The Significance of CMP Cache Sharing on Contemporary Multithreaded Applications

Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen, *Member*, *IEEE*

Abstract—Cache sharing on modern Chip Multiprocessors (CMPs) reduces communication latency among corunning threads, and also causes interthread cache contention. Most previous studies on the influence of cache sharing have concentrated on the design or management of shared cache. The observed influence is often constrained by the reliance on simulators, the use of out-of-date benchmarks, or the limited coverage of deciding factors. This paper describes a systematic measurement of the influence with most of the potentially important factors covered. 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 in the PARSEC benchmark suite, regardless of the types of parallelism, input data sets, architectures, numbers of threads, and assignments of threads to cores. After a detailed analysis, we find that the main reason is the mismatch between the software design (and compilation) of multithreaded applications and CMP architectures. By performing source code transformations on the programs in a cache-sharing-aware manner, we observe up to 53 percent performance increase when the threads are placed on cores appropriately, confirming the software-hardware mismatch as a main reason for the observed insignificance of the influence from cache sharing, and indicating the important role of cache-sharing-aware transformations—a topic only sporadically studied so far—for exerting the power of shared cache.

Index Terms—Shared cache, thread scheduling, parallel program optimizations, chip multiprocessors.

1 Introduction

Most modern Chip Multiprocessors (CMPs) feature on-chip cache sharing. On a system with multiple chips, the sharing further shows nonuniformity: cores on different chips typically do not share cache as the cores in a chip do.

The sharing is a double-edged sword. It may cause destructive cache contention: data accesses by corunners (processes or threads running on sibling cores) may conflict in the shared cache, causing cache thrashing. On the other hand, it may be constructive: corunners may directly communicate through shared cache with lower latency than cross-chip communications, and one thread may access the data that other threads have brought into the shared cache, forming synergistic prefetching.

The importance of using shared cache effectively has recently drawn much attention. For example, cache-sharing-aware scheduling in operating systems (OS) research has shown that a suitable assignment of corunning processes to cores may alleviate the cache contention among corunners. Considerable performance improvements have been observed on sets of independent jobs [10], [11], [25], [29] as well as parallel threads inside certain classes of single applications [28].

However, in this work (Sections 2 and 3), through a systematic measurement, we find that contrary to the commonly perceived significant effects, cache sharing has very limited influence, neither positive nor negative, on the performance of the applications in PARSEC—a modern 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 that consists of thousands of runs and covers various potentially important factors of programs (number of threads, parallel models, phases, and input data sets), 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.

A detailed analysis uncovers the fundamental reason for the observed insignificance: the development and the currently standard compilation of the programs are oblivious to cache sharing, hence causing a mismatch between the generated programs and the CMP cache architecture. The mismatch exhibits in 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 nonuniform cache sharing on CMPs, explaining the insensitivity of the program performance to 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

• *The authors are with the Department of Computer Science, The College of William and Mary, PO Box 8795, Williamsburg, VA 23187. E-mail: {eddy, jiang, xshen}@cs.wm.edu.*

Manuscript received 3 Mar. 2010; revised 14 Dec. 2010; accepted 6 Mar. 2011; published online 4 Apr. 2011.
Recommended for acceptance by R. Bianchini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2010-03-0139.
Digital Object Identifier no. 10.1109/TPDS.2011.130.

than the shared cache. The difference between the sharing and nonsharing cases in terms of cache size per thread is not enough to make significant changes in cache misses. Hence, cache contention shows no obvious effects either.

The second part of this paper (Section 4) explores the implications of the observed insignificance. At the first glance, it seems to suggest that exploitation of cache sharing is unimportant for the executions of the multithreaded applications, but a set of experiments demonstrates the exact opposite conclusion. Exploiting cache sharing has significant potential, but to realize the potential, it is critical to apply cache-sharing-aware transformations.

In the experiments, we increase the amount of shared data among sibling threads (the threads sharing the same cache) through certain code transformations. The transformations yield nonuniform data sharing among threads, matching with the nonuniform 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 reduces cache misses by over 50 percent and improves performance by up to 53 percent, compared to other placements and the original programs.

In the third part of this paper (Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130), based on a careful examination of an example program, *streamcluster*, we investigate issues related to the automation of cache-sharing-aware transformations. This includes the principles, challenges of optimizing cache performance, as well as the capabilities the automatic optimizers ought to have, and the possible roles of programmers for the optimization.

