A Capacity-Aware Thread Scheduling Method Combined with Cache Partitioning to Reduce Inter-Thread Cache Conflicts

Masayuki SATO††††, Nonmember, Ryusuke EGAWA††††, Hiroyuki TAKIZAWA††††,†††† Members, and Hiroaki KOBAYASHI††††,†††† Members

SUMMARY Chip multiprocessors (CMPs) improve performance by simultaneously executing multiple threads using integrated multiple cores. However, since these cores commonly share one cache, inter-thread cache conflicts often limit the performance improvement by multi-threading. This paper focuses on two causes of inter-thread cache conflicts. In shared caches of CMPs, cached data fetched by one thread are frequently evicted by another thread. Such an eviction, called inter-thread kickout (ITKO), is one of the major causes of inter-thread cache conflicts. The other cause is capacity shortage that occurs when one cache is shared by threads demanding large cache capacities. If the total capacity demanded by the threads exceeds the actual cache capacity, the threads compete to use the limited cache capacity, resulting in capacity shortage. To address inter-thread cache conflicts, we must take into account both ITKOs and capacity shortage. Therefore, this paper proposes a capacity-aware thread scheduling method combined with cache partitioning. In the proposed method, inter-thread cache conflicts due to ITKOs and capacity shortage are decreased by cache partitioning and thread scheduling, respectively. The proposed scheduling method estimates the capacity demand of each thread with an estimation method used in the cache partitioning mechanism. Based on the estimation used for cache partitioning, the thread scheduler decides thread combinations sharing one cache so as to avoid capacity shortage. Evaluation results suggest that the proposed method can improve overall performance by up to 8.1%, and the performance of individual threads by up to 12%. The results also show that both cache partitioning and thread scheduling are indispensable to avoid both ITKOs and capacity shortage simultaneously. Accordingly, the proposed method can significantly reduce the inter-thread cache conflicts and hence improve performance.

key words: chip multiprocessors, shared caches, inter-thread cache conflicts, thread scheduling, cache partitioning

1. Introduction

Microprocessors have profited from the advance in CMOS process technology, growth in the number of integrated transistors and an increase in the clock frequency. However, in recent years, it has become difficult to utilize the integrated transistors by a single execution context due to lack of instruction-level parallelism (ILP), and further increase the clock frequency due to the heating problem. Hence, chip multiprocessors (CMPs) have become a major architecture for microprocessors. A single CMP includes multiple cores while they are on a single chip. These cores simultaneously execute multiple contexts as threads. As a result, CMPs can get a higher performance than single-core processors without exploiting ILP and without increasing the clock frequency.

To flexibly use hardware resources in CMPs, multiple cores often share some resources. In the case where one thread requires a large part of the shared resources while the other threads do not, CMPs with shared resources can further improve the performance compared with CMPs whose cores do not share resources. On the other hand, resources sharing often causes competition for limited resources among threads. As a result, some threads may not be able to obtain sufficient resources, resulting in severe performance degradation. Especially, competition in shared caches, the so-called inter-thread cache conflicts is a serious problem [1], [2]. Modern CMPs have large last-level shared caches to hide a long access latency to main memories. Under this situation, cache misses due to inter-thread cache conflicts have a large negative impact on performances.

There are several approaches to the above problem. Cache partitioning mechanisms [3]–[5] divide a cache into several parts and allocate them to threads exclusively. These parts are dynamically resized so as to satisfy demands for capacity of the threads. Therefore, the mechanisms can reduce inter-thread cache conflicts without losing the advantage of flexible resource usage among threads. Thread scheduling methods against inter-thread cache conflicts have also been discussed [6]–[10]. These methods decide combinations of threads, which share one cache, because performance degradation by inter-thread cache conflicts varies depending on the combinations.

Through discussions in this paper, we clearly classify these conflicts into two types, inter-thread kickouts (ITKOs) [11] and capacity shortage. ITKOs are replacements of data, in which data fetched by one thread are evicted by another thread in shared caches. Capacity shortage occurs when the total capacity demanded by the threads is larger than the capacity of the shared cache. To address these two types of inter-thread cache conflicts at the same time for modern CMPs with multiple shared caches [12], [13], the approach of combining cache partitioning and thread scheduling can work better than the individual approaches. The cache partitioning mechanisms can avoid ITKOs by allocating divided parts to threads exclusively. To
alleviate capacity shortage, the thread combinations sharing one cache can be changed by a thread scheduler in an operating system, which manages an assignment of threads to cores, so that the total capacity demanded by the threads in one cache does not become much larger than the cache capacity. Even if the thread combinations are changed, the cache partitioning mechanism flexibly resizes allocated parts to match the demands for capacity of the threads.

