszewski, A.W., Connors, D.A.: Dynamically controlled resource allocation in SMT processors. In: Proceedings of International Conference on Computing, Communications and Control Technologies (2004)
26. Tian, K., Jiang, Y., Shen, X.: A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In: Proceedings of the 6th ACM Conference on Computing Frontiers, pp. 41–50. ACM, New York, NY, USA (2009). https://doi.org/10.1145/1531743.1531752
27. Jiang, Y., Tian, K., Shen, X.: Analysis and approximation of optimal co-scheduling on chip multiprocessors. In: Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, pp. 201–215. Springer, Berlin, Heidelberg (2010)
28. Ding, C., Zhong, Y.: Predicting whole-program locality through reuse distance analysis. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 245–257. ACM, New York, NY, USA (2003). https://doi.org/10.1145/781131.781159
29. Suh, G.E., Devadas, S., Rudolph, L.: A new memory monitoring scheme for memory-aware scheduling and partitioning. In: Proceedings of the 8th International Symposium on High Performance Computer Architecture, pp. 117–128. IEEE Computer Society, Washington, DC, USA (2002)
30. Sugumar, R.A., Abraham, S.G.: Set-associative cache simulation using generalized binomial trees. ACM Trans. Comput. Syst. **13**(1), 32–56 (1995). https://doi.org/10.1145/200912.200918
31. DeVuyst, M., Kumar, R., Tullsen, D.M.: Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, pp. 140–149. IEEE Computer Society, Washington, DC, USA (2006)