Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs?

Eddy Z. Zhang Yunlian Jiang Xipeng Shen

Department of Computer Science
The College of William and Mary
Williamsburg, VA, USA 23187-8795
{eddy,jiang,xshen}@cs.wm.edu

Abstract

Most modern Chip Multiprocessors (CMP) feature shared cache on chip. For multithreaded applications, the sharing reduces communication latency among co-running threads, but also results in cache contention.

A number of studies have examined the influence of cache sharing on multithreaded applications, but most of them have concentrated on the design or management of shared cache, rather than a systematic measurement of the influence. Consequently, prior measurements have been constrained by the reliance on simulators, the use of out-of-date benchmarks, and the limited coverage of deciding factors. The influence of CMP cache sharing on contemporary multithreaded applications remains preliminarily understood.

In this work, we conduct a systematic measurement of the influence on two kinds of commodity CMP machines, using a recently released CMP benchmark suite, PARSEC, with a number of potentially important factors on program, OS, and architecture levels considered. The measurement shows some surprising results. Contrary to commonly perceived importance of cache sharing, neither positive nor negative effects from the cache sharing are significant for most of the program executions, regardless of the types of parallelism, input datasets, architectures, numbers of threads, and assignments of threads to cores. After a detailed analysis, we find that the main reason is the mismatch of current development and compilation of multithreaded applications and CMP architectures. By transforming the programs in a cache-sharing-aware manner, we observe up to 36% performance increase when the threads are placed on cores appropriately.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming; D.3.4 [*Programming Languages*]: Processors—optimization, compilers; D.4.1 [*Operating Systems*]: Process Management—Scheduling

General Terms Performance, Measurement

Keywords Shared Cache, Thread Scheduling, Parallel Program Optimizations, Chip Multiprocessors

PPoPP'10, January 9–14, 2010, Bangalore, India.

1. Introduction

One of the features that distinguish modern Chip Multiprocessors (CMP) from traditional processors is the presence of cache sharing among multiple computing units on a chip. The sharing reduces communication latency among co-running threads, but also results in cache conflicts and contention among threads. On a system with multiple chips, the sharing further shows non-uniformity: Cores across chips typically do not share cache as the cores in a chip do.

Researchers have recognized the importance of an effective use of shared cache and developed a number of techniques to exploit it. For example, cache-sharing-aware scheduling in operating systems (OS) research has shown that by assigning suitable programs or threads onto the same chip, one can alleviate the cache contention among co-runners (processes or threads running on sibling cores) and reduce inter-thread communication latency, improving program performance considerably. The effectiveness of those techniques has shown on sets of independent jobs [8, 24, 7, 20] as well as parallel threads inside certain classes of single applications [23].

However, in this work, through a systematic measurement, we find that contrary to the commonly perceived significant effects, cache sharing has very limited influence, either positive or negative, on the performance of the applications in PARSEC—a recently released benchmark suite that "focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors" [3]. Our experiments show that for those programs, no matter how the threads are placed on cores (they may share the cache in various ways or do not share cache at all), the performance of the programs remains almost the same.

This surprising finding comes from a systematic measurement consisting of thousands of runs, covering various potentially important factors on the levels of programs (number of threads, parallel models, phases, input datasets), OS (thread binding and placement), and architecture (types of CMP and number of cores). It is derived from the measured running times, and confirmed by the low-level performance reported by hardware performance counters.

After conducting a detailed analysis, we find that the fundamental reason for the insignificant influence is that the development and the currently standard compilation of the programs are oblivious to cache sharing, causing a mismatch between the generated programs and the CMP cache architecture. The mismatch shows on three aspects. First, the data sharing among threads in those programs is typically uniform, that is, the amount of data a thread shares with one thread is typically similar to the amount it shares with any other thread. The uniformity mismatches with the non-uniform cache sharing on CMPs, explaining the insensitivity of the program

performance on the placement of threads. Second, the accesses to shared cache lines are limited for most of the programs because of the uniform partition of computation and data among threads, explaining the small constructive effects from shared cache. Finally, the working sets of the programs are typically much larger than the shared cache. The difference between the sharing and non-sharing cases in terms of cache size per thread is not enough to make significant changes in cache misses. Cache contention hence show little influence as well.

The second part of this paper explores the implications of the observed insignificance. At the first glance, the observation might seem to suggest that exploitation of cache sharing is unimportant for the executions of the multithreaded applications. But a set of experiments demonstrate the exact opposite conclusion: Exploiting cache sharing has significant potential, but to exert the power, cache-sharing-aware transformations are critical.

In the experiments, we increase the amount of shared data among sibling threads (the threads to run on the same chip) through certain code transformations. The transformations yield non-uniform data sharing among threads, matching with the non-uniform cache sharing on the architecture. The influence of cache sharing becomes much more significant than on the original programs. Appropriate placement of threads on cores cuts over half of cache misses and improves performance by up to 36%, compared to other placements and the original programs.

To the best of our knowledge, this work is the first that *systematically* examines the influence of cache sharing in modern CMP on the performance of *contemporary multithreaded* applications. Many previous explorations [8, 24, 9, 7, 20] are concentrated on co-runs of independent programs, on which, cache contention is the single main influence by shared cache. The studies on multithreaded programs have been focused on certain aspects of CMP, rather than a systematic measurement of the influence from cache sharing. For instance, many of them have used simulators rather than real machines; some [25] have used old benchmark suites (e.g., SPLASH-2 [26]), or have concentrated on a specific class of applications, such as server programs [23]; some [12] have used old CMP machines with no shared cache equipped. These limitations may not be critical for the particular focus of the previous research—in fact, sometimes they are unavoidable (e.g., using simulators for cache design). However, they may cause biases to a comprehensive understanding of the influence of cache sharing on program performance. As far as we know, none of the previous explorations has included the many factors as covered in this work. These differences explain the departure between the observations made in this work and the previous.

Similar to the observation made by Sarkar and Tullsen [16], we have found only a small number of studies [16, 10, 13] on exploiting *program transformations* for the improvement of shared cache usage (a clear contrast to the large body of work in OS and architecture areas.) With the importance of program transformations demonstrated in this work, hopefully more research efforts will be triggered in this direction.

In summary, this work consists of three-fold contributions.