To this end, this paper proposes a capacity-aware thread scheduling method combined with cache partitioning. The cache partitioning mechanisms are applied to the shared caches in a CMP. These mechanisms also provide information of the necessary cache capacity of each thread with an estimation method used in the cache partitioning mechanism. Based on the estimation, the thread scheduler in the operating system decides thread combinations sharing one cache so that threads with high demands for capacity do not share one cache. As a result, the proposed method can reduce both types of inter-thread cache conflicts at the same time. The evaluation results show that the proposed method improves the overall performance and the performance of the threads with high demands for capacity. It is also demonstrated that considering both ITKOs and capacity shortage by the proposed method achieves a larger improvement than considering only either ITKOs or capacity shortage.

The rest of this paper is organized as follows. Section 2 focuses on the details of the inter-thread cache conflicts. Section 3 proposes a capacity-aware thread scheduling method combined with cache partitioning. In Sect. 4, the effects of the proposed method are discussed thorough the evaluation results. Finally, Sect. 5 concludes this paper.

2. Inter-Thread Cache Conflicts

Inter-thread cache conflicts have a harmful effect on the performance of CMPs. Hily et al. described that ignoring the effect of inter-thread cache conflicts causes over-estimation of the performance [1]. Thekkath et al. investigated the effectiveness of supporting multiple hardware contexts for microprocessors, and concluded that the impact of inter-thread cache conflicts becomes larger as the number of threads increases [2].

The purpose of this work is to reduce inter-thread cache conflicts. Therefore, in this section, we briefly review their two important causes, ITKOs and capacity shortage, through discussions of related work and preliminary evaluations.

2.1 Performance Degradation by ITKOs

There exists a report to claim that the main cause of inter-thread cache conflicts is inter-thread kickouts (ITKOs) [11], which means that data fetched by one thread are evicted by other threads. If the former thread again accesses the data that have already been replaced by the others, the cache access results in a cache miss that does not occur in single-thread execution. In [11], the authors indicate that the number of ITKOs is an effective metric to estimate the performance degradation by inter-thread cache conflicts.

Cache partitioning mechanisms have been proposed to avoid ITKOs [3]–[5]. These mechanisms group the ways of a set-associative cache into several parts. These parts are exclusively allocated to threads so that data of a thread are not put in the ways allocated to the other threads. Furthermore, these mechanisms dynamically resize the allocated parts by changing the number of ways in each part. When one thread requires more capacity than another thread, the mechanisms allocate a larger number of ways to the former thread than the latter thread. As a result, these mechanisms can avoid ITKOs without losing the advantage of flexible usage of shared cache resources. In addition, these mechanisms are implemented with low hardware overheads.

2.2 Performance Degradation by Capacity Shortage

Cache partitioning mechanisms can avoid ITKOs because the cache ways are exclusively allocated to threads. However, Moreto et al. have indicated performance degradation even if cache partitioning mechanisms are used [14]. They show that performance degradation is often experienced if the sum of the numbers of ways, which is required for maintaining 90% performance on each thread, exceeds the total number of ways in the cache.

To highlight inter-thread cache conflicts by capacity shortage more clearly, we carry out a preliminary evaluation. The objective of this evaluation is to know the relationship between capacity shortage and performance degradation. This evaluation assumes a 2-core CMP with a 32-way 1 MB L2 (last-level) shared cache, to which the cache partitioning mechanism [5] is applied. Hence, ITKOs do not occur. On this CMP, various combinations of two threads are executed. These two threads are selected from the benchmarks in the SPEC CPU2000 benchmark suite [15].

Figure 1 shows preliminary evaluation results. The horizontal axis shows required capacity by two threads. Here, the required capacity means the capacity that can achieve 95% performance of fully using the cache. The vertical axis shows the weighted speedup [16], which is a performance metric for multi-thread execution. Figure 1 indicates that the

![Fig. 1](image-url) Relationship between the sum of requested capacities of threads and performance when two threads are simultaneously executed.
performance degradation becomes larger as required capacity increases. Namely, the performance obviously degrades when capacity shortage occurs. This fact reveals the importance of avoiding capacity shortage as much as possible.