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 [10], [11], [12], [25], [29] are concentrated on coruns 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 [30] have used old benchmark suites (e.g., SPLASH-2 [31]), or have concentrated on a specific class of applications, such as server programs [28]; some [16] 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—the plausible reason for the departure between the observations made in this work and the previous.

Similar to the observation made by Sarkar and Tullsen [20], we have found only a small number of studies [15], [17], [20] 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). The importance of program transformations demonstrated in this work will hopefully spur more research efforts in this direction.

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

This section introduces the benchmark suite, the factors we study and the rationales, the measurement schemes, and the statistic techniques for data analysis.

2.1 Benchmarks

The selected benchmark suite is PARSEC v1.0, a suite released in 2007 for CMP research [3]. It includes emerging applications in recognition, mining and synthesis, as well as systems applications that mimic large-scale multithreaded commercial programs. Studies [2], [3] 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 appealing over some old parallel benchmark suites such as SPLASH-2 [31]. Table 1 lists the 10 programs we use and their working set sizes (on *simlarge* inputs). Programs *dedup* and *ferret* are both pipelining applications with a dedicated pool of threads for each pipeline stage. Programs *facesim*, *fluidanimate*, and *streamcluster* have streaming behaviors. Other programs are data-level parallel programs with various synchronizations and inter-thread communications. All the programs use Pthreads API, and employ standard Pthreads schemes (locks and barriers) for synchronizations. An exception is *canneal*, which uses an aggressive synchronization strategy based on data race recovery. We exclude two other programs, *vips* and *freqmine*, because their non-Pthread implementations cause difficulties for our tool to bind their threads with processors.

2.2 Factors

To achieve a comprehensive understanding on how much cache sharing influences the performance of multithreaded applications, our experiments include a number of factors that are potentially important for the influence. This section briefly introduces these factors and the rationale for selecting them. The next section elaborates on the treatment of these factors in the systematic measurement.

As shown in Table 2, the considered factors come from the program, OS, and architecture levels. (The boldface words correspond to the dimensions in Table 2.)

- *Program level.* The major factors in this level include the **input** data sets 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

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 Appendix B.3, which can be found on the Computer Society Digital Library).*

of cache contention. We use four input data sets coming with PARSEC. Table 2 lists them in increasing order of size. The number of threads varies from 1 to 8. The third factor, parallel models, determines the patterns of data sharing and computation.

- *OS level.* The main effect from the OS is thread scheduling, which determines the corunners 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 various corunning scenarios, as detailed in Section 3. Because the experiment needs binding threads to cores, we examine the effects of **binding** by comparing to nonbinding cases (Appendix B.4, which can be found on the Computer Society Digital Library).
- *Architecture level.* The types of machines we use include a Dell PowerEdge 2950 server hosting two quad-core Intel Xeon E5310 processors, and a Dell PowerEdge R80 hosting two AMD Opteron 2352 processors. They represent 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 Nonuniform Memory Access (ccNUMA) CMP with HyperTransport links and an exclusive cache hierarchy.[1] Both machines run Linux 2.6.22 with GCC4.2.1 installed. Table 3 reports their details.

  When the number of threads is smaller than the total number of cores in a machine (8 in our experiments), the threads may be assigned to different **subsets of cores**. We experiment with up to seven (depending on the number of threads) different sets to cover most representative sharing scenarios. In the case of two threads on the Intel machine, for instance, the sets of cores we use include two sibling cores that share cache, two nonsibling cores on a single chip which share the same memory-processor bus, and two cores residing on different chips. The 4-thread case has three corresponding sets. The 8-thread case has only one set, the set of all cores.

Program phase changes may affect the measurement results, especially on the measured potential of thread scheduling. Appendix B.2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130, will show how this factor is examined in our experiments.

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

TABLE 3
Machine Configurations

	CPU	L1	L2	L3	Memory
Intel	Xeon E5310 1.6GHz quad-core	32KB	2x4MB, shared	None	8GB
AMD	Opteron 2352 2.1GHz quad-core	64KB	512KB	2MB, shared	8GB cc-NUMA

2.3 Measurement Schemes

Our measurement concentrates on running times, cache miss rates, and 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. Besides using the average performance of the repetitive runs, we employ the statistical analysis (described in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130) to prevent measurement noise from causing possibly biased conclusions.