- We conduct a systematic measurement on the influence of cache sharing in modern CMP on the performance of contemporary multithreaded applications with seven factors on three levels (program, OS, architecture) considered. The measurement reveals novel observations on the influence of cache sharing.
- We uncover the reasons for the insignificant influence of cache sharing on the multithreaded applications, pointing out that their mismatch with the underlying CMP cache architecture is the main obstacle for exerting the potential of shared cache.
- Through a set of experiments, we demonstrate the potential of cache-sharing-aware program transformations, and conclude that program transformations are the key for exerting the power of shared-cache management (e.g., shared-cache-aware scheduling).

In the rest of this paper, we describe the design of the measurement in Section 2, report the measurement results and findings in Section 3, present the exploration on cache-sharing-aware transformation in Section 4, discuss related work in Section 5, and conclude the paper in Section 6.

Table 1. Benchmarks

Program	Description	Parallelism	Working Set
Blackscholes	Black-Scholes diff-eqtn	data	2MB
Bodytrack	body tracking	data	8MB
Canneal	sim. annealing	unstruct.	256MB
Dedup	stream compression	pipeline	256MB
Facesim	face simulation	data	256MB
Ferret	image search	pipeline	64MB
Fluidanimate	fluid dynamics	data	64MB
Streamcluster	online clustering	data	16MB
Swaptions	portfolio pricing	data	512KB
X264	video encoding	pipeline	16MB

∗: see [3] for detail.

2. Experiment Design

In this section, we first introduce the benchmark suite we use, then present the factors that we vary in the measurement and the corresponding rationales, and finally describe the schemes used for the measurement of times and hardware performance.

2.1 Benchmarks

We use PARSEC [3] as the benchmark suite. It is a recently released suite designed for CMP research. The suite includes emerging applications in recognition, mining and synthesis, as well as systems applications that mimic large-scale multithreaded commercial programs. Studies [3, 2] have shown that the suite covers a wide range of working set sizes, and a variety of locality patterns, data sharing, synchronization, and off-chip traffic, making it an attractive choice over some old parallel benchmark suites such as SPLASH-2 [26]. Table 1 lists the 10 programs we use with the working set sizes (on *simlarge* inputs). Programs *dedup* and *ferret* both use the pipeline parallelization model with a dedicated pool of threads for each pipeline stage. Programs *facesim*, *fluidanimate*, and *streamcluster* have streaming behavior. Other programs are data-level parallel programs with different amount and patterns of synchronizations and inter-thread communications. We are unable to use two other programs, *vips* and *freqmine*, because we had difficulty in binding the threads in those programs with processors. All the programs we use are written in Pthreads API. All employ standard Pthreads schemes (locks and barriers) for synchronizations, except *canneal*, which uses an aggressive synchronization strategy based on data race recovery.

2.2 Factors

To achieve a comprehensive understanding on how much cache sharing influence the performance of multithreaded applications, our experiments include a number of factors that are potentially important for the influence. In this section, we briefly describe those factors and the rationale for selecting them. Table 2 summarizes the

Table 2. Dimensions covered in the measurement

Dimension	Variations	Description
benchmarks	10	from PARSEC
inputs	4	*simsmall, simmedium, simlarge, native*
# of threads*	4	1,2,4,8
parallelism	3	data, pipeline, unstructured
binding	2	yes, no
assignment*	3	thread assignment to cores
platforms	2	Intel Xeon & AMD Opteron
subset of cores	7	the cores a program uses

*: *Dedup* and *Ferret* have more threads and assignments (see Section 3.3).

variations of the factors, and Section 3 elaborates on the treatment of the factors in the systematic measurement.

The considered factors come from the program, OS, and architecture levels as follows. (Words in bold fonts correspond to the dimensions in Table 2.)

- *Program Level* The major factors include the **input** datasets to the program, the **number of threads**, and the **parallel models**. The first two factors determine the working set of a thread and the intensity of cache contention. We use four input datasets included in PARSEC, as listed in Table 2 in increasing order of size, and vary the number of threads from one to eight. The third factor, parallel models, determines the patterns of data sharing and computation.
- *OS Level* The major effect from the OS is thread scheduling, which determines the co-runners on a chip. To examine the potential of the scheduling, we avoid using any particular scheduling algorithms. Instead, we experiment with different **thread-core assignments** to cover different co-running scenarios as detailed in Section 3. Because the experiment needs binding threads to cores, we examine the effects of **binding** by comparing to non-binding cases (detailed in Section 3.4.)
- *Architecture Level* We use a Dell PowerEdge 2950 server equipped with 2 quad-core Intel Xeon E5310 processors, and a Dell PowerEdge R80 hosting 2 AMD Opteron 2352 processors. The two machines are the representatives of two typical CMP **architectures** on the market. The Intel machine is based on Front-Side-Bus (FSB) with an inclusive cache hierarchy; the AMD machine is a Cache Coherent None-Uniform Memory Access (ccNUMA) CMP with HyperTransport links and an exclusive cache hierarchy[1]. The explorations on both of them may exhibit the impact of architecture features on how cache sharing influences the performance of multithreaded applications. Both machines run Linux 2.6.22 with GCC4.2.1 installed. Table 3 reports the detail of the hardware.
  When the number of threads is smaller than the total number of cores in a machine (8 in our experiment), the threads may be assigned to different **subsets of cores**. We experiment with up to 7 (depending on the number of threads) different sets to cover most representative sharing scenarios. In the case of 2 threads on the Intel machine, for instance, the sets of cores we use include 2 sibling cores that share cache, 2 non-sibling cores on a single chip which share the same memory-processor bus, and 2 cores residing on different chips. The 4-thread case has 3 corresponding sets. The 8-thread case has only 1 set, the set of all cores.

[1] The new Intel CMP, Nehalem, resembles this AMD architecture but with an inclusive cache hierarchy.

Program phase changes may affect the measurement results, especially on the measured potential of thread scheduling. We address this factor as described in Section 3.2.

2.3 Measurement Schemes

Our measurement concentrates on running times, cache miss rates, and the amount of shared-data accesses. We use the built-in utility HOOKS in the PARSEC suite to measure running times, and employ the Performance Application Programming Interface (PAPI) library [4] to read memory-related hardware performance counters, including cache miss rates, memory bus transactions, and the reads to cache lines in a "shared" state for every thread. (As required by PAPI for thread-level measurement, we set the pthread scheduling scope to "system" in the hardware performance monitoring.)

