The Significance of CMP Cache Sharing on Contemporary Multithreaded Applications

Eddy Z. Zhang, Yunlian Jiang, Xipeng Shen

Abstract—Cache sharing on modern Chip Multiprocessors (CMP) reduces communication latency among co-running threads, but also causes inter-thread 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 The observed influence is often constrained by the reliance on simulators, the use of out-of-date benchmarks, or the limited coverage of deciding factors.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 datasets, 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% 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 (CMP) feature on-chip cache sharing. On a system with multiple chips, the sharing further shows non-uniformity: 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 co-runners (processes or threads running on sibling cores) may conflict in the shared cache, causing cache thrashing. On the other hand, it may be constructive: Co-runners 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 co-running processes to cores may alleviate the cache contention among co-runners. 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 (Section 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, 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-

- *The authors are with the Department of Computer Science, The College of William and Mary, VA, USA 23185. E-mail: {eddy,jiang,xshen}@cs.wm.edu. Xipeng Shen is an IEEE member.*
- *This article 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 Figures 7 and 11). 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 intra-thread optimizations and inter-thread transformations (Section 4.1.2), and the application of the transformation to a new benchmark (Section 4.4). Third, we add Appendix D, 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., Table 5, Figure 10(c,d)) and enhance the presentation throughout the paper.*

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 non-uniform 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 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. 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 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 reduces cache misses by over 50% and improves performance by up to 36%, compared to other placements and the original programs.

In the third part of this paper (Appendix D), 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 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 [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.

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.

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.

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 B.3).

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** 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 coming with PARSEC. Table 2 lists them in increasing order of size. The number of threads varies from one to eight. 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 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 various 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 (Section B.4).
- *Architecture Level* The types of machines we use include a Dell PowerEdge 2950 server hosting 2 quad-core Intel Xeon E5310 processors, and a Dell PowerEdge R80 hosting 2 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 None-Uniform 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 detail.
  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 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.

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

Program phase changes may affect the measurement results, especially on the measured potential of thread scheduling. Appendix B.2 will show how this factor is examined in our experiments.

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

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

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) 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.

We find that, contrary to commonly perceptions, cache sharing has insignificant (either constructive or destructive) influence on the performance of the programs. The main reasons are the large working sets and the limited inter-thread 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 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. The transformations are manual; Appendix D discusses the automation of such transformations.

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 pseudo-code in Figure 1 for the explanation, and assume there are 2 cores per chip.

The original version of the program is outlined in Figure 1(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*, *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.

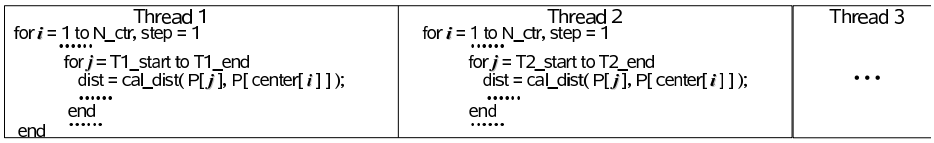
Figure 1(b) illustrates a transformation for improving the matching between the program and CMP shared cache. It tries to enhance the data sharing among sibling threads by letting them compute the distances from the same chunk of data points (e.g., thread 1 & 2 on data from *T1_start* to *T2_end*) to 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 1(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 to the previous transformation, except that it is inside a thread rather than between threads. The intra-thread and inter-thread 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 inter-thread transformation benefit the program both without and with the intra-thread optimizations.

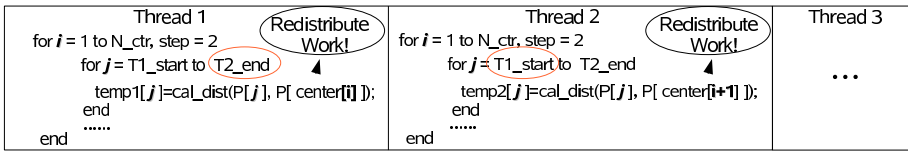
4.1.2 Performance

Figure 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 inter-thread and intra-thread transformations add extra store operations to the temporary arrays, the results show that their benefits

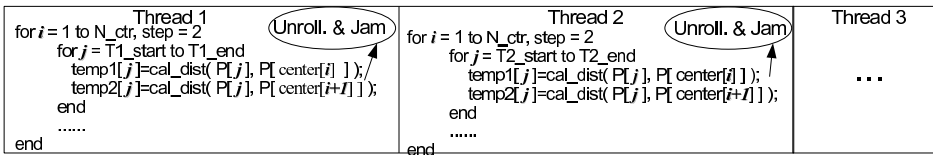
2. Inside the inner loop, after *cal_dist*, there is an update to a data structure corresponding to the point *P[j]*, which is then used in the computation following the inner loop, causing loop carried dependencies



(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.