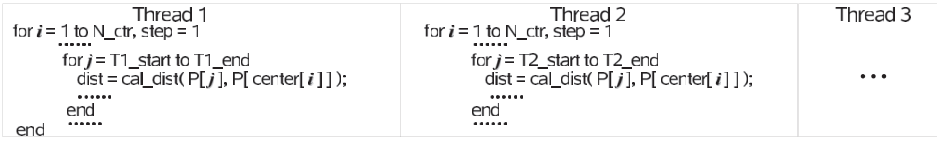
3 Measurement and Findings

In this section, we summarize some major findings of our systematic measurements. Appendix B contains the details, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130.

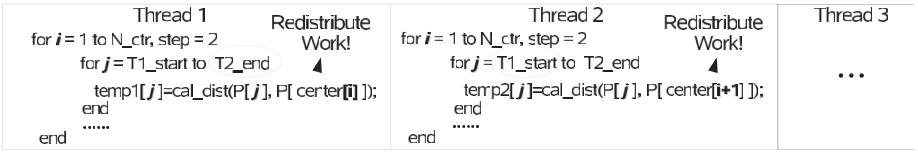
We find that, contrary to commonly perceptions, cache sharing has insignificant (neither constructive or destructive) influence on the performance of the programs. The main reasons are the large working sets and the limited interthread data sharing of the multithreaded programs. Furthermore, we reveal that adjusting the placement of threads on cores has limited potential for performance enhancement of the programs. It is because of the uniform relations among parallel threads, which mismatches with the nonuniform 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

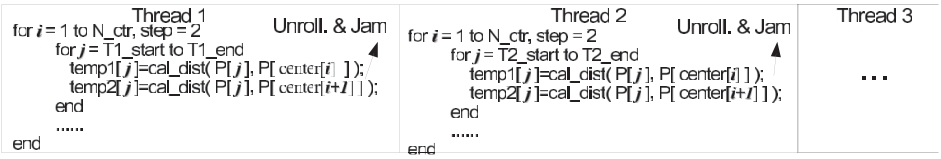
Although the previous section reports insignificant influence of cache sharing for the performance of PARSEC programs, we maintain that the results do not suggest that cache sharing is a factor ignorable in the optimization of the execution of those programs. The implication is actually the



(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.

Fig. 1. Simplified pseudocode illustrating the original and optimized versions of the function *pgain()* in *streamcluster*. It is assumed that two threads constitute a sibling group that share cache.

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 nonuniform cache sharing on CMPs. The transformations are manual; Appendix D discusses the automation of such transformations, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130.

Our experiments concentrate on four representative programs. The transformations on them share a single theme: to increase the data sharing among sibling threads but not other threads. This section uses *streamcluster* as an example to explain the transformations in detail, and then reports the results on other programs.

4.1 *Streamcluster*

The program, *streamcluster*, is a data-mining program that clusters a stream of data points. One part of the program takes a chunk of array points and calculates their distances to a center point. This calculation occurs many times and accounts for a major part of the program's running time.

4.1.1 Transformation

To highlight the transformation, we use the simplified pseudocode in Fig. 1 for the explanation, and assume there are two cores per chip.

The original version of the program is outlined in Fig. 1a. 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*, *T2_start*, *T2_end* for *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.

Fig. 1b illustrates a transformation for improving the matching between the program and CMP shared cache. 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., threads 1 and 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 nonuniform: for instance, thread 2 shares substantially more data with thread 1 than with thread 3. When sibling threads corun 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 Fig. 1c. (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 to the previous transformation, except that it is inside a thread rather than between threads. The intrathread and interthread transformations are complementary to each other. They can be applied to a program at the same time. In the next section, we report how the interthread transformation benefits the program both without and with the intrathread optimizations.

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.