Each instance of the set of factors listed in Table 2 determines a setting of a run. We call such an instance a *configuration*. For each configuration, we conduct 5 to 10 repetitive runs to reduce the interference from random noises. By default, we use the average performance of the repetitive runs; when necessary, we report the variations as well.

3. Measurement and Findings

In this section, we report the detail of the experiments, the results, and findings. As the focus of this work is on the performance influence from cache sharing, our experiments center on the comparisons between the sharing and non-sharing cases—that is, when the threads are bound to sibling or non-sibling cores respectively, as shown in Section 3.1. To prevent the effects of thread scheduling from blurring the observations, for the sharing case, we also examine the performance difference caused by different assignments of threads on cores, as reported in Section 3.2. We describe the results of *dedup* and *ferret* separately in Section 3.3. They are two typical pipeline programs with task-level parallelism and numerous pipeline stages. Each stage is handled by a pool of threads. Unlike other programs, the interactions among the threads in these two programs exist both within and between stages, requiring a different set of measurements. The other pipeline program, *x264*, behaves like a data-parallel program, with each thread working on an image frame. So we report its results together with the non-pipeline programs.

3.1 Sharing Versus Non-Sharing

To study the influence of cache sharing, we compare the sharing case where the threads are bound to sibling cores, and the non-sharing case where the threads run on non-sibling cores. Let a be the number of threads per chip in the sharing case. The average cache size per thread in the sharing case is $1/a$ of the size in the non-sharing case. The reduced size is part of the effects of cache sharing. We will see that the resulting influence on performance is insignificant.

We use two and four threads in the experiments. (We did not use 8 threads as there would be no interesting non-sharing case to compare.) On the AMD machine, because of the quad-core sharing, the two 4-thread cases actually both have some cache sharing: In the 4-thread sharing case, all four threads run on one chip, thus share one cache; In the 4-thread non-sharing case, there are two threads per chip.

When there are more than one way to assign the threads to cores, we pick the most straightforward way. For instance, in the case of 4-thread sharing case on the Intel machine, we assign threads 0 and 1 to two sibling cores and threads 2 and 3 to the other two sibling cores on a chip. Section 3.2 will show that other ways of assignments produce similar results.

Figure 1 presents the performance comparison. The running time shown by a bar is the running time of the program in the

Table 3. Configuration of machines

	CPU	L1	L2	L3	Memory
Intel	Xeon E5310 1.6GHz quad-core	32KB	2x4MB, each shared by 2 cores	None	8GB shared bus
AMD	Opteron 2352 2.1GHz quad-core	64KB-Icache 64KB-Dcache	512KB	2MB shared by 4 cores	8GB ccNUMA

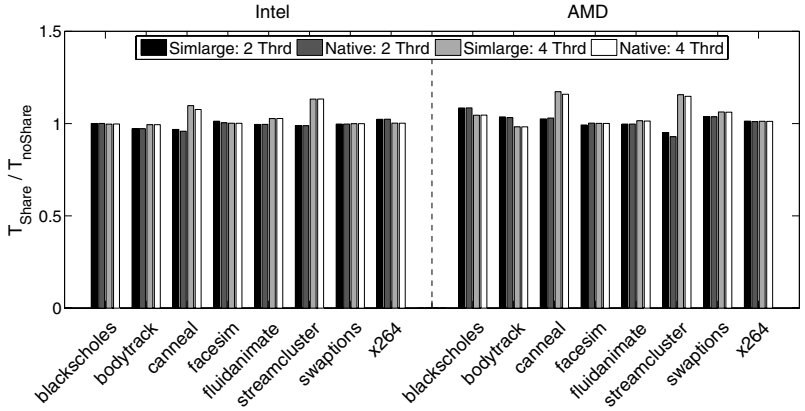


Figure 1. The running time of each program in the sharing case normalized to its running time in the non-sharing case. The bars in a group from left to right correspond to the cases of 2 threads on the *simlarge* input, 2 threads on the *native* input, 4 threads on the *simlarge* input, and 4 threads on the *native* input.

sharing case normalized to the time in the non-sharing case. So, a bar higher than 1 means the contention on the shared cache and memory bus causes slowdown to the program in the sharing case; a bar lower than 1 indicates that the constructive sharing improves the performance of the program. The contention caused by shared cache generates some slowdown to *canneal* and *streamcluster* when the large inputs, *native* inputs, are used, but not much for other programs. On the other hand, the sharing improves the performance of *canneal* and *streamcluster* slightly for *simlarge* inputs, showing that the constructive effects outweigh the cache contention influence when inputs become small. But overall, the sharing shows insignificant influence for most of the programs performance.

The measured cache miss rates further confirm the observed small influence on the performance. Figure 2 plots the cache accesses and misses averaged over the threads on the Intel machine for the 2-thread cases on *native* inputs. The cache misses are similar in the sharing and non-sharing scenarios for every program, consistent with the running time results shown in Figure 1.

The reasons for the insignificance of the influence come from two aspects. First, the small amount of inter-thread data sharing determines the limited constructive effects from shared cache. Figure 3 shows the portion of all the reads on shared cache that happen to access a cache line with a "shared" state (i.e., more than one cores have been reading the data in the cache line.) The larger the portion is, the more data that the co-running threads may prefetch for each other, and hence the more constructive effects cache sharing may impose. The portions are less than 7% for all the programs. Analysis of the source code of the programs confirms the finding. Take the program *canneal* as an example. Each of its threads operates on randomly picked two nodes in a network in every iteration. Because of the large size of the network and the randomness in node selection, it is no surprise to see the small amount of references on shared data blocks.

