PAPER

Revisiting Shared Cache Contention Problems: A Practical Hardware-Software Cooperative Approach

Eunji PAK†a), ***Member***, **Sang-Hoon KIM**†, **Jaehyuk HUH**†, ***and*** **Seungryoul MAENG**†, ***Nonmembers***

SUMMARY Although shared caches allow the dynamic allocation of limited cache capacity among cores, traditional LRU replacement policies often cannot prevent negative interference among cores. To address the contention problem in shared caches, cache partitioning and application scheduling techniques have been extensively studied. Partitioning explicitly determines cache capacity for each core to maximize the overall throughput. On the other hand, application scheduling by operating systems groups the least interfering applications for each shared cache, when multiple shared caches exist in systems. Although application scheduling can mitigate the contention problem without any extra hardware support, its effect can be limited for some severe contentions. This paper proposes a low cost solution, based on application scheduling with a simple cache insertion control. Instead of using a full hardware-based cache partitioning mechanism, the proposed technique mostly relies on application scheduling. It selectively uses LRU insertion to the shared caches, which can be added with negligible hardware changes from the current commercial processor designs. For the completeness of cache interference evaluation, this paper examines all possible mixes from a set of applications, instead of using a just few selected mixes. The evaluation shows that the proposed technique can mitigate the cache contention problem effectively, close to the ideal scheduling and partitioning.

key words: *multi-core, resource contentions, cache partitioning, application scheduling*

1. Introduction

Sharing caches allows dynamic allocation of cache capacity among cores, thus improving the efficiency of limited on-chip cache resources. However, such sharing also causes an application to be negatively affected by co-running applications, when contentions on the shared cache lead to an inefficient allocation of cache resources. To address this contention problem, cache partitioning and application scheduling techniques have been extensively studied. Cache partitioning estimates the performance benefit of the given cache capacity and enforces some explicit capacity allocation to each core to maximize the overall throughput [1]–[4]. Application scheduling determines the grouping of applications sharing a cache, considering the possible positive or negative interference among applications [5]. Although scheduling is useful only when there are multiple shared caches, it reduces the negative impact of cache sharing with no extra hardware cost.

In this study, we revisit the cache contention problem with respect to scheduling and partitioning strategies. By examining all possible mixes from 24 benchmark applications, we evaluate possible negative interferences among applications, the effectiveness of scheduling and cache partitioning, and the mutual effects between the two mechanisms. We observed that a relatively small number of mixes exhibit severe negative interferences, and application scheduling mitigates such contentions quite effectively for many cases. However, there are some cases with a significant performance loss, where scheduling alone cannot reduce the contention, requiring additional partitioning support.

Based on these observations, we propose an inexpensive approach that mostly relies on application scheduling with a minor HW support for cache LRU insertion. In the proposed approach, scheduling uses only an approximate classification of application types based on cache miss rates. The proposed mechanism does not require any extra hardware to characterize application behaviors. Instead, performance monitoring counters already available in commercial microprocessors are used to measure the cache miss counts. Furthermore, we mostly eliminate the hardware complexity of cache partitioning technique with scheduling support. It removes any extra hardware to measure the utility of cache capacity for each application. Instead of using full hardware partitioning as proposed by prior work, the proposed approach uses the support for cache LRU insertion, which simply puts the new cache blocks of an offending application to the LRU position in common set-associative caches.

This study shows that instead of using complicated conflict-mitigating mechanisms previously proposed, scheduling based on approximate classification, backed by simple LRU insertion, is effective enough for practical purposes. The prior partitioning-based approaches require extensive changes to the current commercial processor designs. Considering severe cache contention problems occur only for a small subset of application mixes, such extensive changes may not be justified for processor designs, which commonly pursue conservative and evolutionary changes.

We evaluate the proposed practical approach through exhaustive application mix analysis so as not to exaggerate the benefit of the proposed technique unfairly. Instead of using tens of selected mixes from a benchmark suite, this paper examines all possible combinations of mixes in multiprogrammed environments, and the evaluation represent realistic contention scenarios. Our experiments show that the performance of the proposed approach is close to the ideal scheduling and partitioning scheme, when two cores

Manuscript received September 13, 2012.
Manuscript revised February 25, 2013.

†The authors are with the Computer Science Department, KAIST, South Korea.
a) E-mail: pakej@calab.kaist.ac.kr

DOI: 10.1587/transinf.E96.D.1457

share a cache, and the performance gain is slightly reduced, when four cores share a cache. However, in general, the approach with little extra design changes can provide near optimal performance.

The main contributions of this paper are as follows.

- This paper explores mutual effects between scheduling and partitioning by evaluating all possible mix combinations from a given set of applications. Instead of using several selected application mixes, this paper examines the entire space of application mixes to truly evaluate the impact of partitioning and scheduling.
- The proposed approach differs from the prior partitioning work, as it uses only application scheduling with a minor extension in the cache insertion mechanism. Unlike prior HW-based partitioning techniques, this work does not require complicated changes to estimate the utility of partitioning.
- This work also differs from the prior scheduling approaches, which rely purely on scheduling. Our analysis shows that pure scheduling cannot mitigate every case of cache interference. We add an inexpensive LRU insertion mechanism to address such cases.

In the rest of the paper, we first describe the existing cache partitioning and scheduling techniques in Sect. 2, and in Sect. 3, we explore the mutual impact of scheduling and partitioning with the exhaustive evaluation of application mixes and scheduling. In Sect. 4, we describe the contention behaviors with different types of application mixes, and in Sect. 5, we present our approach. Section 6 presents the experimental evaluation, Sect. 7 describes the cost of additional hardware, and Sect. 8 concludes the paper.

2. Background

2.1 Cache Partitioning

In shared caches, common LRU replacement policies treat all cache references from different cores equally. Such LRU replacement policies may allocate more cache blocks to the application which touches a large number of unique addresses but hardly accesses them again. It leads to an inefficient allocation of cache capacity and degrades the performance of co-running applications.

