A CACHE-AWARE THREAD SCHEDULING POLICY FOR MULTI-CORE PROCESSORS

Masayuki Sato ∗ Isao Kotera ∗ Ryusuke Egawa ∗† Hiroyuki Takizawa ∗ Hiroaki Kobayashi ∗†

∗ Graduate School of Information Sciences, Tohoku University, † Cyberscience Center, Tohoku University
Aramaki-Aza-Aoba, Aoba-ku, Sendai 980-8578, Japan
email: {masayuki, isao}@sc.isc.tohoku.ac.jp, {egawa, tacky, koba}@isc.tohoku.ac.jp

ABSTRACT
A modern high-performance multi-core processor has large shared cache memories. However, simultaneously running threads do not always require the entire capacities of the shared caches. Besides, some threads cause severe performance degradation by inter-thread cache conflicts and shortage of capacity on the shared cache. To achieve high performance processing on multi-core processors, effective usage of shared cache memories plays important role.

In this paper, we propose a cache-aware thread scheduling policy for multi-core processors with multiple shared cache memories. The total processor performance becomes more sensitive to the cache capacity shortage, as larger caches are requested by the threads sharing one cache. The proposed policy can prevent multiple threads requesting a large cache capacity from sharing one cache. As a result, the policy can prevent inter-thread resource conflicts and hence severe performance degradation. Experimental results clearly demonstrate that the policy assists the cache partitioning mechanisms and avoids unfair performance degradation among threads. Thread scheduling based on the proposed policy can improve the performance by up to 10% and an average of 5% compared with thread scheduling without the proposed policy.

KEY WORDS
Parallel Computing Systems, Multi-Core Processors, Thread Scheduling, Dynamic Cache Partitioning

1 Introduction
In the last four decades, a continuous increase in the number of transistors on a chip brings a large number of functional units into a processor to simultaneously execute more instructions exploited by instruction-level parallelism (ILP). Nowadays, however, this approach hardly leads to a significant performance improvement due to the limitation of the number of instructions that can be simultaneously executed [1].

Recently, thread-level parallelism (TLP), which is coarser-grain parallelism than ILP, has become to attract attention. Taking advantage of TLP, a program is decomposed into multiple threads, and a microprocessor can simultaneously execute those threads. Chip multiprocessors (CMP) [2], simultaneous multithreading [3], and these mixtures have been proposed to realize the simultaneous execution of threads. The number of threads executed simultaneously on a microprocessor will increase, and hence performance improvements of future microprocessors will rely on TLP.

When simultaneously executing multiple threads spawned from different applications, a problem of such microprocessors is performance degradation due to shared resource conflicts among threads. As a result of resource conflicts, the execution time of a multi-core processor may become longer than that of a single-core processor, even though the former can simultaneously execute multiple threads while the latter executes those threads one by one. Therefore, it is important for a multi-core processor to prevent inter-thread resource conflicts.

In this paper, we focus on cache memories and propose a cache-aware thread scheduling policy to assist dynamic cache partitioning. Dynamic cache partitioning mechanisms [4, 5, 6, 7, 8] have been proposed to eliminate inter-thread cache conflicts on shared cache memories. A certain amount of performance improvement is obtained by such mechanisms. However, since dynamic cache partitioning limits the cache capacity allocated to each thread, performance may severely degrade if the total capacity requested by all the threads exceeds the cache capacity. The proposed policy schedules threads based on estimation of the cache capacity requested by each thread, so as not to degrade the performance due to the lack of cache capacity.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. In Section 3, we discuss performance issues about dynamic cache partitioning and propose a cache-aware thread scheduling policy. Section 4 shows experimental results and discussion for evaluation of our proposal. Finally, Section 5 concludes this paper.

2 Related Work
In multithreading, cache conflicts among threads occur if the data of one thread in a cache memory are replaced by the data of the other thread. If the former thread again accesses the data that have already been replaced by the latter thread, the cache access results in a cache miss that does not occur in a single-threaded environment. Such conflicts
are called inter-thread kickouts [9]. Because of this, replacement for one thread may increase the execution time of another thread.

Dynamic cache partitioning has been proposed to prevent inter-thread kickouts. To reduce the conflicts, dynamic cache partitioning divides a cache memory into multiple parts, and each part is dynamically allocated to only one thread. Suh et al. described a cache partitioning algorithm based on the marginal-gain [4], and a low-overhead control scheme. However, the above algorithm cannot take into account the fairness among threads sharing a cache. Kim et al. [6] defined the fairness as equality of performance degradation among threads. Their partitioning algorithm decides allocation of partitioned cache capacities by minimizing difference in the numbers of miss accesses among threads or difference in miss rates among threads. Chang et al. [7] introduced multiple time-sharing partitions to improve throughput and fairness while maintaining QoS.