Second, because the working sets of the programs, as shown in Table 1, are typically much larger than the shared cache on

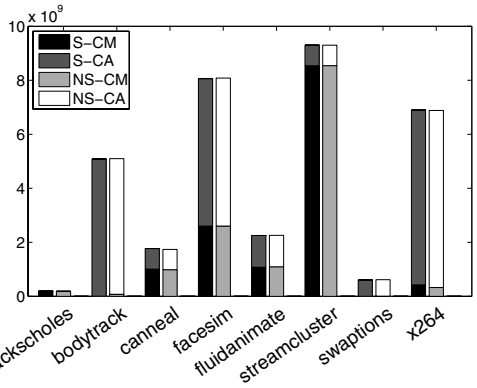


Figure 2. Comparisons of L2-cache accesses and misses for 2-thread cases on the Intel machine. ("S" for cache-sharing cases; "NS" for non-cache-sharing cases; "CM" for cache misses; "CA" for cache accesses.)

a processor, the difference of the cache size per thread between the sharing and non-sharing cases is not enough to make significant changes in cache misses. The cache sharing therefore shows no clear negative effects either. The working set of the program *blackscholes* is smaller than the shared cache on the Intel machine, but it has very few L2 cache line reuses, as shown in Figure 2. So the cache sharing has little influence on it either.

3.2 Comparisons Among Sharing Cases

The threads in a parallel program usually have certain differences among one another. Threads in a data-level parallel program may compute on different sections of data, resulting in different working sets. Threads in pipeline programs may execute different tasks. In

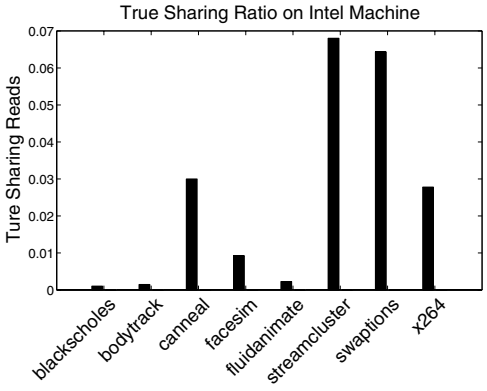


Figure 3. Read sharing for 2-thread cases on Intel. It is computed as the number of read accesses to the L2-cache lines with a "shared" state, normalized by the total number of L2-cache accesses.

both types of programs, there may be non-uniform communication and data sharing across threads.

In light of the non-uniform cache sharing, the differences among threads may offer opportunities for performance improvement through appropriate placement of threads on cores. The sharing cases considered in the previous subsection contain just one thread-core assignment for each scenario. This section examines the impact that different assignments may have by binding threads to cores in various ways.

For 4-thread cases, we permutate the thread-core assignments and exhaust distinctive co-running combinations. For 8-thread cases, we use three representative thread assignments in both the Intel and AMD architectures. We place the threads in such a way that threads whose indices differ by a given distance are assigned to sibling cores. For example on Intel machine, with the distance set to 1, every two consecutive threads reside on two sibling cores. We vary the distance from 1 to 2 to 4.

Table 4 shows the performance difference caused by the different assignments when the *native* inputs are used. Similar results are observed on other input sets. As the table shows, 6 of the 8 benchmarks have less than 5% maximum difference. For the programs that have over 5% differences, *canneal, facesim*, we find that the times and cache miss rates for the multiple runs of a fixed *configuration* fluctuate considerably. For instance, five *simlarge* runs of *canneal* on the AMD machine have running times as 0.85, 0.72, 0.83, 0.96, and 0.93. After applying a statistical analysis on the data (Student-distribution with 90% as the confidence value), we observe overlaps of the confidence intervals of different bindings, indicating that the difference in running times is statistically insignificant.

Overall, the different thread-core assignments do not show considerable effects on the program performance. There are two possible reasons. First, the threads in those programs may have similar interactions (communications, synchronizations, etc.) with one another, that is, for each thread, its relations with any other threads may be similar. The second possible reason is program phases. It could be that even though the interactions among threads are not similar among one another, but the interactions show different patterns in different phases of the execution so that no particular assignments work well for all the phases.

We conduct a more detailed experiment to determine the exact reason. We collect the cache miss rates of every 100 million instructions (a typical interval granularity used in phase detection [19, 18]) when the program runs in different thread-core assignments. Figure 4 plots the temporal traces of the L2-cache miss rates of *swaptions* and *fluidanimate* when they run on the Intel machine with threads placed on two pairs of sibling cores differently. The three curves in each graph correspond to three sharing cases, in which, the cache on a processor is shared by a different pair of threads. The two programs show different phase change patterns. But on both of them, the three sharing cases show similar L2-cache miss rate curves. Similar phenomena are seen on other programs, indicating that the uniform interplay among threads rather than phase changes is the reason for the observed insignificance of the influence of thread-core assignments.

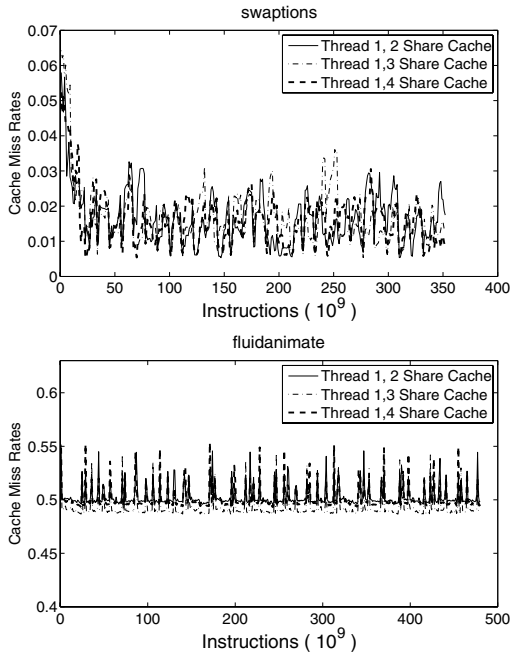


Figure 4. Temporal traces of the L2 cache miss rates on the Intel machine when 4 threads are placed on the same set of cores differently.

Table 4. Maximal percentage of the performance differences caused by different bindings of threads to a given set of cores

	AMD		Intel	
Benchmarks	4-t	8-t	4-t	8-t
Blackscholes	0.03	0.01	0.12	0.02
Bodytrack	0.6	0.97	0.64	1.02
Canneal	3.4	7.18	9.34	2.56
Facesim	0.16	11.15	0.43	0.23
Fluidanimate	0.25	0.71	1.23	2.29
Streamcluster	1.88	0.08	0.13	0.05
Swaptions	0.3	1.08	0.1	1.01
X264	0.32	1.12	0.17	0.2

As a side note, the insignificant influence seems to suggest little potential of thread co-scheduling (or thread clustering) for improving the performance of these programs, a contrast to previous results on independent jobs [8, 20] and server programs [23]. However, Section 4 will show that program transformations would lead to an opposite conclusion.