2.3 Related Work

There are several researches on thread scheduling methods considering inter-thread cache conflicts. The importance of thread scheduling methods has increased because a high degree of freedom to change thread combinations sharing a cache is given by modern CMPs and computer systems with multiple CMPs.

Some researches focused on thread scheduling methods that consider ITKOs. Settle et al. [6] indicated that the main cause of performance degradation in simultaneous multithreading (SMT) processors is inter-thread cache conflicts. Furthermore, they proposed the cache monitoring scheme to find out cache sets that frequently cause misses, which result in ITKOs. Based on the monitoring results, a thread scheduling method prevents threads, which have the same cache sets causing misses frequently, from sharing a cache.

Knauerhase et al. [7] focused on cache weight, cache misses per cycle, as a metric for inter-thread cache conflicts. Their scheduling method lets a light-weight thread and a heavy-weight thread share a same cache. The former thread is not performance-sensitive to the efficiency of cache usage, and the latter thread frequently causes ITKOs.

Some researches focused on thread scheduling methods that consider cache capacity. Suh et al. [8] have proposed a thread scheduling method that considers the required capacity by the threads. However, they did not consider ITKOs in the scheduling method, while they presented the cache partitioning mechanisms separately. On the other hand, to reduce both ITKOs and capacity shortage, this paper originally proposes an approach of combining cache partitioning and thread scheduling. This paper also demonstrates that thread scheduling methods without cache partitioning suffer from the performance degradation due to ITKOs.

Banikazemi et al. [9] have proposed a thread scheduling method based on a performance prediction model considering cache line occupancy, miss rate, and cycles per instruction of threads. They use cache line occupancy as one of metrics for thread scheduling. However, they do not consider the reduction in ITKOs.

Jaleel et al. [10] have proposed a thread scheduling method, CRUISE, which considers cache capacity. They also consider the combined method of a thread scheduling method and an advanced cache replacement policy. Their policy eventually reduces the performance impact of ITKOs. Although they show the effectiveness of the method, they do not mention the existence of ITKOs. This paper clarifies that the proposed method can improve performance because it can reduce both ITKOs and capacity shortage.

These researches can partially prevent the performance degradation caused by inter-thread cache conflicts. However, they do not consider both ITKOs and capacity shortage. The discussions in this section clearly indicate the necessity for the consideration of both ITKOs and capacity shortage to further reduce performance degradation due to inter-thread cache conflicts. The proposed method enables it by using a thread scheduling method and a cache partitioning mechanism simultaneously.

3. A Capacity-Aware Thread Scheduling Method Combined with Cache Partitioning

3.1 Overview

This paper proposes a capacity-aware thread scheduling method combined with cache partitioning. The proposed method has two novelties. One is that this method can reduce both types of inter-thread cache conflicts by combining thread scheduling and cache partitioning. Thanks to the combination, we can expect the synergy effect that can further reduce inter-thread cache conflicts in comparison with the case of considering either inter-thread cache kickouts or capacity. As a result, the proposed method can further improve the performance.

The other is the proposal of the capacity-aware thread scheduling method working together with a cache partitioning mechanism. With help of the cache partitioning mechanism, the proposed method can know the capacity demanded by each thread and decide the thread combinations sharing one cache. As the proposed method can share the profile data with the cache partitioning method, it does not require any extra hardware for profiling. This profiling method is described in Sect. 3.2. Then, the proposed method decides the thread combinations using a capacity-aware scheduling algorithm as mentioned in Sect. 3.3.

Figure 2 shows a CMP architecture assumed in the proposed method. The CMP includes n cores and m shared caches. Each of the shared caches is shared by multiple cores. The cache partitioning mechanisms proposed in [5] manages each of the shared caches, and their hardware overhead is not significant.

The thread scheduler divides the execution time into two phases, profiling phases and scheduled phases, and alternately repeats these phases as shown in Fig. 3. In the profiling phases, required capacity of each thread is esti-
mated. Then, the scheduler decides the thread combinations sharing a cache. In the scheduled phases, all the threads are scheduled based on the results of the previous profiling phase and executed. Such a modification is implemented with a low software overhead as discussed in [17].