Fig. 1. Simplified pseudo-code 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.

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

Figure 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 "inter-share" and "both-share" bars correspond to the cases when the inter-thread transformation is applied without and with the intra-thread 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 intra-thread optimizations are applied.

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

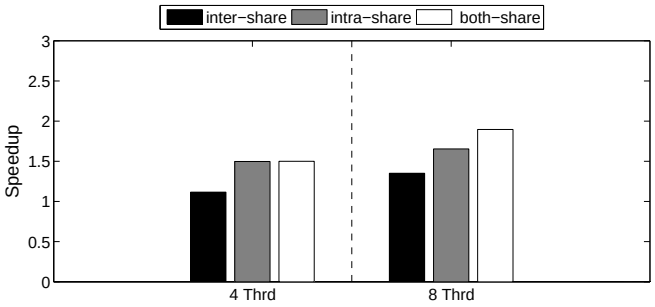


Fig. 2. Speedup by inter-thread, intra-thread, and combined transformations on the Intel machine. Sibling threads share L2 cache.

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

We stress that the application of the transformations requires the cooperation from thread schedulers. The "inter-noshare" and "both-noshare" bars in Figure 3 shows the result when sibling threads are placed on non-sibling 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 co-scheduling 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 intra-

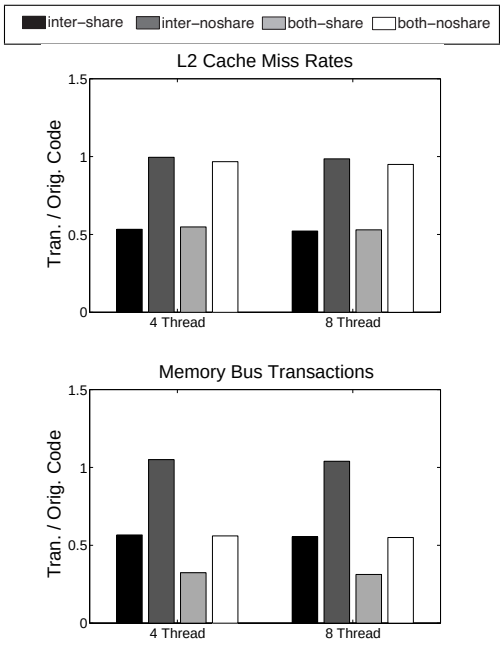


Fig. 3. The reduction of L2 cache miss rates and memory bus contention on the Intel machine. In each bar group, the first two correspond to inter-thread transformations, and the next two for the combined transformation. In each pair, the two bars correspond to the cases when sibling threads share L2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

thread transformation of a factor of four 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 intra-thread optimization. Hardware performance counters show that the code from combined transformations yields 27–50% fewer L1 cache misses and 8–32% fewer L2 cache misses than that from the corresponding intra-thread transformations does. Source code analysis reveals that in the processing of one data point, the intra-thread 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-dimension vector).

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 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

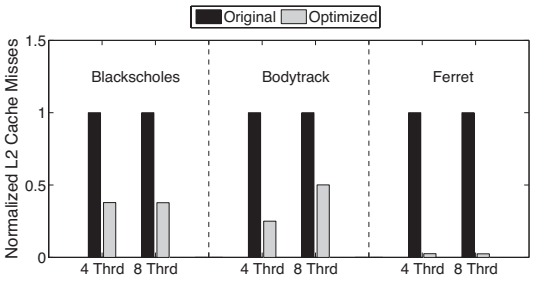


Fig. 4. The reduction of L2 cache misses due to cache-sharing-aware transformation. The Intel machine is used.

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 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 Figure 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 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.

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 non-sibling cores on different images in the procedure *CreateEdgeMap*, resulting in a 15% speedup with 8 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% speedup with 8 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 Figure 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. This transformation creates a non-uniform relation among threads: Sibling threads share similar data accesses in the same database section, but non-sibling ones do not. As shown in the right part of Figure 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.