3.3 Pipeline Programs

Unlike data-parallel programs, typical pipeline programs contain numerous concurrent computation stages, and the interactions among the threads exist both within and between stages.

Table 5. Performance of *ferret* on the Intel machine with different thread placement on cores. (S: pipeline stage)

Thread-core Binding of 6 Stages			Time
S1	S2,S3,S4,S5	S6	(s)
0	{0 2 4 6},{0 2 4 6},{1 3 5 7},{1 3 5 7}	1	210.1
0	{0 4 1 5},{2 6 3 7},{0 4 1 5},{2 6 3 7}	2	160.9
0	{1 5 0 4},{3 7 2 6},{1 5 0 4},{3 7 2 6}	3	161.1
0	{0 2 4 6},{1 3 5 7},{0 2 4 6},{1 3 5 7}	1	161.6
0	{0 2 1 3},{4 5 6 7},{0 2 1 3},{4 5 6 7}	4	161.3
No binding			165.7

In PARSEC, *ferret* and *dedup* are two such programs. The program *ferret* is a search engine, which finds a set of images that best match a query image by analyzing their contents. The program *dedup* is a program that detects and eliminates redundancy in a data stream. The two programs use a similar producer-consumer model for task parallelism: Every job has to go through several stages before completion; The processing results in one stage is passed to the next stage; Every stage has a dedicated thread pool. The product queue between every two stages is protected by lock-based synchronization schemes. As the programs show similar experimental results, we concentrate on *ferret* for explanation.

The program *ferret* has 6 concurrent pipeline stages. The first and final stages are the initialization and completion stages with only one thread in each. The other four stages have the same number of threads. The number is specified in the program input.

Unlike the observations in the previous section, different thread-core assignments for *ferret* sometimes cause significant performance difference. However, the reason for the difference is not the effects from the shared cache, but the load balance across stages. We illustrate the phenomenon by showing the performance of five representative assignments in Table 5, each having four threads in every middle stage. Each tuple in the table represents the thread-core assignment in a stage. For example, the first $\{0,2,4,6\}$ tuple in the top assignment means that the four threads in the second stage are bound to cores $0, 2, 4, 6$ in their creation order. (The core layout on the machine is that cores of even index numbers are on one chip and odd numbers on the other.)

Among the five binding cases in Table 5, four have about 160s running times, much smaller than the other binding case. What is common about the four good cases is that half of the 8 cores are assigned to stages 1 and 3, and the other cores are assigned to stages 2 and 4. A detailed analysis shows that stages 3 and 4 are the bottleneck, taking more time than other stages. Further experiments confirm that as long as stages 3 and 4 do not share cores, the running times are always about 160s; otherwise, the performance is considerably worse. The non-binding case takes about 160s as well, indicating that the dynamic load balancing in the default Linux scheduler successfully avoids the contention between stages 3 and 4.

The cache miss rate results further confirm that the shared cache is not the main reason for the performance difference. Figure 5 shows the L2-cache miss rates on the Intel machine when we run *ferret* using the top 3 assignments listed in Table 5. Every box in the plot presents the distribution of the cache miss rates of the threads in a stage in 5 runs. The results show that even though the cache miss rates in the different stages differ significantly, the different thread-core assignments impose minor influence on the cache miss rates. In fact, the first configuration (*c1*) shows slightly lower cache miss rates than others, but its performance is the worst, echoing that load balance rather than shared cache is the main factor for such programs.

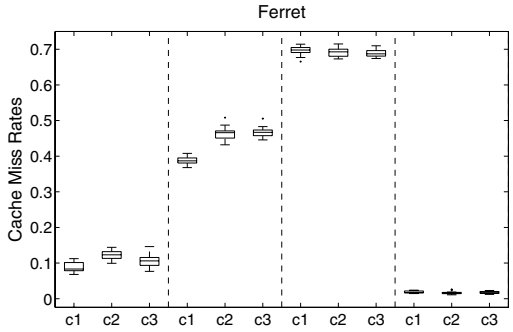


Figure 5. Box plot of L2-cache miss rates per thread on the Intel machine when different thread-core assignments are used. (c1,c2,c3 refer to the top 3 configurations in Table 5.)

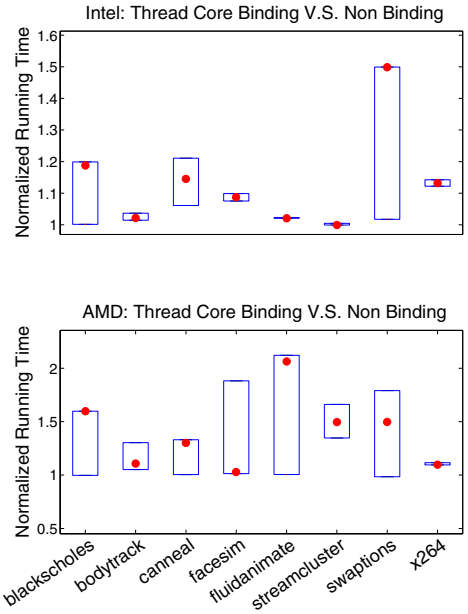


Figure 6. Effects of thread-core binding on Intel and AMD machines. Every box shows *min*, *max* and *median* $(\cdot)$ of the five runs for every configuration. The Y-axis is the running time in the non-binding case normalized to the average time in the binding case.

3.4 Effects of Thread Binding

As many of the measurements bind threads with cores, in this part, we examine the effect of the binding, showing that binding threads to cores typically does not worsen the program performance on CMP and thus is a valid way for the study of the influence of cache sharing.

In the binding cases, we bind each thread to a particular core by inserting an invocation of the system function "pthread_setaffinity_np" into each benchmark at the point where threads are created. In the non-binding case, we rely on the default Linux scheduler to schedule the threads; the scheduler periodically migrates threads to maintain load balance if necessary. It is important to note that as mentioned in Section 2.2, in both binding and non-binding cases, when the number of threads is smaller than the total number of cores in a machine, we use the "taskset" command in Linux to specify which set of cores to use.