However, if cache capacities requested by threads sharing a cache are too large, dynamic cache partitioning degrade the performance. Suppose that two threads are running using a shared cache. Then, if the sum of cache capacities requested by the two threads exceeds the capacity of the shared cache, dynamic cache partitioning cannot simultaneously meet both of their requests. In this situation, severe performance degradation may occur on simultaneously executing threads. Although dynamic cache partitioning is effective to avoid the problem about inter-thread kickouts, it cannot avoid the one about capacity.

In this paper, dynamic cache partitioning with effective thread scheduling is considered not only to avoid inter-thread kickouts, but also to avoid performance degradation caused by shortage of cache capacity.

3 A Cache-Aware Thread Scheduling Policy

3.1 Performance degradation in dynamic cache partitioning

Dynamic cache partitioning is effective to eliminate inter-thread kickouts. However, it cannot resolve the problem induced by the shortage of cache capacity.

Modern microprocessors rely on large cache memories to hide long memory latency. However, all threads do not always need the entire capacity of these cache memories. The cache capacity necessary for a thread to maintain the performance depends on its working set. Figure 1 shows the utility graph [5] of the gzip benchmark from SPEC CPU2000 [10] with 1MB, 32-way L2 cache memory. The utility graph shows the relationship between performance of the normalized IPC and cache capacity. The capacity is resized by a way-allocation mechanism in units of cache ways. Figure 1 indicates that approximately 16 ways (0.5MB) are required for the thread to achieve the maximum performance, and 95% of the maximum performance can be achieved even with 8 ways (0.25MB). The number of ways required for a thread to achieve a certain performance varies depending on the application. Figure 2 shows the number of ways required for maintaining 95% of maximum performance from SPEC CPU2000.

If a dynamic cache partitioning mechanism can satisfy such the number of requested ways of all threads sharing a cache, it would not degrade the performance. If the mechanism cannot satisfy any requirements, it will severely degrade the performance. Figure 3 shows relationship between the total number of ways requested by two threads to maintain 95% of the maximum performance, and the average normalized IPC of threads sharing one cache. In this figure, each point corresponds to a pair of two threads. This indicates that the performance becomes more sensitive to resource conflicts as the sum of numbers of requested ways increases. These results show that the performance degradation strongly depends on the number of ways requested by the threads. Consequently, the capacity requested by each thread is a key to estimate the performance degradation.

3.2 Motivation

Thread scheduling plays an important role to extract the potential of multi-core processors with shared caches. When
two threads with high requirements share a cache, the performances are degraded due to shortage of its cache capacity. On the other hand, two threads with low requirements share a cache, it will be waste of the cache capacity. Figure 4 shows an example of such scheduling. In Figure 4, both Thread0 and Thread1 that have high requirements are scheduled to Cache0, resulting in cache capacity shortage. On the other hand, Thread2 and Thread3 that have low requirements are scheduled into Cache1. Although Thread2 and Thread3 may achieve the maximum performance, a part of cache resources are unused. If we can use the unused cache resources for Thread0 or Thread1, the processor can resolve the resource shortage.

3.3 Proposal of a thread scheduling policy

Based on the above discussions, we propose a cache-aware scheduling policy for multi-core processors with multiple caches shared by multiple threads. Our proposal consists of two stages; one is profiling of the number of ways requested by a thread, and the other is thread scheduling based on the results of the profiling.

3.3.1 Profiling

In practical, it is costly to use the utility graph to assess the number of ways required to maintain performance. Hence, we use the way-adaptable cache mechanism [11] for the assessment. This mechanism estimates the minimum number of ways by locality assessment of cache accesses using stack distance profiling [12]. Therefore, we can use the mechanism to obtain the cache requirement of a thread, i.e. the number of requested ways. The cache requirement of a thread is defined by the time-averaged number of cache ways assigned to the thread by the mechanism:

$$CR_{thread} = \frac{\sum_{t=t_0}^{t_1} W(t)}{t_1 - t_0},$$ (1)

where $W(t)$ is the number of cache ways assigned to the thread at time $t$ ($t_0 < t < t_1$).