3.2 A Profiling Method for Capacity Demand Estimation

To estimate the demand for capacity of each thread in the profiling phase, the proposed method uses a profiling method that leverages the cache partitioning mechanism applied to each shared cache.

In the mechanism, the number of allocated ways to each thread is decided based on stack distance profiling [18], which represents the data access locality. Its concept is illustrated in Fig. 4. $C_x$ means the number of accesses to blocks at the $x$-th LRU position ($1 < x < N$ in the case where $N$ ways are allocated by the cache partitioning mechanism). If the thread has a low locality, accesses are widely spread over all the positions as shown in Fig. 4(a). It means that the capacity required by the thread is larger than the cache capacity. On the other hand, if the thread has a high locality, accesses are frequently occur at the MRU and nearby positions as shown in Fig. 4(b). To quantify this difference, a locality assessment metric $D$ is defined as $C_1/C_N$ [19]. The mechanism combines $D$ and two predefined thresholds, $t_1$ and $t_2$, for cache resizing. If $D$ becomes larger than $t_2$, the cache partitioning mechanism allocates one more way to the thread. If $D$ becomes smaller than $t_1$, the mechanism deallocates one way.

From the above behavior of the cache partitioning mechanism, we define the number of actually-allocated ways obtained by single-thread execution as the necessary cache capacity of the thread. The definition of the capacity required by the thread is formulated as follows.

$$W = \frac{\sum_{t=t_0}^{t_1} w(t)}{t_1 - t_0}.$$  

(1)

Here, $W = \text{(the capacity required by the thread,)}$
capacity of each thread by the proposed profiling method. Then, a CMP executed a workload that consists of these threads. Here, their assignment to the cores is decided by the proposed algorithm based on the profiling data. Therefore, simulation results provide valuable insights into the performance improvement of the proposed method.

For this evaluation, we developed a simulator based on the M5 simulator [20]. The simulated CMP has four cores and two last-level L2 shared caches, each of which is shared by two cores. The cache partitioning mechanism [5] is applied to each of the L2 caches. The other simulation parameters of the CMP are listed in Table 1. The workloads executed on the CMP consist of four threads. Each workload is executed for 1 billion cycles.

The threads included in the workloads are selected from the SPEC CPU2000 benchmark suite [15]. From the suite, the six representative benchmark programs are selected based on the characteristics of cache accesses. Workloads using these threads become representatives of all workloads from the viewpoint of demand for capacity. This process can reduce the number of evaluated workloads without loss of generality.

In the selection, all benchmarks are categorized into three classes based on their utility graphs [4]. As a result of the classification, the benchmark programs are classified into high-utility, saturating-utility, and low-utility. Then, two representative benchmarks are selected from each category, and the six benchmarks listed in Table 2 are used in the following evaluation. Figure 6 shows the utility graphs of the selected benchmarks. In this figure, the performance is evaluated by IPC normalized by that when 32 ways are fully allocated to the thread by the cache partitioning mechanism. VprPlace and Twolf are categorized into high-utility (H) benchmarks. High-utility benchmarks increase their performances gradually as the number of ways increases. Mesa and Equake are saturating-utility (S) benchmarks. Their performances are drastically improved by increasing the number of ways. However, their performances are hardly improved when the number of allocated ways exceeds a certain value. Wupwise and Apple are low-utility (L) benchmarks. The performances 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, moderate, and low demands for capacity, respectively. The number of ways allocated by a cache partitioning mechanism in Table 2, i.e. W, clearly indicates that the profiling method can properly estimate the necessary cache capacity of each thread.

From these benchmarks, six workloads are generated, whose characteristics differ from each other. For example, both [VprPlace, Equake, Wupwise, Apple] and [Twolf, Mesa, Wupwise, App] consist of one H thread, one S thread, and two L threads. In this case, only the former workload is used. Table 3 shows the six workloads finally selected. The workloads are denoted by the labels that represent the utility characteristics of the included threads. For example, HSSL means the combination by H, S, L, and L threads.