To mitigate such inefficient allocation of shared cache resource among cores, cache partitioning aims to maximize the utility of cache capacity by assigning more resources to applications which can benefit most from the given cache capacity [1]–[4]. Such per-application benefit curves for increasing capacity are often generated from *stack distance counts* which can be collected either online from an extra hardware [3], [4] or various software-based techniques [6]. The stack distance count is the number of cache hits for every recency position from the MRU to LRU. The number of hits in the ith position implies the number of extra hits which would be happen if the number of allocated ways increases from i to $i+1$. Based on that information, cache partitioning techniques decide the capacity allocation of each core, and enforce the allocation using either hardware-oriented mechanisms or software-oriented mechanisms with page coloring.

Hardware-based techniques partition a shared cache by modifying the cache replacement or allocation policies [1], [3], [4]. While these approaches may resolve cache contentions efficiently without any software changes, they require an additional hardware to profile cache access behaviors for stack distance counts and control cache allocation. Moreover, the way-granularity of partitioning may restrict the possible partitioning combinations, as the number of ways in a cache is limited.

Tracking the utility information of an application when it shares a cache with other application may lead to inaccurate utility curves, because the stack distance count is different from when it has the entire cache. To isolate the effect of accesses from other cores, several techniques have been used to estimate the utility curves on-line by auxiliary cache tags or by reserving a subset of cache sets for certain cores. UCP adds one set of shadow tags per core to track the cache utility [3]. However, to reduce the HW overheads, instead of tracking the utility of all cache sets, it tracks the utility from a small number of sampling sets. PIPP uses the leader set to track the utility information of specific core [4]. Some sets are reserved for a specific core by changing the replacement policy and those sets track the utility of that core. RRIP is based on the NRU (Not Recently Used) replacement policy, and uses dual insertion policies [1]. At runtime, it uses set dueling, which dedicates a subset of cache sets for each policy, to identify which policy is best suited for the application and change the policy.

Alternatively, software-oriented page coloring controls the allocation of cache capacity by managing the memory mapping from the logical to physical address space [2]. It does not require extra hardware and allows more partitioning choices. However, page coloring causes a well-known problem called re-coloring. When the allocation of cache capacity needs to be changed, the operating system must change virtual-to-physical page mappings for certain pages, allocating new physical pages, and copying the contents of the old pages to the new ones. Also, coloring can restrict the memory management policies by the operating system, since each application must be assigned to a specific set of colors.

2.2 Scheduling

When there are multiple shared caches, thread scheduling can also mitigate the resource contention on shared cache resources [5]. It groups threads to different shared caches, so that the contention on each shared cache is minimized. The approach by Zhuravlev et al. [5] profiles the miss rates of applications and distributes applications across caches to balance the number of total misses for each shared cache. Such a scheduling-based contention mitigation technique is cost-effective, because it requires neither extra hardware nor page coloring compared to partitioning. Although scheduling is useful only when there are multiple shared caches, most of the current servers use multiple sockets for their high efficiency per space. Furthermore, it is likely to have multiple shared caches even in a processor as the number of cores increases.

Jaleel et al. [7] study the scheduling strategy considering the underlying cache replacement policy. They have studied the interactions between the cache partitioning and scheduling for the first time and showed that the scheduling policy assuming an LRU replacement policy may be ineffective in a cache with a dynamic partitioning scheme, RRIP [1]. The proposed scheduler co-locates applications with the knowledge of underlying cache replacement policy and the cache utility information provided by extra hardware logic. Their approach requires some changes to examine the effect of different cache allocation policies for a subset of cache sets. Certain sets can be dedicated to each core, and other sets enforces a static partitioning scheme. The approach uses the sample sets with different allocation policies to determine the miss behaviors of each core.

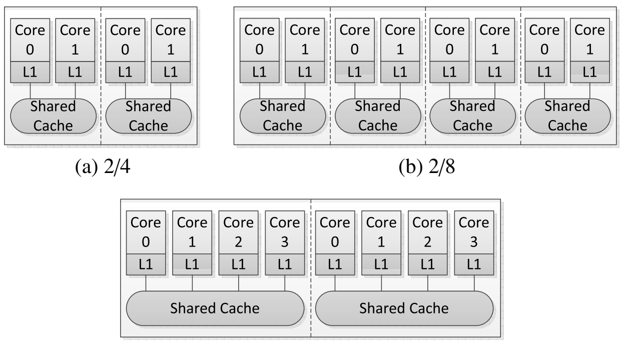
In our work, we also consider the scheduling and the cache partitioning at the same time. Unlike the work by Jaleel et al. [7], we are more focused on the mutual effects of those two mechanisms and suggest a practical approach that the scheduling and the cache partitioning complements each other. We mostly eliminate the hardware cost of cache partitioning by relying on scheduler support, and requires a negligible change to the current processor design for LRU insertion on caches.

3. Motivation

3.1 Experimental Methodology

To simulate a system with shared caches, we use the GEM5 simulator [8]. The target system has multiple shared caches, shared by either two or four cores. We evaluate three different configurations, 2/4, 2/8, and 4/8 with 4 and 8 total cores. In the rest of the paper, '*m*/*n*' denotes a configuration which has *n* total cores, and each shared L2 cache is shared by *m* cores. For example, 2/8 configuration has 8 cores with four shared L2 caches, and two cores share each shared cache. Figure 1 shows the three configurations used in this paper. Such systems with multiple shared caches are common in server systems with multiple sockets. Recent commercial multi-core architectures use an L2 shared by two cores, and a chip has multiple shared caches. The other details of core and cache configuration are shown in Table 1.

We use 24 SPEC CPU2006 benchmark applications in Table 2. The `type` in the table represents the characteristics of each application which is described in Sect. 4. `C`, `P`, or `M` indicates cache-sensitive, cpu-bound, or memory-bound application type. For the completeness of results, we evaluate all possible combinations of mixes out of 24 applications. For a system with 2/4 or 2/8 cofigurations, there are 10,626 or 735,471 application mixes, respectively. We ran each mix



(a) 2/4 (b) 2/8 (c) 4/8

Fig. 1 Configurations of cores and shared L2 caches.