3.3.2 Scheduling

The proposed scheduling policy aims to avoid unfairly degrading normalized IPCs among threads due to resource conflicts. In our scheduling policy, as shown in Figure 5, a thread with a high cache requirement is coupled with that with a low requirement. As a result, the policy can prevent underutilized cache resources such as Cache1 in Figure 4. If Thread0 and Thread3 share a cache, a dynamic cache partitioning mechanism can allocate more cache capacity to Thread0 compared to the worst case assignment, in which Thread0 and Thread1 share a cache. Similarly, Thread1 can also use more capacity if it runs with Thread2. Accordingly, the proposed scheduling policy can assist dynamic cache partitioning to prevent the resource conflicts while avoiding the cache capacity shortage.

To achieve such scheduling, a scheduling algorithm is required to equalize the sum of cache requirements among shared caches. In this paper, we propose an algorithm assuming a $k$-core microprocessor with $k/2$ shared caches and $k$-thread simultaneous execution. Figure 6 shows the overview of the algorithm. Let $T$ and $G$ be a list of $k$
threads and a list of $k/2$ groups. Here, each group corresponds to a shared cache; threads in one group are assigned so as to share the corresponding cache. Then, in Step 1 of the algorithm, threads in $T$ are sorted according to their cache requirements in descending order. In Figure 6, $CR_i$ denotes the cache requirement of thread $i$. In Step 2, each of top $k/2$ threads in $T$ is allocated into one of groups in $G$. As a result of Step 2, the $i$-th thread in $T$ is allocated to the $i$-th group, and then top $k/2$ threads in $T$ are removed from $T$. In Step 3, groups in $G$ are sorted according to the sum of cache requirements of the allocated threads in ascending order. In Figure 6, $SCR_j$ is the sum of cache requirements of threads allocated to group $j$. This step is required to allow a larger $CR_i$ to be summed to a smaller $SCR_j$ in Step 2 of the next iteration. Finally, the thread scheduling is finished if $T$ is empty. Otherwise, go to Step 2.

4 Performance Evaluation

4.1 Evaluation Methodology

In the section, we evaluate the effectiveness of the proposed scheduling algorithm using the M5 simulator [13]. Detailed parameters of components are shown in Table 1. The processor configuration used in the simulation is as follows.

- Four cores, each has a private L1 cache.
- Two L2 caches, each shared by two cores.
- Each L2 is highly-associative with the LRU policy, and employs the dynamic cache partitioning mechanism[8].

In the evaluations, six benchmark programs are selected from the SPEC CPU2000 benchmark suit [10] based on the characteristics of cache accesses. Then, we select four-benchmark combinations from six benchmarks, and simultaneously execute them as four independent threads in the experiments. By this selection, benchmark combinations and scheduling results are representative of all benchmark combinations.

In the selection, all the benchmarks are categorized into three classes based on their utility graphs. As a result of the classification, the benchmark programs are classified into high-utility, saturating-utility, and low-utility. From each class, we selected two representative benchmarks. All the six benchmarks are shown in Table 2. Figure 7 shows the utility graphs of the selected benchmarks. In this figure, the performance is evaluated by the ratio of IPC achieved with allocated ways to that with all ways. VprPlace and Twolf are categorized into high-utility benchmarks. High-utility benchmarks increase their performances gradually as the number of ways increases. Mesa and Equake are saturating-utility benchmarks. Their performances are drastically improved by an increase in the number of ways when the number of ways is small. However, their performances are not improved when the number of ways is large. Wupwise and Apple are low-utility benchmarks. The performance of low-utility benchmarks are not improved even if the number of ways increases.

From the above observations, it can be expected that high-, saturating-, and low-utility benchmarks have high, middle, and low cache requirements, respectively. We can see such trends in the cache requirements ($CR$) in Table 2. This indicates that the profiling method used in the proposed scheduling mechanism can appropriately estimate the cache requirements of these benchmarks.

With these benchmarks, we generate six benchmark combinations for the experiments. Although we can consider 15 combinations from six benchmarks ($6C_4 = 15$), we reduce the number of benchmark combinations by removing the combinations having the same characteristics. For example, both of [VprPlace, Equake, Wupwise, Apple] and [Twolf, Mesa, Wupwise, Apple] consist of one high-utility, one saturating-utility, and two low-utility bench-

<table>
<thead>
<tr>
<th>Table 1. Simulation parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch width</td>
</tr>
<tr>
<td>decode width</td>
</tr>
<tr>
<td>issue width</td>
</tr>
<tr>
<td>commit width</td>
</tr>
<tr>
<td>inst. queue</td>
</tr>
<tr>
<td>L1 I-cache</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>L1 D-cache</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>main memory</td>
</tr>
</tbody>
</table>
Table 2. Benchmark Programs