### 4.2 Effects of Capacity-Aware Thread Scheduling

Section 4.2 discusses the effects of capacity-aware thread scheduling. For the discussion, the performance of the thread combination decided by the proposed method is compared with that of the other combinations. When four threads in a workload are scheduled to cores on the CMP, all possible combinations from the viewpoint of cache sharing are shown in Fig. 7. Naturally, the proposed method selects one of the three combinations, considering the shared cache capacities. To evaluate the performance of the CMP, the weighted speedup [16] is used as the performance metric in this evaluation.
Figure 7 Possible thread combinations considering cache sharing for each workload.

Figure 8 Comparing the performances of the worst, the random, the best, and the proposed combination.

4.3 Synergy Effects of Combining Thread Scheduling and Cache Partitioning

Section 4.3 discusses the synergy effects of combining thread scheduling and cache partitioning. To this end, this paper compares the cases that are not supported by the cache partitioning mechanisms with the proposed method.

It should be noted that the weighted speedup might be degraded by ITKos if cache partitioning is not used. Thus, even if thread scheduling works best, the best performance is not achieved without cache partitioning. To discuss the synergy effects of combining capacity-aware thread scheduling and cache partitioning, Fig. 9 shows the weighted speedups of the worstTS, randomTS, bestTS and caTS thread combinations, which are obtained without using cache partitioning. For comparison, the weighted speedups achieved by the proposed method CP+caTS are also given in the figure.

As shown in this figure, even the bestTS cannot outperform CP+caTS. In almost all the workloads, the performance of caTS is less than that of bestTS and is sometimes equal to that of worstTS, even though the performance of CP+caTS is always comparable with or higher than that of bestTS. These results clearly demonstrate the importance of combining capacity-aware thread scheduling and cache partitioning to reduce inter-thread cache conflicts. Accordingly,
it is clearly demonstrated that the proposed method outperforms conventional methods that consider either thread scheduling or cache partitioning.

4.4 Performance Impact for Individual Threads

To further discuss the performance improvement by the proposed method, we focus on individual threads in each workload using Fig. 10. This figure represents the performances of the threads with the first, second, third, and fourth largest $W$ in a workload, which are calculated by averaging the performances of the threads among all the workloads. The vertical axis means the normalized IPC, which is IPC normalized by that when 32 ways are fully allocated. The horizontal axis indicates threads in descending order of $W$. In a similar fashion with Fig. 8, Fig. 10 includes the results of all the combinations.

Compared with $CP+\text{randomTS}$, $CP+caTS$ achieves 2% and 3.5% performance improvements for the first and second threads, respectively. Moreover, compared with $CP+\text{worstTS}$, $CP+caTS$ realizes 5% and 12% performance improvements for the first and second threads, respectively. From these results, it is clear that the proposed method can improve the performances of the two threads with the larger $W$. Performance improvements of the second thread in each workload tends to become larger than those of the first thread. The second thread cannot get enough cache capacity compared with the first thread if these threads share a same cache. The proposed method can solve this situation due to capacity shortage, and the performance improvement of the second thread becomes larger.

On the other hand, $CP+caTS$ reduces 1% and 0.3% of performance on an average in the third and fourth threads, respectively, compared with $CP+\text{bestTS}$. This is because these threads are coupled with the threads with larger $W$ by the proposed method. As a result, the numbers of allocated ways for the third and fourth threads decrease. However, since their performance degradations are not significant, the overall performance of the CMP improves.

In addition, fairness of the performances among the threads improves by the proposed method. In $CP+\text{worstTS}$, the performance degradations of the first and second threads are large. However, in $CP+caTS$, their performances significantly improve while the performances of the third and fourth threads slightly degrade. As a result, the difference in performance among the four threads becomes small.

5. Conclusions

This paper has focused on reducing two different types of inter-thread cache conflicts, ITKOs and capacity shortage, and has proposed a capacity-aware thread scheduling method combined with cache partitioning. In the proposed method, the cache partitioning mechanisms are applied to the shared caches on the CMP. These mechanisms provide the thread scheduler with information about the necessary cache capacity of each thread. The scheduler decides thread combinations so that it avoids scheduling two threads with high demands for capacity sharing the same cache. As a result, the proposed method alleviates a performance loss due to ITKOs and capacity shortage.