Table 1 Architecture configuration.

Core	4 cores/8 cores, 4-wide, out-of-order
Private L1	2-way 64 KB I and D-cache, 1-cycle latency
Shared L2	64 B line-size, LRU shared by 2 cores: 1 MB, 8-way, 16-cycle latency shared by 4 cores: 2 MB, 16-way, 20-cycle latency
Memory	400-cycle access latency

Table 2 SPEC CPU2006 benchmarks.

Type	Applications
`C`	bzip2, hmmer, *omnetpp*, astar, *gamess*, *gromacs*, *soplex*, dealII
`P`	h264ref, *namd*, povray, calculix, GemsFDTD, *tonto*, *sphinx3*
`M`	sjeng,mcf,gobmk,*libquantum*,*bwaves*,milc,*zeusmp*,*leslie3d*,*lbm*

until it executes 200 million instructions after a warm-up execution of 1 billion instructions. For the 4/8 configuration, to reduce the simulation time, we pick 12 representative workloads which exhibit the characteristic of each category, and generate 495 application mixes with the workloads. The 12 selected workloads are expressed in *italic* in Table 2.

To evaluate the effect of scheduling, we run all possible scheduling choices for each mix and choose the best performing scheduling as an ideal scheduling case. For a 2/4, 2/8, and 4/8 configuration, there are 3, 105, and 35 different possible scheduling combinations respectively. In this section, to evaluate the effect of cache partitioning, we implemented cache partitioning similar to [3], [9], with an assumption that per-application *stack distance counts* are already known. However, we will later eliminate such requirements in our approach, and propose a much simpler approach with only miss rate information.

We use two metrics which are commonly used for measuring performance of concurrently running applications: *Weighted speedup* [3], [4] and *Unfairness* [10]. Weighted speedup indicates the average ratio of execution time reductions from an application mix. Unfairness is the performance variance of applications and it is computed as the difference between the maximum weighted speedup and minimum weighted speedup of all applications. Metrics are expressed quantitatively as follows:

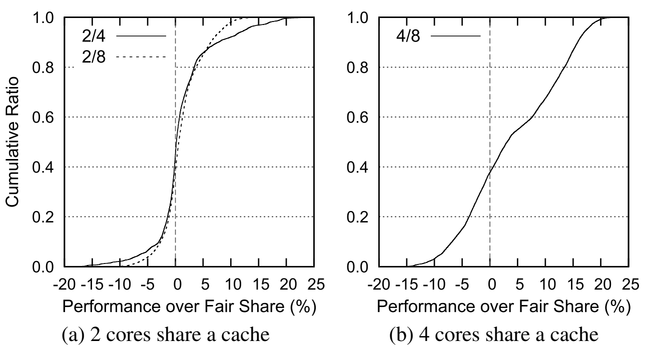


Fig. 2 Cumulative distribution of the performance over fair share. The negative performance indicates that an application cannot utilize the shared cache capacity due to cache contentions.

$$Weighted\ speedup = \sum_{i=1}^{N} \frac{IPC_i}{singleIPC_i} \ / \ N \tag{1}$$

$$Unfairness = \frac{\max(WS_1, WS_2 \cdots WS_N)}{\min(WS_1, WS_2 \cdots WS_N)} \tag{2}$$

where N is the number of applications which is equal to the number of cores in this paper, IPC_i is the instruction per cycle (IPC) of the i-th application when it concurrently executes with other applications, $singleIPC_i$ is the IPC when the application executes without sharing a cache, and $WS\,i$ is the $Weighted\ Speedup$ of the i-th application.

3.2 Effect of Scheduling and Partitioning

In this section, we evaluate the effect of cache contentions among applications, the effect of scheduling, and the effect of cache partitioning on the 2/4, 2/8, and 4/8 systems.

Figure 2 presents the cumulative distributions of the performance of shared caches with a common LRU replacement policy, compared to the *fair share*, which allocate the same amount of cache capacity for each application. To evaluate the effect of contentions without any mitigation by scheduling, we present the results from every possible scheduling combination of all mixes. For each run, the figure shows the average weighted speedup for a mix of applications. As shown in Fig. 2 for 2/4 mixes, 45% of mixes shows negative performance compared to the fair share. However, only less than 6.5% of runs exhibit more than 5% performance degradation due to contention on shared caches. Unlike our initial expectation, for the entire space of application combination and scheduling choice, the cases with performance degradation due to contention are relatively infrequent in 2/4 and 2/8 cases, although for such contention cases, performance impact can be significant for some applications. For 4/8, the effect of cache contention increases, compared to 2/4 or 2/8 cases. When four cores share a cache, one obtrusive application can potentially affect the rest three applications sharing the same cache, lowering the weighted speedup of the three applications. Due to the amplified effect, the performance effects appear much higher in 4/8 than in 2/4 or 2/8 cases.