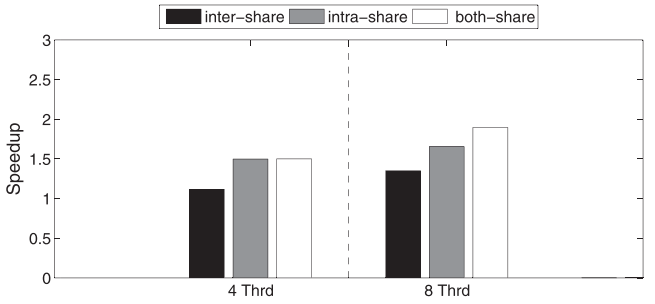


Fig. 2. Speedup by interthread, intrathread, and combined transformations on the Intel machine. Sibling threads share L2 cache.

4.1.2 Performance

Fig. 2 shows the speedup brought by the transformations on the Intel machine. In all these runs, we assign sibling threads to adjacent cores with L2 cache shared. Even though both the interthread and intrathread transformations add extra store operations to the temporary arrays, the results show that their benefits outweigh the overhead substantially. An examination of the source code shows the reason. Each point involved in the distance calculation is of 128 dimensions. As a result, the temporary arrays weight only a small portion of the entire working set.

One may notice that the benefits from the interthread transformation are not as significant as those from the intrathread transformation.[3] It is because the intrathread transformation increases the hits in L1 cache, while the interthread transformation only benefits L2 usage. However, it is important to note that these two transformations are not competitors. As the "both-share" bars in Fig. 2 show, based on the code optimized through the intrathread transformation, the interthread transformation further improves the performance by 23 percent, demonstrating the complementary relations between these two kinds of transformation.

Fig. 3 reports the normalized L2 cache miss rates and the numbers of memory bus transactions. The performance of the original program is the baseline. In each group of bars, the "intershare" and "both-share" bars correspond to the cases when the interthread transformation is applied without and with the intrathread transformations, respectively. In both cases, the transformation reduces L2 cache miss rates and memory bus transactions substantially, confirming the benefits of the transformation for data locality enhancement despite whether intrathread optimizations are applied. We stress that the application of the transformations requires the cooperation from thread schedulers. The "inter-noshare" and "both-noshare" bars in Fig. 3 show the result when sibling threads are placed on nonsibling cores. The clear contrast with the other bars demonstrates that the shared-cache-aware program transformation creates opportunities to better exert the power of thread coscheduling or clustering.

The better performance by the combined transformation comes from the extra data reuses it creates in the shared L2 cache. It is tempting to think that the intrathread transformation of a factor of 4 yields the same amount of data reuse, and hence may produce similar performance as the combined transformation. Experiments show that the two transformations indeed produce similar performance on some inputs, but the combined transformation still excels on some other inputs, as exemplified in the second column of Table 4. In fact, on all reuse levels listed in Table 4, the combined transformation all outperforms the intrathread optimization. Hardware performance counters show that the code from combined transformations yields 27-50 percent fewer L1 cache misses and 8-32 percent fewer L2 cache misses than that from the corresponding intrathread transformations does. Source code analysis reveals that in the processing of one data point, the intrathread transformations entail references to as much as twice of data centers and temporary arrays over the corresponding combined case, hence the significantly more L1 cache conflicts (note, each data point is a 128-dimensional vector).