The evaluation results show that the proposed method can improve performance by up to 8.1% for overall performances and 12% for performance of threads with high demands for capacity. It is also clarified that giving consideration to both ITKOs and capacity shortage enables us to reduce the performance degradation by inter-thread cache conflicts better than considering only either ITKOs or capacity shortage. Furthermore, since the cache partitioning mechanism [5] and the modified thread scheduler [17] are implemented with low overheads, the hardware and software overheads to realize the proposed method is affordable. In future work, we evaluate additional performance improvement by dynamically adapting thread assignments to phase changes of applications.

Acknowledgment

This research was partially supported by Core Research for Evolutional Science and Technology (CREST) from Japan Science and Technology Agency (JST).

References

[1] S. Hily and A. Seznec, “Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading,” Tech. Rep. PI-1086, INRIA, Feb. 1997.
[2] R. Thekkath and S.J. Eggers, "The effectiveness of multiple hardware contexts," ACM SIGOPS Operating Systems Review, vol.28, no.5, pp.328–337, Dec. 1994.

[3] G.E. Suh, L. Rudolph, and S. Devadas, “Dynamic partitioning of shared cache memory,” J. Supercomputing, vol.28, no.1, pp.7–26, April 2004.

[4] M.K. Qureshi and Y.N. Patt, “Utility-based cache partitioning: An overhead, high-performance, runtime mechanism to partition shared caches,” Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp.423–432, Dec. 2006.

[5] I. Kotera, K. Abe, R. Egawa, H. Takizawa, and H. Kobayashi, “Power-aware dynamic cache partitioning for CMPs,” Trans. High-Performance Embedded Architectures and Compilers, vol.3, no.2, pp.149–167, 2008.

[6] A. Settle, J. Kihm, A. Janiszewski, and D.A. Connors, “Architectural support for enhanced SMT job scheduling.” Proc. 13th International Conference on Parallel Architecture and Compilation Techniques, pp.63–73, Sept. 2004.

[7] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, “Using OS observations to improve performance in multicore systems,” IEEE Micro, vol.28, no.3, pp.54–66, May 2008.

[8] G.E. Suh, S. Devadas, and L. Rudolph, “A new memory monitoring scheme for memory-aware scheduling and partitioning,” Proc. Eighth International Symposium on High-Performance Computer Architecture, pp.117–128, Feb. 2002.

[9] M. Banikazemi, D. Poff, and B. Abali, “PAM: A novel performance/power aware meta-scheduler for multi-core systems,” Proc. ACM/IEEE Conference on Supercomputing, Nov. 2008.

[10] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer, “CRUISE: Cache replacement and utility-aware scheduling,” ACM SIGARCH Computer Architecture News, vol.40, no.1, pp.249–260, April 2012.

[11] J. Kihm, A. Settle, A. Janiszewski, and D.A. Connors, “Understanding the impact of inter-thread cache interference on ILP in modern SMT processors,” J. Instruction-Level Parallelism, vol.7, pp.1–28, June 2005.

[12] “Intel Core 2 Quad Processor New version now available based on Intel’s 45 nm technology,” Intel Product Brief, Intel Corporation, 2007.

[13] “Quad-Core Intel Xeon Processor 5300 Series,” Intel Product Brief, Intel Corporation, 2007.

[14] M. Moreto, F.J. Cazorla, A. Ramirez, and M. Valero, “Explaining dynamic cache partitioning speed ups,” IEEE Computer Architecture Letters, vol.6, no.1, pp.1–4, Jan. 2007.

[15] J.L. Henning, “SPEC CPU2000: Measuring CPU performance in the new millennium,” Computer, vol.33, no.7, pp.28–35, July 2000.

[16] A. Snively and D.M. Tullsen, “Symbiotic job scheduling for a simultaneous multithreading processor,” ACM SIGPLAN Notices, vol.35, no.11, pp.234–244, Nov. 2000.

[17] F.J. Cazorla, P.M.W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, “Predictable performance in SMT processors: Synergy between the OS and SMTs,” IEEE Trans. Comput., vol.55, no.7, pp.785–799, June 2006.

[18] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” Proc. 11th International Symposium on High-Performance Computer Architecture, pp.340–351, Feb. 2005.

[19] H. Kobayashi, I. Kotera, and H. Takizawa, “Locality analysis to control dynamically way-Adaptable caches,” ACM SIGARCH Computer Architecture News, vol.33, no.3, pp.25–32, June 2005.

[20] 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,” IEEE Micro, vol.26, no.4, pp.52–60, July 2006.