The results indicate that binding makes the programs perform much more stably than non-binding, reducing performance variations by as much as a factor of 122. We use the average times in the binding case to normalize the running times in the non-binding cases, and plot the 4-thread (2 sibling cores per chip on *native* input) results in Figure 6. The heights of the boxes show the large variations of the non-binding running times. Most boxes are above 1, indicating that binding makes the programs run faster than non-binding. We observe the similar phenomena on other configurations, despite the changes in the number of threads, core sets, and inputs.

The observed effects are mainly because the binding reduces cache thrashing and thread migrations. On the other hand, binding may hurt load balance, but that effect is not obvious for those programs due to the uniformity of the threads. This experiment, besides justifying the use of binding in the following explorations, also suggests that, similar to prior observations on traditional SMP (Symmetric multiprocessing) machines, the binding may serve as a strategy for the performance improvement of multithreaded applications running on CMP, despite the presence of shared cache.

Short Summary This section has shown that due to the large working sets and the limited inter-thread data sharing of the multithreaded programs, cache sharing has insignificant (either constructive or destructive) influence on the performance of the programs. Furthermore, we reveal that adjusting the placement of threads on cores has limited potential for performance enhancement of the programs. The main reason is the uniform relations among parallel threads, which mismatches with the non-uniform cache sharing on CMP machines. These conclusions, drawn from the extensive measurements, appear to hold across inputs, number of threads, sets of cores, and architectures.

4. Program-Level Transformation

Although the previous section reports insignificant influence of cache sharing for the performance of PARSEC programs, we maintain that the results do not suggest that cache sharing is a factor ignorable in the optimization of the execution of those programs. The implication is actually the opposite: Cache sharing deserves more attention especially in program transformations.

The conclusion comes from a set of experiments, in which, we transform several programs to make them better match the non-uniform cache sharing on CMPs. Our experiments concentrate on three representative programs. The transformations on them share the same theme, which is to increase the data sharing among sibling threads (but not among threads to run on different chips). This section first uses *streamcluster* as an example to explain the transformations in detail, and then reports the results on other programs at the end.

4.1 *Streamcluster*

The program, *streamcluster*, is a data-mining program that clusters a stream of data points. A major step in it is to take a chunk of array points and calculate their distances to a center point. This task occurs many times and accounts for the majority of the program's running time.

Transformation To highlight the transformation, we use the simplified pseudo-code in Figure 7 for the explanation, and assume there are 2 cores per chip.

The original version of the program is outlined in Figure 7 (a). Each of the threads computes the distances of a chunk of data to the center points. The variables *T1_start, T1_end* represent the start and end of the data chunk assigned for *Thread 1*, and *T2_start, T2_end* corresponding to *Thread 2*. The outer loop iterates over every candidate cluster center, and the inner loop iterates over every data point in a chunk. The function *cal_dist* computes the distance between a point and a candidate center.

Figure 7 (b) illustrates a transformation toward improving the matching between the program and shared cache on CMP. It tries to enhance the data sharing among sibling threads by letting them compute the distances from the same chunk of data points (e.g., thread 1 & 2 on data from *T1_start* to *T2_end*) to two different center points. The chunk size becomes twice as large as before. The computed distances are stored into two temporary arrays for later uses. (The use of temporary arrays is necessary to circumvent some loop carried dependencies[2].) With this transformation, the data sharing among threads becomes non-uniform: For instance, thread 2 shares substantially more data with thread 1 than with thread 3. When sibling threads co-run on a CMP processor, they would form synergistic prefetching with one another. One thread can use the data point brought into the shared cache by the other thread.

We notice that one may improve data locality inside a thread using traditional unroll-and-jam transformation [1]. The transformed code is shown in Figure 7 (c). (In our implementation, the inner loop is staged to circumvent loop carried dependencies.) In one iteration of the inner loop, each thread computes the distances between a point and two centers, increasing the reuse of the loaded data points. The increase of data reuse is similar as the previous transformation, except that it is inside a thread rather than between threads. Although the two transformations may be applied at the same time, we use this version as a second baseline besides the original version for highlighting the importance of shared-cache-aware transformations for programs running on CMP.

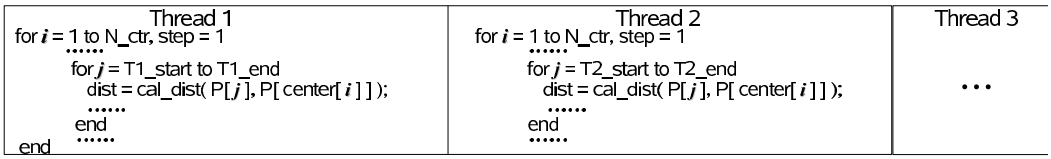
Performance Figure 8 reports the speedup of the two transformed versions over the original version on both the Intel and the AMD machines using the *native* input. On the AMD machine, as four cores share an L3 cache, we let 4 threads cooperate together in the inter-thread transformation. Correspondingly, we unroll the loop 4 times in the intra-thread transformation. On both Intel and AMD machines, the threads are assigned to cores in such a way that sibling threads co-run together.

Both the inter-thread and intra-thread transformations need to do store operations on the temporary arrays. However, as shown in Figure 8, even with that overhead, the inter-thread transformation brings 10–33% speedup compared to the original program. The benefits mainly come from the significant reduction of shared-cache miss rates and bus contention. The black bars in Figure 9 show the reduction on the Intel machine.

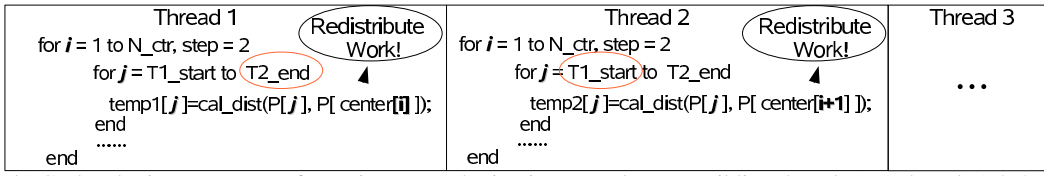
In contrast, the intra-thread transformation shows no clear enhancement to the program performance. The transformation mainly improves the usage of L1 cache but not the shared cache. As indicated by the third and fourth bars in every bar group in Figure 9, the transformation shows almost no reduction to the shared-cache miss rates and bus contention. The benefits on L1 cache usage turn out to be not significant enough to clearly offset the extra overhead. We have explored other several unrolling levels but have seen no clear improvement. The large slowdown on the AMD machine is due to the increased remote memory accesses caused by the use of temporary arrays.