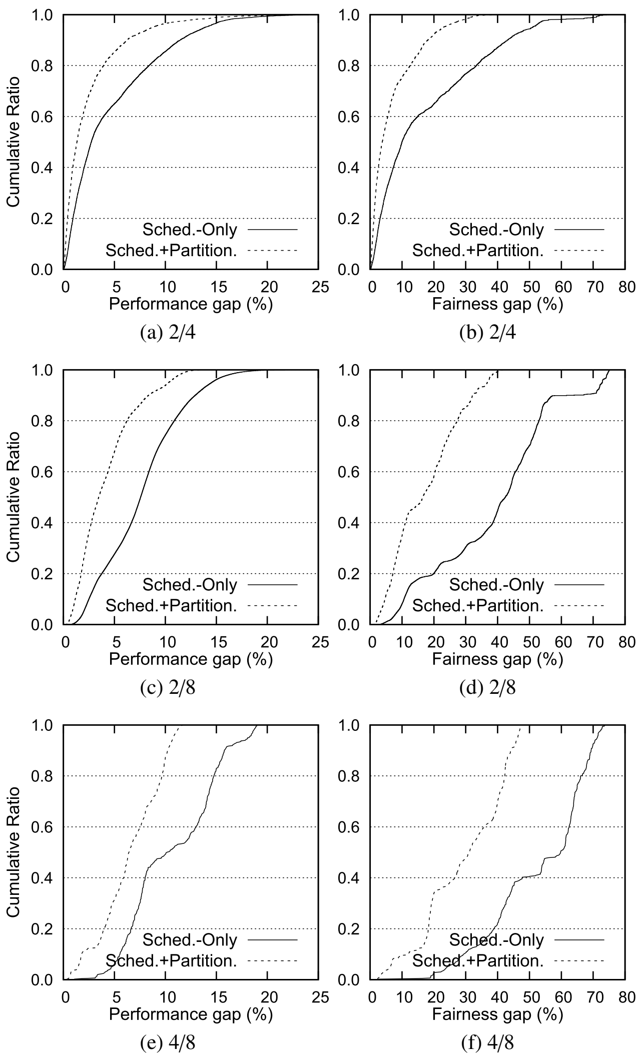


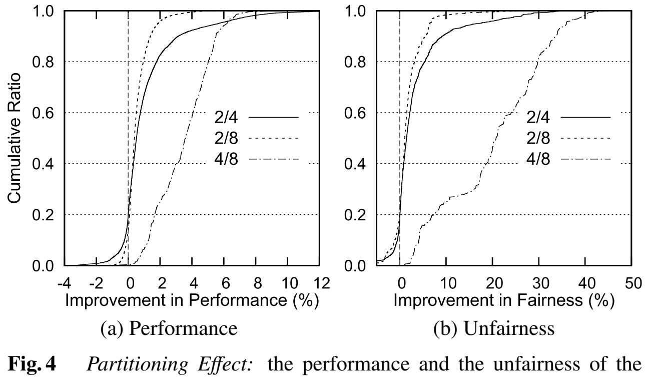
Fig. 3 *Scheduling Effect:* the performance and fairness improvement by the best scheduling over the worst scheduling with and without cache partitioning support.

To show the impact of good scheduling, Fig. 3 presents the performance and fairness improvement by the best scheduling over the worst scheduling, with or without cache partitioning. `Sched.-Only` curves represent the gap between the best and worst scheduling without cache partitioning and `Sched.+Partition.` curves represent the gap between the best and worst scheduling with cache partitioning. Scheduling has noticeable impacts on the overall speedups for systems both with and without partitioning. Without partitioning, the performance gap between the best and worst scheduling is more significant than with partitioning supports. The configuration without partitioning shows high gaps for the majority of mixes – with 35% of possible mixes having more than 5% performance gap between the best and worst scheduling for the 2/4 system. In the 2/8 and 4/8 system, the performance gap is much wider than that of the 2/4 as the mixes are more diverse in their demands on a cache. Partitioning reduces the gap for the majority of mixes – for the 2/4 system, only 14.6% of possible mixes having more than 5% performance gap. However, grouping of applications is still important for maximizing performance benefit of partitioning mechanism. Performance gap is up to 26% even with partitioning. The fairness improvements are generally higher than the performance improvements, since the fairness metric shows the difference between min and max performances. Fairness is an essential requirement of some computing environments like cloud computing. In the cloud computing, customer pay for the computing resources and require some fairness guarantees when multiple applications compete for the same resources.

Figure 4 is the improvement with the cache partitioning over the best scheduling. Cache partitioning improves performance further from what scheduler did in some cases, although the majority of mixes, 89% and 99.8% of all the mixes, has less than 3% extra performance gains with partitioning compared to the best scheduling in the 2/4 and 2/8 systems. With 2/8 systems, as the number of scheduling choices increases, we can achieve the ideal performance with smart scheduling without cache partitioning support. However, in the 4/8 system, the benefit of partitioning increases significantly, compared to the 2/4 and 2/8 systems. This indicates that although scheduling without any partitioning support can potentially mitigate a significant portion of the negative interference with shared caches, partitioning has some potential for further improvements in certain mixes.

The complete analysis of application combinations and scheduling choices leads to three observations.

- Contention cases are relatively infrequent, but their effects can be quite significant for certain cases. However, costly contention-mitigation mechanisms may not provide enough return for the cost.
- Scheduling can be quite effective for mitigating cache contentions, if there exist multiple shared caches.
- Partitioning complements scheduling, although the extra benefit of partitioning over the best scheduling is relatively small. Such a marginal improvement may not justify a full support for hardware partitioning. However, for certain cases, the extra gain from partitioning is significant, so recovering the gain with the least extra cost is important.



(a) Performance (b) Unfairness

Fig. 4 *Partitioning Effect:* the performance and the unfairness of the cache partitioning over the best scheduling without cache partitioning support.

4. Application Classification

In this section, we classify applications based on their cache miss rates, and explore the contention behaviors among several different combinations of application types sharing a cache. We use the simple metrics which can be measured in performance monitoring counters currently available, so that the classification does not require extra hardware support. We will use this classification method as a basis for our scheduling policy described in Sect. 5.

4.1 Applications Classification

We categorize applications into three groups according to their cache behavior – `C(cache-sensitive)`, `P(cpu-bound)` and `M(memory-bound)`. This classification with three types are commonly found in prior work [1], [5], [11]. However, we revised the classification method to use only simple metrics such as cache miss rates and misses per 1K instructions. Applications with high MPKI (Miss Per Killo Instructions) or miss rate are classified in to M group. Those applications tend to hurt the performance of co-located applications by polluting a shared cache with blocks which will not be reused. Applications with relatively low cache misses is categorized as P or C group. P group is comprised of applications which exhibit a very low miss ratio, thus not only do not cause cache contentions, but are not affected by cache contentions because they have a working set size that fits in the smaller levels of the cache. Applications which benefit from more cache capacity are classified to the C group and these applications are affected by cache contentions the most.