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., [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 co-scheduling and thread clustering. Many job co-scheduling 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 inter-program cache contention rather than the influence of shared cache on parallel threads. Tam and others [28] propose thread clustering to group threads of server programs through runtime hardware performance monitoring. Ding and others [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 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 [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 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 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 and others [15], [20] have proposed compiler techniques based on traditional cache-concious 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 stud-

ies, 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 thank Jie Chen and Michael Barnes at DoE Thomas Jefferson National Accelerator Facility for their help on setting up experimental platforms. This material is based upon work supported by the National Science Foundation under Grant No. 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 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] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.

[6] 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.

[7] M. DeVuyst, R. Kumar, and D. M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, 2006.

[8] C. Ding and T. Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical report, Microsoft Research, MSR-TR-2009-107, 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. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 367–378, 2008.

[10] 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.

[11] 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.

[12] 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)*, pages 201–215, 2010.

[13] Y. Jiang, E. Zhang, and X. Shen. Array regrouping on cmp with non-uniform cache sharing. In *Proceedings of International Workshop on 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? In *Proceedings of the International Conference on Compiler Construction*, 2010.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] D. Schuff, M. Kulkarni, and V. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 53–64, 2010.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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*, pages 66–76, 2002.

[27] 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.

[28] 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.

[29] 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.

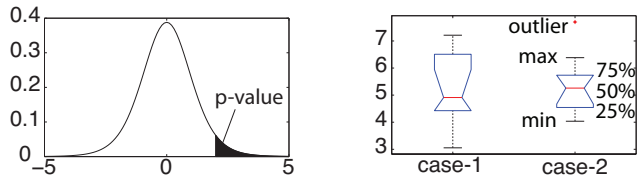
[30] N. Tuck and D. M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 26–35, 2003.

[31] 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*, pages 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? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 203–212, 2010.

[33] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor hardware counter statistics as a first-class system resource. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, 2007.

[34] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, pages 129–142, 2010.



(a) T-distribution (b) Boxplot

Fig. 5. Statistical analysis for performance comparison.

APPENDIX A
STATISTICAL ANALYSIS

As well known, the running time of a program is subject to many random factors in the running environments and software execution stacks, which lead to sometimes significant variations in execution times of repetitive runs. A solution often adopted in previous studies is to repeat the execution and use the average running time. This solution works only when the variation is small. Executions of parallel programs, however, often exhibit large variations, especially when they run on a platform equipped with distributed memory like the AMD machine used in this study. Five repetitive runs of *streamcluster* under a single configuration, for instance, have running times ranging from 376s to 446s.

We employ statistical hypothesis testing to address this problem. As an example, suppose our objective is to determine whether two configurations, A and B, of a program differ significantly in their performance. In statistics, the objective is equivalent to checking whether the *real* mean running times of the two configurations are different. The real mean differs from average running times: The former is the statistical expectation of all possible running times, reflecting the inherent performance of the program; while the latter is just the average of some observed running times. In the following explanation, we use $S_a = \{a_1, a_2, \cdots, a_n\}$ and $S_b = \{b_1, b_2, \cdots, b_m\}$ to represent the sets of observed running times of the two configurations respectively.

The hypothesis testing starts with a *null hypothesis*, which is to assume that the two configurations have the same mean running time. Under the assumption, the statistical variable,

$$T = \frac{\bar{a} - \bar{b}}{s\sqrt{\frac{1}{n} + \frac{1}{m}}},$$

obeys a Student distribution, where, s is the pooled standard deviation of all observed running times, and $\bar{a}$ and $\bar{b}$ are the average running times of the two configurations. Let T' be the value of T computed from S_a and S_b. From the probability distribution function of T, illustrated in Figure 5(a), it is easy to compute the probability that the observed value of T could be as large or larger than T' by chance under the null hypothesis. This probability is called the *p-value*. The smaller the p-value is, the more unlikely the null hypothesis holds. A common practice in statistics is to reject the null hypothesis if the p-value is smaller than a *significance level*, which is typically 0.05.

Besides the p-value, the hypothesis testing produces a *confidence interval*, $CI = [l, h]$, which is an interval the true difference in the means falls into with probability as high as (*1-significance level*). If the interval spans over zero, the two means are not considered to be significantly different. This criterion is equivalent to the one mentioned on p-value.