This experiment demonstrates the importance of shared-cache-aware program transformations. We stress that the exploitation of the transformations often requires the cooperation from thread schedulers. The second bar in each bar group in Figure 9 shows the result when sibling threads are placed on non-sibling cores. The

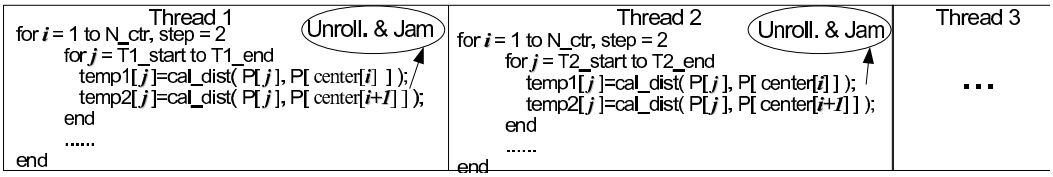
[2] Inside the inner loop, after *cal_dist*, there is an update to a data structure corresponding to the point *P[j]*, which is then used in the computation following the inner loop, causing loop carried dependencies



(a) Original Version (cache-sharing-oblivious)



(b) Cache-sharing-aware transformation. Data sharing increases between sibling threads (e.g. threads 1 & 2), but not across sibling pairs (e.g. threads 2 & 3).



(c) Traditional unroll-and-jam (cache-sharing-oblivious). Intra-thread data locality increases.

Figure 7. Simplified pseudo-code illustrating the original and optimized versions of the kernel computation, the function *pgain()*, in the program *streamcluster*. It is assumed that two threads constituting a sibling thread group, which will run on the same chip.

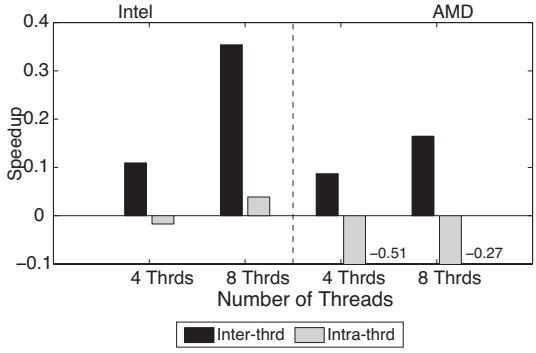


Figure 8. Speedup by the inter-thread and intra-thread transformations.

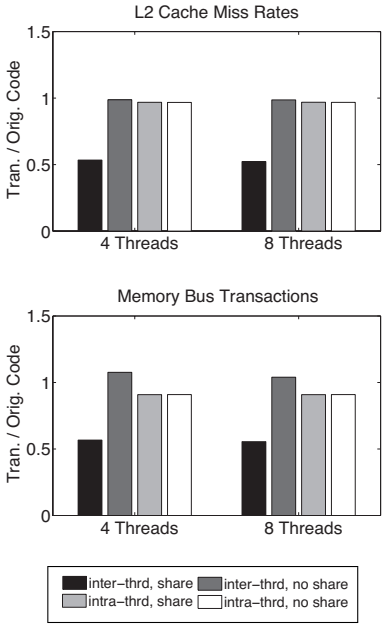


Figure 9. The reduction of L2 cache miss rates and memory bus contention on the Intel machine. The left two bars in each bar group respectively correspond to the cases when the sibling threads of the inter-thread optimized version run on two cores sharing or not sharing cache. The right two bars in each group correspond to the intra-thread optimized versions (using the same thread placement as in the inter-thread case.)

results demonstrate that the shared-cache-aware program transformation creates opportunities to better exert the power of thread co-scheduling or clustering.

4.2 *Blackscholes* and *Bodytrack*

The program, *blackscholes*, is a financial application. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. Because there is no close-form expression for the equation, the program uses numerical computation [3].

The input data file of this benchmark includes an array of options. The program computes the price for each of the options based on the five input parameters in the dataset file. The upper bound of the outermost loop in the program controls the number of times the options need to be priced. There is no inherent dependencies between two iterations of the loop. In the original program, the parallelization occurs inside the loop. In each iteration, the options

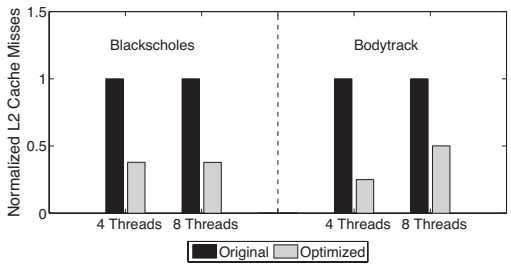


Figure 10. The reduction of L2 cache misses of *blackscholes* and *bodytrack* due to cache-sharing-aware transformation. The Intel machine and *native* inputs are used.

are first evenly partitioned into n (n for the number of threads) chunks. Each chunk is then processed by one thread, which prices the options in the chunk one after one by solving the Black-Scholes equation.

The transformation we apply is similar to the one on *streamcluster*. After the transformation, sibling threads process the same chunk at the same time; their executions correspond to a number of adjacent iterations of the outermost loop.

We observe that the transformation significantly reduces the number of misses on the shared cache on the *native* input, as shown in the left part of Figure 10. However, the program running times have no considerable changes. The document of the benchmark (the README file in the package) mentions that "the limiting factor lies with the amount of floating-point calculation a processor can perform." Through reading the program, we confirm that the program is a compute-bounded application—after reading an option data, the program conducts a significant amount of computation to solve the Black-Scholes equation with only local variables referenced. For further confirmation, we artificially reduce the amount of computation of the kernel in both the original and optimized programs. The optimized program starts showing clear speedup.

The program, *bodytrack*, tracks the 3D pose of a human body through an image sequence using multiple cameras. The algorithm uses an annealed particle filter to track the body pose using edges and foreground segmentation as image features, based on a 10 segment 3D kinematic tree body model. The number of particles and annealing layers are supplied as command line arguments. The program has both OpenMP and Pthread versions; we use the Pthread version.