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>util</th>
<th>CR</th>
</tr>
</thead>
<tbody>
<tr>
<td>VprPlace</td>
<td>FPGA Circuit Placement</td>
<td>high</td>
<td>32</td>
</tr>
<tr>
<td>Twolf</td>
<td>Place and Route Simulator</td>
<td>high</td>
<td>32</td>
</tr>
<tr>
<td>Equake</td>
<td>Wave Propagation Simulation</td>
<td>sat</td>
<td>18</td>
</tr>
<tr>
<td>Mesa</td>
<td>3-D Graphics Library</td>
<td>sat</td>
<td>11</td>
</tr>
<tr>
<td>Wupwise</td>
<td>Quantum Chromodynamics</td>
<td>low</td>
<td>9</td>
</tr>
<tr>
<td>Applu</td>
<td>Partial Differential Equation</td>
<td>low</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 3. Benchmark combinations by thread characteristics

<table>
<thead>
<tr>
<th>Label</th>
<th>characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>HHSS</td>
<td>high high sat sat</td>
</tr>
<tr>
<td>HHSI</td>
<td>high high sat low</td>
</tr>
<tr>
<td>HHLL</td>
<td>high high low low</td>
</tr>
<tr>
<td>HSSL</td>
<td>high sat sat low</td>
</tr>
<tr>
<td>HSLL</td>
<td>high sat low low</td>
</tr>
<tr>
<td>SSLL</td>
<td>sat sat low low</td>
</tr>
</tbody>
</table>

Figure 7. Utility graphs

Figure 8. Performance of each scheduling policy

The IPC of the \(i\)-th thread running with other threads. There are two reasons for using this metric. One reason is that, using the metric, the performance of a thread running alone is compared to that of a thread running with other threads. This comparison can verify that the proposed policy can prevent performance degradation caused by resource conflicts among threads. The other reason is that the metric can assess the fairness between high and low requirement threads, even though the arithmetic mean of normalized IPCs cannot [7].

4.2 Experimental Results and Discussion

Figure 8 shows the performance of each combination. In this figure, bars indicate the performances of the proposed scheduling and the worst case scheduling. The proposed policy can improve the performance by 4.5%. Our proposed policy is effective to prevent remarkable performance degradation due to the inadequate combination of threads sharing a cache that has the dynamic cache partitioning mechanism.

The proposed policy achieves the largest performance improvement of 9.8% compared to the worst case in the HHLL combination. This combination consists of threads with two highest and two lowest cache requirements as shown in Table 2. In the worst case scheduling for this combination, there is a severe performance degradation because two threads with highest cache requirements are assigned to one cache. In addition, two threads with the lowest cache requirements are assigned to the other cache.
This leads to waste of its capacity. On the other hand, our scheduling policy can effectively prevent this unfair situation by allowing two high requirement threads to use different caches. For the SSLL combination, the proposed policy does not significantly improve the performances of threads. This is because the dynamic cache partitioning mechanism can provide a sufficient cache capacity to every thread, and therefore each scheduling policy can maintain those thread performances.

These results indicate that the proposed policy can always work better than or comparable to the worst case policy. Therefore, the proposed policy is significantly effective to prevent dynamic cache partitioning from degrading thread performances due to cache capacity shortage.

5 Conclusions

In this paper, we have proposed a cache-aware thread scheduling policy for multi-core processors with multiple caches. This scheduling policy prevents threads with high cache requirements from sharing one cache and assists dynamic cache partitioning to supply a sufficient cache capacity to every thread. Experimental results demonstrate that thread scheduling with the proposed policy can improve the performance by up to about 10% and an average of 5% compared with thread scheduling with the worst case assignment of threads.

In our future work, we will develop an on-the-fly assessment mechanism for the cache requirements of threads because the profiling cost assumed in this paper is still high. This is important to show the feasibility of our proposed policy in dynamic scheduling and migration of threads. We will also extend our policy to deal with dynamically changing degree of multithreading. Finally, we will continuously carry out more detail evaluation about various applications, microprocessor configurations.

6 Acknowledgments

The authors would like to thank the anonymous reviewers for constructive comments, and also thank Yusuke Funaya and Gao Ye for valuable discussions.

This research was partially supported by Grant-in-Aid for Scientific Research (B), the Ministry of Education, Culture, Sports, Science and Technology, No. 18300011.

References