For the practicality and scalability of the conflict-mitigating approach, unlike the approaches by [11] or [7], our classification method requires neither the prior knowledge on applications from off-line profiling, nor extra hardware for on-line characterization. The classification is purely based on the miss rate information collectible from the performance monitoring counters. As the most of modern processors are equipped with the performance monitoring counter, our approach is more practical than using the stack distance counter. Also, using the performance monitoring counter allows online profiling during the application's run-time. Instead of the additional hardware, we use the miss ratio of the private L1 cache, as well as the shared cache for more accurate classification. The criteria of M/C/P classification we used is presented in the Table 3. M_{L1} and M_{L2} are cache miss ratio of $L1$ and $L2$, respectively.

Table 3 Criteria for the application classification.

Category	Criteria
P	M_{L1} < 0.01 or (M_{L1} < 0.02 and M_{L2} < 0.1)
M	$MPKI_{L2}$ > 50 or M_{L2} > 0.7 or M_{L1} > 0.2
C	*The rest fall into the group C*

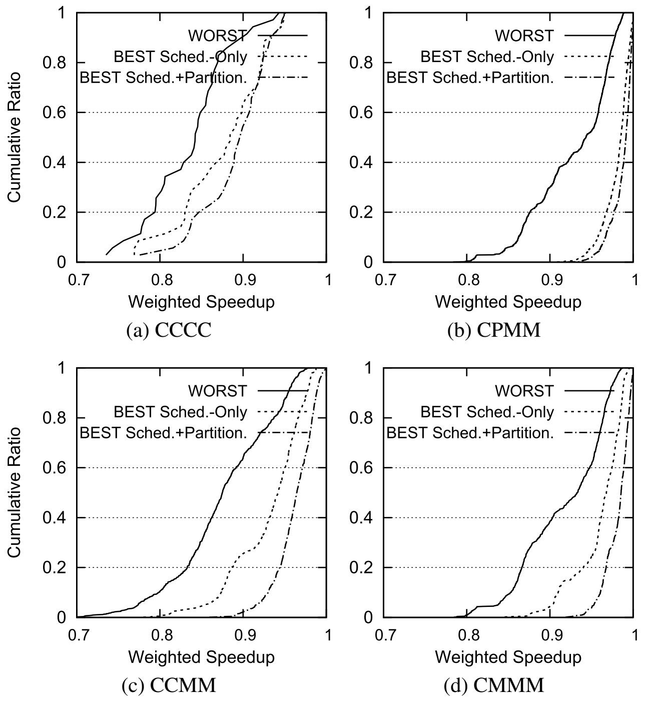


Fig. 5 The performance of the scheduling and the partitioning with various combinations of mixes: We present the worst case performance, the best performance with scheduling, and the best performance of the cache partitioning with appropriate scheduling.

$MPKI_{L2}$ is the number of $L2$ misses per killo instructions. For the accurate classification at runtime even when multiple applications content for the shared cache, we re-examine every M application which is co-located with other M applications and classify application as C if the application has small L1 miss ratio. That application might be misclassified as M group because of the high miss rate which is caused by the cache contentions.

4.2 The Effect of Scheduling and Partitioning on Combinations of Application Classes

In this section, we further investigate the contention problem with 4 representative combinations of application classes as defined in the previous section. We focus on the four combinations, since they represent most of the contention cases. In this section, we use only the 2/4 system for discussion, although the other configurations exhibit similar trends. Our proposed scheduling with LRU insertion control is based on the observations made in this section.

We categorize all the mixes of 4 cores according to the group that each application belong to. Then we present the result of CCCC, CPMM, CCMM, and CMMM mixes in the Fig. 5. Note that each character indicates the group that each application belongs to. For example, CPMM mixes are composed of one application of group C, one from group P, and two applications from group M. Figure 5 represents the weighted speedup of the worst case performance (`WORST`), the best case performance with scheduling without cache partitioning (`BEST Sched.-Only`), and the best case performance of partitioning with appropriate scheduling support (`BEST Sched.+Partition.`). In the rest of the paper, 'C application' denotes that an application from the C group.

CCCC: In Fig. 5 (a), mixes are composed of four C applications. Because all of the applications require more cache capacity, both scheduling and partitioning do not improve the performance effectively. In this case, even with the combination of scheduling and partitioning, the weighted speedup of more than half of the mixes is below the 0.9, meaning that the performance loss by contention is more than 10%, compared to the performance in single core runs. However, in this case, there may not be an effective solution other than increasing the cache capacity.

CPMM: For these mixes, scheduler eliminates almost all the cache contentions as shown in Fig. 5 (b). Without scheduling, 30% of mixes exhibit over than 10% performance loss due to contention. However, with scheduling support, None of mixes exhibit over than 10% performance loss. For those mixes, to eliminate cache contentions, scheduler allocates C application with the P one. C application does not suffer from the cache contentions and occupies almost all of the cache resource. The cache partitioning is not efficient for these mixes because the scheduling eliminates almost all negative interference among applications.

CCMM: Not only the scheduling but the cache partitioning is efficient for improving performance of these mixes. Scheduler isolates C application from the M application to reduce the cache contentions. In this case, the negative impact of cache contentions decreases, but the shared cache may be underutilized. With the cache partitioning support, scheduler co-locates the C and M application on a shared cache then the cache space is isolated so that the C application has the almost all the cache capacity. Without any contention management mechanisms, 64% of mixes experience over than 10% performance loss. With appropriate allocation of applications, portion of mixes suffering from contentions reduces to the 25.5%, and with the cache partitioning with appropriate scheduling, only 2% of mixes experience over than 10% performance loss.