In addition to the quantitative results from the hypothesis testing, boxplots visualize the significance of the differences. The two boxplots in Figure 5(b) illustrates the distributions of T_a and T_b computed from their observed running times. The large overlap between the value ranges of the boxplots suggests that the difference between the mean performance of the two configurations is unlikely to be significant. In the following sections, we will show that these statistical techniques successfully prevent some noises from blurring the analysis of the effects of cache sharing.

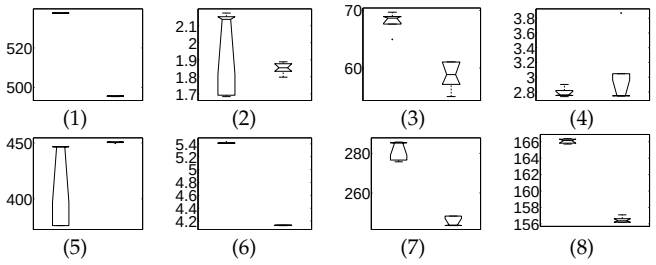
APPENDIX B
DETAILS OF MEASUREMENT AND FINDINGS

In this section, we report the detail of the experiments, the measurement 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 B.1. To prevent thread scheduling from affecting the comparison, for the sharing case, we also examine the performance difference caused by different assignments of threads to cores, as reported in Section B.2. We describe the results of *dedup* and *ferret* separately in Section B.3. They are two pipeline programs with task-level parallelism and several 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.

B.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 for most programs.

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,



config		1	2	3	4	5	6	7	8
p-value		0.0	0.3	0.0	0.4	0.1	0.0	0.0	0.0
CI	high	42	-0.2	6.1	-0.7	-72	1.3	30	9
	low	42	0.4	13	0.3	7.6	1.3	42	10
significant		Y	N	Y	N	N	Y	Y	Y

Fig. 7. Statistical analysis of the eight AMD configurations that have arrows on top in Figure 6 (left-to-right). Each boxplot shows the distribution of the running times of five repetitive runs of a configuration, with the sharing case on the left, and the non-sharing case on the right. The table reports the p-values, the confidence intervals (CI), and the analysis result—whether the difference between the sharing and non-sharing is statistically significant.

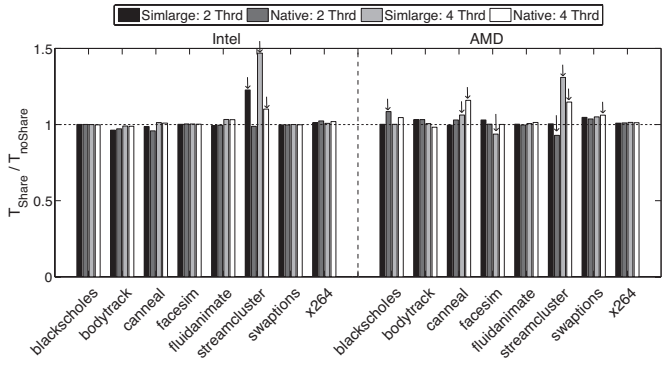


Fig. 6. The running time of each program in the sharing case normalized to its running time in the non-sharing case. The bars with an arrow on top are those whose height departs from 1 by over 5%.

because of the quad-core sharing, the two 4-thread cases actually both have some degree of 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 B.2 will show that other ways of assignments produce similar results.

Figure 6 presents the comparison of the average running times. Each bar in the figure shows the running time (averaged over five runs) of the program in the sharing case normalized to the time in the non-sharing case. So, a bar higher than 1 means that 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. On the Intel machine, only three bars show over 5% departure from 1; all are of *streamcluster*. On the AMD machine, eight bars have over 5% distance from 1.

Some repetitive runs in this experiment show large variations in their running times (e.g., 2.7s to 3.9s for one *simlarge* configuration of *Facesim* on the AMD machine). As mentioned in Section A, for such cases, average values are not enough for performance comparison.