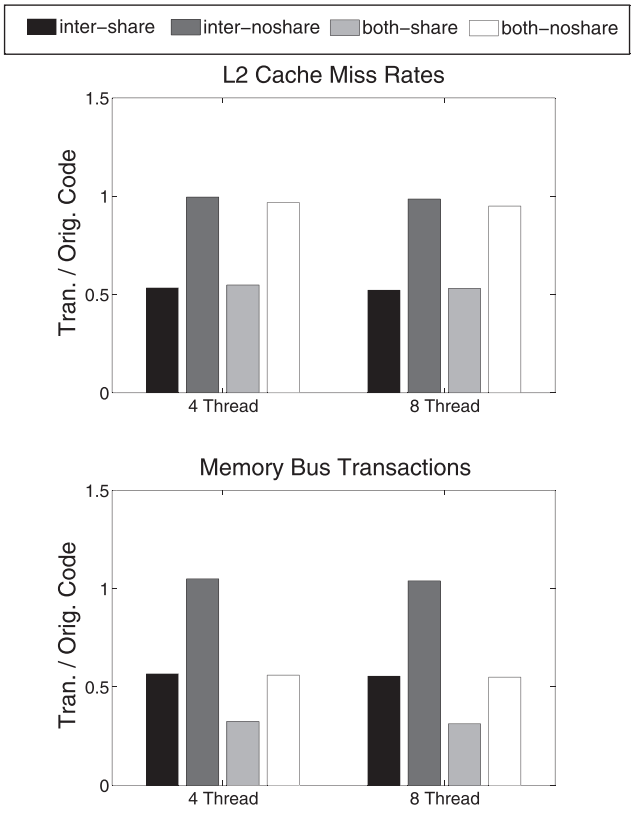


Fig. 3. The reduction of L2 cache miss rates and memory bus contention on the Intel machine. In each bar group, the first two correspond to interthread transformations, and the next two for the combined transformation. In each pair, the two bars correspond to the cases when sibling threads share L2 cache or not.

TABLE 4
Streamcluster Running Times

Factor of Reuse	4	8	16	32	64
Intra-thread opt. (s)	15.8	12.5	10.7	10.9	11.5
Combined opt. (s)	11.3	10.5	10.0	10.1	10.7

* machine: Intel; input: 10 20 128 100000 20000 5000

3. This result differs from our previous observations [32] because we reimplement the transformation, during which, we manage to remove some inefficiency in the intrathread transformed code, including the elimination of stores of some intermediate results and some references to assistant data structures.

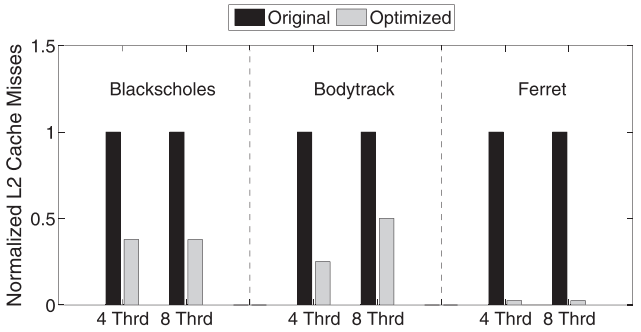


Fig. 4. The reduction of L2 cache misses due to cache-sharing-aware transformation. The Intel machine is used.

4.2 *Blackscholes*

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 closed-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 data set file. The upper bound of the outermost loop in the program controls the number of times the options need to be priced. There are no inherent dependencies between two iterations of the loop. In the original program, the parallelization occurs inside the loop. In each iteration, the options 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 Fig. 4. 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 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.

4.3 *Bodytrack*

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 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 nonsibling cores on different images in the procedure *CreateEdgeMap*, resulting in a 15 percent speedup with eight threads processing the *native* input on the 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 percent speedup with eight threads on the Intel machine. The last level cache misses are significantly reduced. We provide the normalized last level cache miss reduction in the middle part of Fig. 4.

4.4 *Ferret*

The program, *ferret*, is a pipeline program, implementing a search engine for image searching. Our transformation concentrates on the most memory intensive stage, the fourth stage. Appendix C describes a two-step transformation we apply, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130. This transformation creates a nonuniform relation among threads: sibling threads share similar data accesses in the same database section, but nonsibling ones do not. As shown in the right part of Fig. 4, the transformation eliminates most shared-cache misses, and yields a speedup of as much as 1.53.

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. They further confirm that the uniform relation among threads in the original programs is one of the main causes for the limited influence of cache sharing on performance.

In the experiments, all transformations are manual. Appendix D provides a discussion on the challenges and potential solutions for automating the transformations, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130.

5 Related Work

Cache sharing exists in both Simultaneous Multithreading (SMT) 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., [6], [18], [19], [22], [27]) have proposed different ways to design shared cache to strike a good trade-off 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 coscheduling and thread clustering. Many job coscheduling studies [7], [10], [11], [12], [25], [26], [29], [33], [34] 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 interprogram cache contention rather than the influence of shared cache on parallel threads. Tam et al. [28] propose thread clustering to group threads of server programs through runtime hardware performance monitoring. Ding et al. [9] have proposed

the use of OS support for cache partitioning to alleviate the contention in shared cache.

Some studies on workload characterization and performance measurement are relevant to this current work. Bienia et al. [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 et al. [16] examine the performance of OpenMP applications on a machine with private cache only. Tuck and Tullsen [30] have measured the performance of SPLASH-2 when two threads corun on an SMT processor.