CMMM: For these mixes, with any possible scheduling combinations, at least one C application has to be co-located with the M application which leads to the cache contentions. However, not every co-location of C and M application degrades performance significantly. As shown in Fig. 5 (d), scheduler reduces cache contentions quite effectively, if it is possible to find the M application which least contends with C applications. The partitioning improves performance further by isolating the cache space between the C and M applications, so that the C application occupies the almost all cache capacity. Though the scheduler reduces the cache contentions effectively by co-locating the less-interfering applications, it requires the accurate prediction of cache contentions based on the complex application profile like stack distance count. In our approach, for these mixes, we use a limited LRU insertion to reduce the complexity of the scheduling algorithm.

5. Approach

Based on the observations discussed in the previous sections, we propose an inexpensive approach based on scheduling with a simple LRU insertion mechanism, which is occasionally used for certain mixes with a high contention. The proposed approach differs from the prior partitioning or scheduling techniques in two ways. Firstly, unlike prior cache partitioning techniques, our approach does not require costly online or offline construction of stack distance profiles for applications. Instead, it uses only a simple classification method based on miss rates. Secondly, the approach mostly mitigates contentions by scheduling applications with the aforementioned approximate classification, but it also uses a HW support for LRU insertion. The selective use of LRU insertion allows the scheduler to recover the performance potential, which cannot be improved with scheduling-only schemes, or the performance loss caused by our approximate classification.

5.1 Selective Cache LRU Insertion

The proposed approach does not use a full HW cache partitioning support proposed by the prior partitioning approaches. Instead, we use a very simple mechanism of LRU insertion. In the conventional caches with a LRU replacement policy, a cache-block is always inserted to the MRU (Most Recently Used) position of the corresponding set. However, an option to insert a new cache-block to the LRU position can be added without any significant change in the current processor design. Some commercial processors already support such a mechanism, and expose it to the partial control of the system software [12]. Our approach requires only such a simple option to insert a new cache-block selectively to the LRU position, when the scheduler needs.

This LRU insertion support is completely different from the prior partitioning support. Unlike prior partitioning techniques, our approach uses a simple classification without any extra support to accurately measure the application cache behaviors as discussed in Sect. 2. Instead of allocating the appropriate capacity based on the accurate utility information, we focus on eliminating the cache contentions caused by steam-like cache accesses. The blocks belong to an application which generates stream-like accesses are inserted at the LRU position, instead of MRU position to minimize the residency time of such blocks. Our scheduler uses the LRU insertion support selectively only when the scheduling itself cannot handle the contention effectively.

5.2 Scheduler Design

The proposed scheduler groups applications based on their approximate classification, and selectively uses the LRU insertion for certain cases. The proposed scheduling algorithm is based on the contention behavior with different combinations of application classes as discussed in Sect. 4. Table 4 describes the scheduling policy. After the classification, scheduler allocates applications in the way of maximizing the overall cache utilization. The scheduler firstly distributes C applications as evenly as possible across caches, so that the applications can take advantage of a large portion of cache capacity.

Table 4 Scheduling policy of our approach.

1.	distribute all **C** applications across all **caches**
2.	co-locate all **P** applications with **C** applications
3.	co-locate all **M** applications with **C**,**P** applications
4.	for all the **M** applications co-located with **C** application, insert blocks of **M** applications in the LRU position

After allocating all the C applications, P and M applications are scheduled in turn. Firstly, P applications are co-located with C applications, as P applications do not require much of cache capacity, and allow C applications to use the shared capacity. Finally, M applications are distributed to all the caches. However, for M applications which are co-located with C applications, the LRU insertion is turned on for the M applications to mitigate cache conflicts caused by the M applications. When the LRU insertion is turned on, the cache-blocks brought in by the M application are inserted to the LRU position. If P or M applications are co-located with an M application, the LRU insertion is not used, since the M application does not affect the P or M applications.

6. Experimental Results

6.1 Performance Evaluation

Figure 6 depicts the performance of the proposed mechanism in terms of the weighted speedup and unfairness. Three figures in the left are the weighted speedup. `WORST` and `BEST` is the worst and the best performance of the possible combinations of scheduling and cache partitioning. `Sched.+LRU insert` plots the performance of our scheduler which is combined with selective LRU insertion. `WORST` is the performance with the worst scheduling without cache partitioning support and `BEST` is the performance with the best scheduling with the full cache partitioning support. `CRUISE` represents the performance of the prior full HW-based mechanism for partitioning and scheduling [7].

In our experiments, we classify applications with the cache miss behaviors measured every 200M instructions. During a 200M instruction period, cache miss behaviors for each core are measured, and the classification based on the measured cache misses is used to schedule applications for the next 200M instruction period. Figure 6 shows the performance for a 200M instruction period.

The performance of `Sched.+LRU insert` is close to that of the ideal performance. Therefore, the results show that instead of using the state of the art cache partitioning or scheduling mechanisms, our combined approach with a simple hardware extension is enough for the practical purposes. `Sched.+LRU insert` proposed in this paper shows a simi-