The program processes frame by frame, and every frame consists of multiple camera images. The program has mainly two parallelized kernels *CreateEdgeMap* and *CalcWeights*. We make sibling cores share workload of the same image and non sibling cores on different images in the procedure *CreateEdgeMap*, resulting in a 15% speedup with 8 threads on Intel machine. We also increase the chance of true data sharing for the *CalcWeights* by redistributing the comparison workload for edge maps and foreground segment maps, resulting in a 5% speedup with 8 threads on Intel machine. The last level cache misses are significantly reduced. We provide the normalized last level cache miss reduction in the right part of Figure 10.

Overall, the experiments demonstrate that after the transformations, cache sharing starts to show its influence, and the placement of threads on cores becomes important for the programs performance. The observations suggest the importance of program-level transformations for improving the usage of shared cache. On the other hand, they further confirm that the uniform relations among threads in the original programs is one of the main causes for the limited influence of cache sharing.

5. Related Work

Cache sharing exists in both SMT (Simultaneous Multithreading) and CMP architectures. Its presence has drawn lots of research interest, especially in architecture design and process/thread scheduling in OS.

In architecture research, many studies (e.g., [5, 15, 22, 14, 17]) have proposed different ways to design shared cache to strike a good tradeoff between the destructive and constructive effects of cache sharing. These studies, although containing some examination of the influence of shared cache, mainly focus on the hardware design. Their measurements are on simulators and cover limited factors on the program or OS levels.

In OS research, the main focus on shared cache has been job co-scheduling including thread clustering. Many job co-scheduling studies [8, 24, 9, 7, 20, 6, 21], are on multiprogramming environments, attempting to alleviate shared-cache contention by placing independent jobs appropriately. Some of them include parallel programs in the job set, but the main focus is on inter-program cache contention rather than the influence of shared cache on parallel threads. Tam and others [23] propose thread clustering to group threads that share many data to the same processor through runtime hardware performance monitoring. They concentrate on server programs.

Some studies on workload characterization and performance measurement are relevant to this current work. Bienia and others [2, 3] have shown a detailed exploration of the characterization of the PARSEC benchmark suite on CMP. Because their goal is to expose architecture independent, inherent characteristics of the benchmarks, their measurement runs on *simlarge* input only, and uses a CMP simulator rather than actual machines. Liao and others [12] examine the performance of OpenMP applications on a Sun Fire V490 machine with private cache only. Tuck and Tullsen [25] have measured the performance of SPLASH-2 when 2 threads corun on a SMT processor.

Our work is distinctive in that it examines the influence of cache sharing in CMP on multithreaded programs in a *comprehensive* manner. It explores the numerous factors on program, OS, and architecture levels at the same time, employs modern CMP machines and contemporary multithreaded benchmarks. The systematic examination of the various facets of the problem is vital for avoiding biases that partial explorations may have, and thus improving the understanding of the influence of cache sharing on CMP.

We have found only few studies on exploiting program transformations for the improvement of shared cache usage, which echoes a prior observation [16]. Tullsen and others [16, 10] have proposed compiler techniques to change data and instructions placement to reduce cache conflicts among independent programs. Nikolopoulos [13] has examined a set of manual code and data transformations for improving shared cache performance on SMT processors. The inter-thread transformation described in Section 4 may share some similarity with traditional locality optimization on NUMA architectures for main memory usage [11]. Although both try to redistribute computation and data, our transformation aims to promote synergistic prefetching across threads, rather than increase data accesses to local memory banks.

6. Conclusion

In this work, we conduct a series of experiments on Intel and AMD CMP architectures to systematically examine the influence of cache sharing on the performance of modern multithreaded programs. The experiments cover a spectrum of factors related to shared cache performance on various levels. The multidimensional measurement shows that on both CMP architectures and for all the thread numbers and inputs we use, shared cache on CMP has in-

significant influence on the performance of most multithreaded applications in the benchmark suite. The implication, however, is not that cache sharing has no potential to be explored for the execution of such multithreaded programs, but that the current development and compilation of parallel programs must evolve to be cache-sharing-aware. The point is reinforced by three case studies, showing that significant potential exists for program-level transformations to enhance the matching between multithreaded applications and CMP architectures, suggesting the need for further studies on cache-sharing-aware program development and transformations.

Acknowledgments

We owe the anonymous reviewers our gratitude for their helpful comments on the paper. We thank Jie Chen and Michael Barnes at DoE Thomas Jefferson National Accelerator Facility for their valuable support on setting up experimental platforms. This material is based upon work supported by the National Science Foundation under Grant No. 0720499 and 0811791 and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or IBM.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, 2001.

[2] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 47–56, 2008.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[4] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A portable interface to hardware performance counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

[5] J. Chang and G. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, 2007.

[6] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings of the International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[7] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, 2007.

[8] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 220–229, October 2008.

[9] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of The International Conference on High Performance Embedded Architectures and Compilation (HiPEAC)*, 2010. (to appear).

[10] R. Kumar and D. Tullsen. Compiling for instruction cache performance on a multithreaded architecture. In *Proceedings of the International Symposium on Microarchitecture*, pages 419–429, 2002.

[11] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 140–147, 1993.

[12] C. Liao, Z. Liu, L. Huang, and B. Chapman. Evaluating OpenMP on chip multithreading platforms. In *Proceedings of International Workshop on OpenMP*, 2005.

[13] D. Nikolopoulos. Code and data transformations for improving shared cache performance on smt processors. In *Proceedings of the International Symposium on High Performance Computing*, pages 54–69, 2003.

[14] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*, pages 423–432, 2006.

[15] N. Rafique, W. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 2–12, 2006.

[16] S. Sarkar and D. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *Proceedings of The HiPEAC International Conference on High Performance Embedded Architectures and Compilation*, pages 353–368, 2008.

[17] A. Settle, J. L. Kihm, A. Janiszewski, and D. A. Connors. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 63–73, 2004.

[18] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176, 2004.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[20] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 66–76, 2000.

[21] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2002.

[22] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 117–128, 2002.

[23] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. *SIGOPS Oper. Syst. Rev.*, 41(3):47–58, 2007.

[24] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of ACM Computing Frontiers*, pages 41–50, 2009.

[25] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[26] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, 1995.