We apply the statistical techniques described in Section A to all the configurations whose average running times depart from 1 by over 5%, which are highlighted by arrows on top of the corresponding bars in Figure 6. The boxplots in Figure 7 show the distribution of the running times, and the table below the graphs reports the p-values and confidence intervals. The bottom row of the table indicates that three of the eight configurations, contrary to what their average values show, do not show significant differences between the running times of their respective sharing and non-sharing cases. Take the second configuration (*canneal* on the *simlarge* input with 4 threads running on the AMD machine) as an example. Its confidence interval, [-0.2, 0.4], containing 0, suggests that the null hypothesis (the two cases have the same mean running time) cannot be rejected in the significance level (5%). Its p-value (0.3; greater than 0.05) gives the confirmation. The second graph in Figure 7 visualizes the insignificance of the difference by showing that there is substantial overlap between the ranges of the running times of the sharing and non-sharing cases.

The same statistical analysis shows that the three configurations on the Intel machine marked in Figure 6 all indeed have significant differences between the running times of their sharing and non-sharing cases.

The slowdown in the sharing cases of *streamcluster* on the *simlarge* input (especially on the Intel machine) is the most remarkable case among all the cases that have significant differences. The main reason for the slowdown is the cache and bus contention. The slowdown, however, becomes minor or even turns into small speedup for its runs on the *native* input. This change is because the larger size of the *native* input causes the working sets to grow so much that the L2 cache miss rates in the sharing and non-sharing cases become similar (as to be shown in Figure 8).

Overall, the sharing shows insignificant influence for the performance of most of the programs. The measured cache miss rates further confirm the result. Figure 8 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 most programs, consistent with the running time results shown in Figure 6.

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

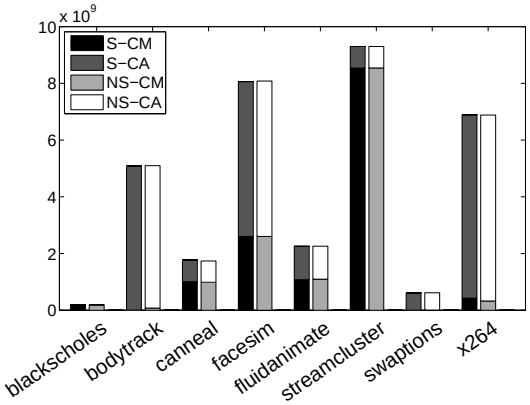


Fig. 8. Comparisons of L2-cache accesses and misses for 2-thread cases on the Intel machine. ("S" for cache-sharing cases; "NS" for non-sharing cases; "CM" for cache misses; "CA" for cache accesses.)

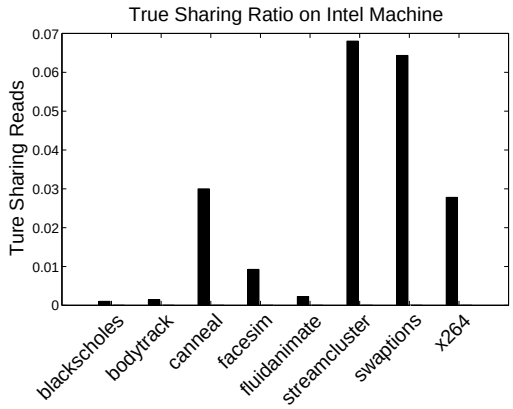


Fig. 9. Shared L2-cache reads in 2-thread cases on Intel, normalized by the total number of L2-cache accesses.

effects from shared cache. Figure 9 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 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 8. So the cache sharing has little influence on it either.

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

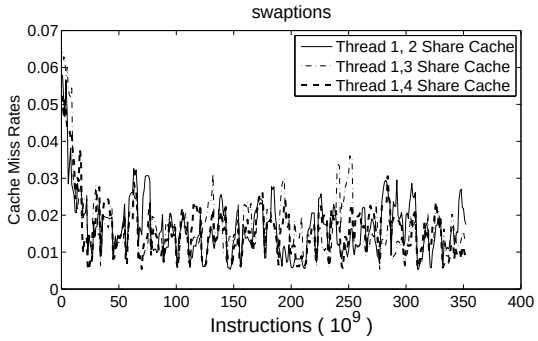
In light of the non-uniform cache sharing, the differences among threads may offer opportunities for performance improvement through appropriate placement of threads on cores. The sharing cases considered in the previous subsection contain just one thread-core assignment for each scenario. This section 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 example, when the 4 threads are assigned to two pairs of sibling cores residing on two AMD chips, the assignments we examine include $\{(T_0,T_1), (T_2,T_3)\}$, $\{(T_0,T_2), (T_1,T_3)\}$, $(T_0,T_3), (T_1,T_2)\}$, where, each pair of parentheses contain two threads assigned to two sibling cores.