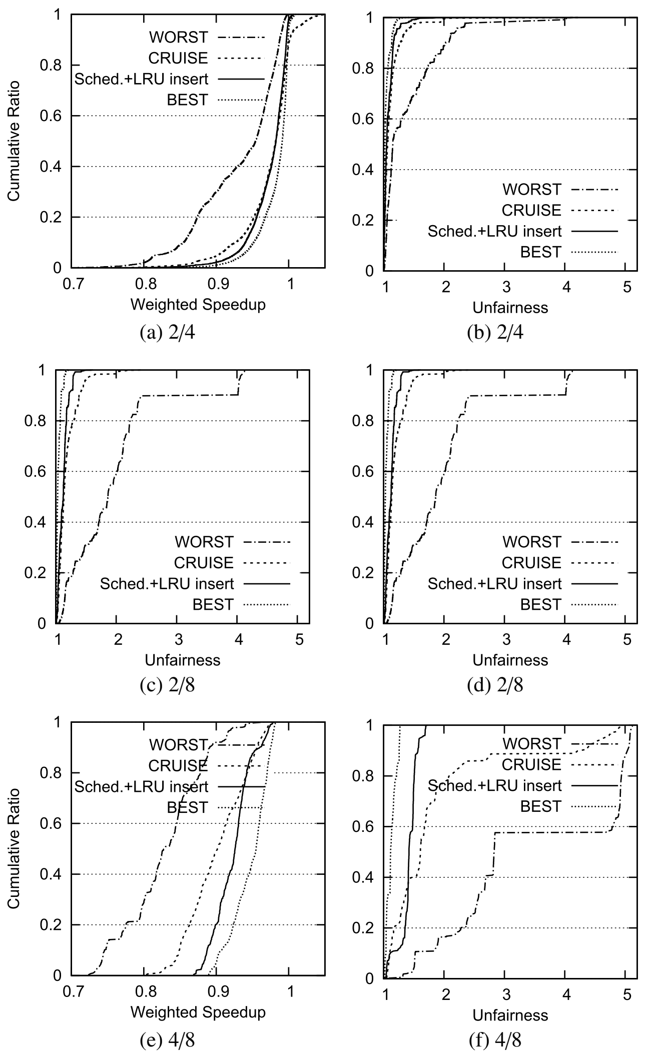


Fig. 6 The cumulative distribution of the *weighted speedup* and *unfairness* of the worst and the best performance of the possible combinations of scheduling and cache partitioning, CRUISE, and proposed approach. Note that the weighted speedup uses the single IPC under LRU replacement.

lar or slightly better performance than the full HW-oriented `CRUISE` mechanism.

As shown in the Fig. 6, the performance of the proposed one can be slightly better than `CRUISE` in some cases, because of the caching efficiency incurred by the set-sampling mechanism used in the CRUISE. For the CRUISE results, we use the ideal implementation of CRUISE which uses the static profiling of applications to decouple the performance of CRUISE and the overhead of logic for the application classification. A realistic implementation of CRUISE requires a dynamic profiling mechanism with cache set-sampling which will be explained in Sect. 7. CRUISE with dynamic application classification with the set-sampling, incurs an additional 2% performance loss, compared to the ideal result shown in the figure.

With 4/8 configuration, the proposed mechanism outperforms the CRUISE because we represent the performance of the selected mixes for the 4/8 configuration. And the set-sampling overhead is increased as the number of cores sharing a cache increases. We will discuss the HW cost and set-sampling mechanism of CRUISE in Sect. 7.

Table 5 The portion of mixes with respect to the number of threads using the LRU insertion. Note that there are 4 threads for 2/4 configuration, and 8 threads for 2/8 or 4/8 configuration.

conf.	# of threads				
	0	1	2	3	≥ 4
2/4	62.8%	27.7%	9.5%	0%	0%
2/8	53.8%	21.5%	16.3%	7.2%	1.2%
4/8	4.3%	3.4%	14.2%	42.4%	35.7%

The performance gap between our approach and the ideal case slightly increases with the 2/8 and 4/8 configurations. For the 2/8 configuration, the number of possible scheduling choices increases as the number of cores increases and it requires a smart scheduling when the number of core becomes large. For the 4/8 configuration, the number of cores sharing a cache increases and it causes more cache contentions among applications. In this case, the approximate approach of the cache partitioning may not reach the ideal performance. However, the performance of our work is close enough to the ideal case in these cases, and the performance difference is marginal.

As in Fig. 6 (b), 6 (d), and 6 (f), fairness is improved enormously. The fairness support from our approach is almost identical to the ideal run in 2/4 and 2/8, and slightly lower than that of the ideal in 4/8.

6.2 LRU Insertion Cases

In the proposed approach, the cache LRU insertion is selectively enabled only when an M-type application is co-located with a C-type application. Table 5 represents the portion of mixes with respect to the number of threads using the LRU insertion. For example, for the 2/4 configurations, 62.8% of mixes does not the LRU insertion for any of threads. For 4/8 configuration, LRU insertion is more frequently used as the number of threads competing a shared cache increases, still less than $\frac{2}{5}$ of mixes use LRU insertion for more than 4 threads of those mixes. As the number of cores sharing a cache increases, the chance to co-locate M and C type applications also increases, enabling the LRU insertion mechanism.

7. Complexity Comparison with CRUISE

In this section, we compare the proposed simple SW-based approach against a state of the art full HW-oriented partitioning and scheduling mechanism, `CRUISE` [7]. Compared to the full HW mechanism, the proposed approach does not require any extra hardware support except for a simple LRU insertion mechanism. Even with such a low additional HW complexity, the proposed approach exhibits performance similar to or better than the full HW approach, as shown in the previous section. In this section, we elaborate the implementation cost and complexity of two extra mechanisms necessary for the prior HW-based CRUISE mechanism.

7.1 Cache Partitioning and Replacement in CRUISE

CRUISE requires a relatively complicated HW-based partitioning mechanism to improve cache utility. The partitioning mechanism in CRUISE uses the DRRIP [1] cache replacement policy. DRRIP is a modified NRU (Not Recently Used) cache replacement to mitigate the cache contentions among applications with different cache utility. For the n-way associative cache, DRRIP requires the hardware cost of *2n* while the baseline NRU replacement policy requires the hardware cost of *n*.

DRRIP selectively uses the two different cache replacement policies, scan-resistant SRRIP and thrash-resistant BRRIP. SRRIP attempts to preserve the active working set of an application after scan-like cache accesses. BRRIP prevents thrashing blocks from evicting the other cache blocks. To keep track of which policy is better for an application, DRRIP uses a set-sampling mechanism. It dedicates 32 sets for each policy, and makes those sets follow only either SRRIP or BRRIP. A 10-bit counter counts the number of cache misses for those sets and DRRIP uses the replacement policy with low cache misses for the rest of cache sets. Compared to separate tags to measure cache utility on-line [3], the set-sampling mechanism require a relatively low hardware area because it uses a subset of sets as a sampling set. However, it imposes the performance overhead, in addition to the increased complexity of tag lookups and updates, because the dedicated sets may have to use an inferior performing replacement policy. For example, for the 16-way 4 MB cache shared by 8 cores, 512 among 4096 sets are dedicated to follow a specified replacement, even though it is not the best performing one for the sets.