Our work is distinctive in that it examines the influence of cache sharing in CMP on multithreaded programs in a comprehensive manner by exploring the manifold factors and employing modern machines and contemporary multithreaded benchmarks. The systematic examination of the various facets of the problem is vital for avoiding biases.

There are only a few studies that exploit program transformations for improving shared-cache usage. Tullsen et al. [15], [20] have proposed compiler techniques based on traditional cache-conscious data placement [5] to reduce cache conflicts among independent programs. Nikolopoulos [17] has examined a set of manual code and data transformations for improving shared-cache performance on SMT processors. We recently investigate the benefits of cross-thread array regrouping for locality enhancement in CMP [13]. Some recent studies [8], [14], [21] start to extend traditional locality models—such as reuse distance—to characterize data references in CMP platforms.

6 Conclusion

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

The authors thank Jie Chen and Michael Barnes at US Department of Energy (DOE) Thomas Jefferson National Accelerator Facility for their help on setting up experimental platforms. This material is based upon work supported by the US National Science Foundation (NSF) under Grant Nos. 0720499, 0811791, and 0954015, and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the US National Science Foundation (NSF) or IBM.

This paper is an extended version of a paper that received the Best Paper Award in the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '10). The extensions are mainly in three aspects. First, we introduce a set of statistical techniques into the analysis of the performance measurement (Appendix A, and part of Appendices B.1 and B.2 including Figs. 3 and 8 in Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130). By addressing the sometimes large fluctuations of the measured running times, these techniques filter out most random factors in the measurement, offering more conclusive results than before. Meanwhile, the results suggest that using average running times, as most existing studies do, is not rigorous enough for many parallel program performance analyses—the statistic analysis should be adopted, especially when the performance varies considerably across repetitive executions. Second, we extend the evaluation of the cache-sharing-aware transformations with a further investigation in the relation between intrathread optimizations and interthread transformations (Section 4.1.2), and the application of the transformation to a new benchmark (Section 4.4). Third, we add Appendix D, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.130, which, based on a careful examination of an example program, *streamcluster*, investigates issues related to the automation of cache-sharing-aware transformations, including the principles, challenges, the capabilities the automatic optimizers ought to have, and the possible roles of programmers for the optimization. In addition, we add some extra results (e.g., Appendix B.2, which can be found on the Computer Society Digital Library, Table 1, Figs. 10c and 10d) and enhance the presentation throughout the paper.

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 versus SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors," *Proc. IEEE Int'l Symp. Workload Characterization,* pp. 47-56, 2008.

[3] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 72-81, 2008.

[4] S. Browne, C. Deane, G. Ho, and P. Mucci, "PAPI: A Portable Interface to Hardware Performance Counters," *Proc. Dept. of Defense HPCMP Users Group Conf.,* 1999.

[5] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-Conscious Data Placement," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 139-149, 1998.

[6] J. Chang and G. Sohi, "Cooperative Cache Partitioning for Chip Multiprocessors," *Proc. 21st Ann. Int'l Conf. Supercomputing,* pp. 242-252, 2007.

[7] M. DeVuyst, R. Kumar, and D.M. Tullsen, "Exploiting Unbalanced Thread Scheduling for Energy and Performance on a Cmp of smt Processors," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2006.

[8] C. Ding and T. Chilimbi, "A Composable Model for Analyzing Locality of Multi-Threaded Programs," Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[9] X. Ding, J. Lin, Q. Lu, P. Sadayappan, and Z. Zhang, "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA),* pp. 367-378, 2008.

[10] A. Fedorova, M. Seltzer, and M.D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques,* pp. 25-38, 2007.

[11] Y. Jiang, X. Shen, J. Chen, and R. Tripathi, "Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques (PACT),* pp. 220-229, Oct. 2008.

[12] Y. Jiang, K. Tian, and X. Shen, "Combining Locality Analysis with Online Proactive Job Co-Scheduling in Chip Multiprocessors," *Proc. Int'l Conf. High Performance Embedded Architectures and Compilers (HiPEAC),* pp. 201-215, 2010.

[13] Y. Jiang, E. Zhang, and X. Shen, "Array Regrouping on Cmp with Non-Uniform Cache Sharing," *Proc. Int'l Workshop Languages and Compilers for Parallel Computing,* 2010.