Our approach uses only a simple HW mechanism to allow a new cache block to be inserted in the LRU (Least Recently Used) position of the corresponding set. The approach needs neither a set-sampling mechanism nor separate tags to support complicated partitioning or dual replacement policies used by the prior work.

7.2 Application Classification for CRUISE

In addition to the complexity and inefficient use of a subset of cache sets for DRRIP partitioning, CRUISE assigns certain sets for each core to identify the caching behavior of the core. To classify applications, CRUISE uses the RICE (Runtime Isolated Cache Estimator) mechanism to estimate the cache utility of an application in isolation at runtime. For every application, RICE dedicates 32 sets for each core as sampling sets, and forces the other cores to bypass the cache sets. Based on cache misses in the sample sets, it estimates the caching behavior of the core, if it were to have solo access to the cache. In addition, RICE dedicates other 32 sets and forces other cores to use only the half of the capacity to estimate the caching behavior of an application with the half of the cache. Based on the number of cache misses from the two types of sampling sets for each core, CRUISE classifies applications according to their cache utility and schedules them to minimize the contention. As in the DRRIP, such a set-sampling may cause the performance degradation in addition to the increased hardware complexity to embed such sampling sets with different policies. As presented in the CRUISE paper and our own evaluation results, performance degradation due to RICE alone is about 2% on average, due to the inefficient use of some sets.

In our approach, we do not use any extra hardware to classify applications. Instead, we use the performance monitoring counter that almost all the modern processors already have. Existing software can utilize our application classification and scheduling policy by dynamically profiling the application's cache behavior through the performance monitoring counters.

8. Conclusions

This paper revisited the contention problem in shared caches and proposed a new approach combining scheduling with a simple LRU insertion policy. The paper first analyzed the contention behaviors with an exhaustive analysis of all possible mixes from a given benchmark suite. Based on the analysis, this study introduced a classification-based scheduling which occasionally uses the LRU insertion support to reduce contentions unresolved by scheduling. Unlike prior partitioning schemes, the approach requires negligible changes in the current hardware designs. Compared to the prior scheduling-only schemes, this study addresses the contentions with the LRU insertion support, which cannot be handled effectively by scheduling.

This new direction, managing contention only if it is necessary, reduces the design and implementation costs, and suggests a practical way of contention management which is applicable on actual platforms. We will examine the proposed mechanism on a real platform to verify the practicality and scalability of our work and we expect that it also enables us to observe the effect of other resource factors such as bus bandwidth or memory controllers.

Acknowledgements

This research was supported by the SW Computing R&D Program of KEIT (2011-10041313, UX-oriented Mobile SW Platform) funded by the Ministry of Knowledge Economy

References

[1] A. Jaleel, K.B. Theobald, S.C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," Proc. 37th Annual International Symposium on Computer Architecture (ISCA), 2010.

[2] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," Proc. International Symposium on High Performance Computer Architecture (HPCA),

2008.

[3] M.K. Qureshi and Y.N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” Proc. 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2006.

[4] Y. Xie and G.H. Loh, “PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches,” Proc. 36th Annual International Symposium on Computer Architecture (ISCA), 2009.

[5] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “Addressing shared resource contention in multicore processors via scheduling,” Proc. 13th International Conf. Architectural support for programming languages and operating systems (ASPLOS), 2010.

[6] J. Mars, L. Tang, R. Hundt, K. Skadron, and M.L. Soffa, “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations,” Proc. 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.

[7] A. Jaleel, H.H. Najaf-abadi, S. Subramaniam, S.C. Steely, and J. Emer, “CRUISE: Cache replacement and utility-aware scheduling,” Proc. 17th International Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[8] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt, “The m5 simulator: Modeling networked systems,” ACM SIGARCH Computer Architecture News, 2011.

[9] G.E. Suh, L. Rudolph, and S. Devadas, “Dynamic partitioning of shared cache memory,” J. Supercomputing, vol.28, pp.7–26, 2004.

[10] E. Ebrahimi, C.J. Lee, O. Mutlu, and Y.N. Patt, “Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems,” SIGPLAN Notices, vol.45, pp.335–346, March 2010.

[11] Y. Xie and G.H. Loh, “Dynamic classification of program memory behaviors in cmps,” Proc. 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2008.

[12] BIOS and kernel developer’s guild for AMD family 15h processors, March 2012.

Eunji Pak is a Ph.D. candidate of Computer Science at Korea Advanced Institute of Science and Technology (KAIST). Her research interests are in computer architecture and operating systems. She received the B.S. and M.S. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2002 and 2004, respectively.

Sang-Hoon Kim is a Ph.D. candidate of Computer Science at Korea Advanced Institute of Science and Technology (KAIST). His research interests are in operating systems and storage systems and flash memory technology. He received the BS degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 2002. He worked for “FnBC” as a software engineer in 2002 and worked for “Inzen” as a software engineer from 2003 to 2005.

Jaehyuk Huh is an Associate Professor of Computer Science at Korea Advanced Institute of Science and Technology (KAIST). His research interests are in computer architecture, parallel computing, virtualization and system security. He received a BS in computer science from Seoul National University, and an MS and a PhD in computer science from the University of Texas at Austin.

Seungryoul Maeng is a Professor of Computer Science at Korea Advanced Institute of Science and Technology (KAIST) since 1984. His research interests are in computer architecture, parallel computing, and system security. He received the B.S. degree in Electronics Engineering from Seoul National University, Korea, in 1977, and the M.S. and Ph.D. degrees in Computer Science from KAIST in 1979 and 1984, respectively. From 1988 to 1989, he was with the University of Pennsylvania as a visiting scholar.