[14] Y. Jiang, E. Zhang, K. Tian, and X. Shen, "Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors," *Proc. Int'l Conf. Compiler Construction,* 2010.

[15] R. Kumar and D. Tullsen, "Compiling for Instruction Cache Performance on a Multithreaded Architecture," *Proc. Int'l Symp. Microarchitecture,* pp. 419-429, 2002.

[16] C. Liao, Z. Liu, L. Huang, and B. Chapman, "Evaluating OpenMP on Chip Multithreading Platforms," *Proc. Int'l Workshop OpenMP,* 2005.

[17] D. Nikolopoulos, "Code and Data Transformations for Improving Shared Cache Performance on SMT Processors," *Proc. Int'l Symp. High Performance Computing,* pp. 54-69, 2003.

[18] M.K. Qureshi and Y.N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *Proc. Int'l Symp. Microarchitecture,* pp. 423-432, 2006.

[19] N. Rafique, W. Lim, and M. Thottethodi, "Architectural Support for Operating System-Driven CMP Cache Management," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques,* pp. 2-12, 2006.

[20] S. Sarkar and D. Tullsen, "Compiler Techniques for Reducing Data Cache Miss Rate on a Multithreaded Architecture," *Proc. Int'l Conf. High Performance Embedded Architectures and Compilers (HiPEAC),* pp. 353-368, 2008.

[21] D. Schuff, M. Kulkarni, and V. Pai, "Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 53-64, 2010.

[22] A. Settle, J.L. Kihm, A. Janiszewski, and D.A. Connors, "Architectural Support for Enhanced SMT Job Scheduling," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques,* pp. 63-73, 2004.

[23] X. Shen, Y. Zhong, and C. Ding, "Locality Phase Prediction," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 165-176, 2004.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 45-57, 2002.

[25] A. Snavely and D. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading processor," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 66-76, 2000.

[26] A. Snavely, D. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 66-76, 2002.

[27] G. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture,* pp. 117-128, 2002.

[28] D. Tam, R. Azimi, and M. Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors," *SIGOPS Operating Systems Rev.,* vol. 41, no. 3, pp. 47-58, 2007.

[29] K. Tian, Y. Jiang, and X. Shen, "A Study on Optimally Co-Scheduling Jobs of Different Lengths on Chip Multiprocessors," *Proc. ACM Conf. Computing Frontiers,* pp. 41-50, 2009.

[30] N. Tuck and D.M. Tullsen, "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 26-35, 2003.

[31] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. Int'l Symp. Computer Architecture,* pp. 24-36, 1995.

[32] E.Z. Zhang, Y. Jiang, and X. Shen, "Does Cache Sharing on Modern Cmp Matter to the Performance of Contemporary Multithreaded Programs?," *PPoPP '10: Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 203-212, 2010.

[33] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen, "Processor Hardware Counter Statistics as a First-Class System Resource," *Proc. 11th Workshop Hot Topics in Operating Systems,* 2007.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 129-142, 2010.

Eddy Zheng Zhang received the BS degree in electronic engineering from Shanghai Jiao Tong University and the MS degree in computer science from the College of William and Mary. She is working toward the PhD degree at the Department of Computer Science in the College of William and Mary. Her research interests include compilers and programming systems, GPU program optimization and performance tuning, multicore and many core, data locality and memory management, and performance evaluation.

Yunlian Jiang received the bachelor's and master's degrees from the University of Science and Technology of China in 2003 and 2006, respectively, both in computer science. He is working toward the PhD degree at the Computer Science Department in the College of William and Mary. His research interests focus on cache-contention-aware job coscheduling, program behavior profiling, and analysis for performance improvement.

Xipeng Shen received the master's and PhD degrees in computer science from the University of Rochester. He is an assistant professor of computer science at the College of William and Mary. He is an IBM CAS faculty fellow, and a recipient of the US National Science Foundation (NSF) CAREER Award. He received the Best Paper Award from ACM PPoPP 2010. His research in Compiler Technology and Programming Systems aims at helping programmers achieve high performance as well as good programming productivity on both uniprocessor and multiprocessor architectures. He is particularly interested in the effective usage of memory hierarchies, the exploitation of program inputs in program behavior analysis, and the employment of statistical learning in runtime systems and dynamic program optimizations. He is a member of the IEEE.