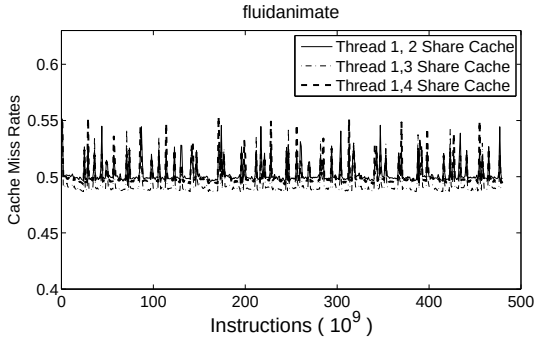
For 8-thread cases, we use three representative thread assignments in both the Intel and AMD architectures. We place the threads in such a way that threads whose indices differ by a given distance are assigned to sibling cores. For example on Intel machine, with the distance set to 1, every two consecutive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

Table 5 shows the maximum performance difference caused by the different assignments when the *simlarge* and *native* inputs are used. As the table shows, 6 of the 8 benchmarks have less than 4% maximum difference. For the 6 configurations (highlighted by boldface numbers) that show over 4% differences, we apply the statistical analysis introduced in Section A to each of them, and report the results in Figure 11. The clear overlap of the data ranges in most of the boxplots suggests the insignificance of the performance differences. The table below the boxplots gives quantitative confirmations. Among all the six configurations, only the third one, 4-thread of *Canneal* on the *native* input, shows significant difference between the running times of its two thread-core assignments. This result is a clear contrast against the over 4% differences observed on the average running times, which underscores the necessity of the statistical analysis.

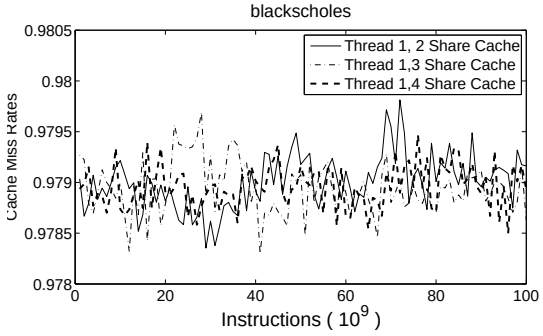
Overall, the different thread-core assignments do not show considerable effects on the program performance. There are two possible reasons. First, the threads in those



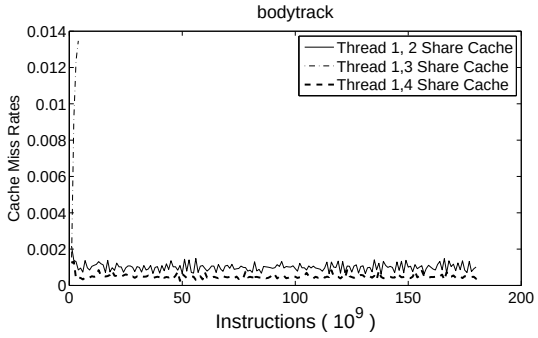
(a) Swaptions



(b) Fluidanimate

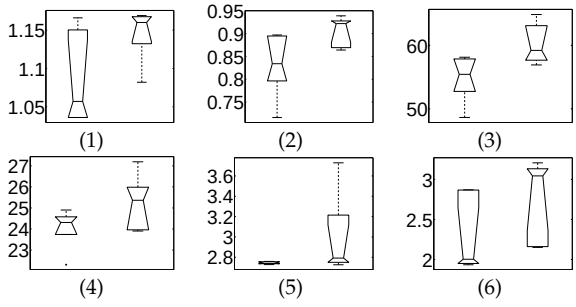


(c) Blackscholes



(d) Bodytrack

Fig. 10. Temporal traces of the L2 cache miss rates of four programs. Each has 4 threads running on the Intel machine. The threads are in three different co-running configurations.



config		1	2	3	4	5	6
p-value		0.1	0.09	0.04	0.2	0.2	0.2
CI	high	-0.1	-0.2	-11	-2.9	-0.7	-1.2
	low	0.01	0.01	-0.2	0.6	0.17	0.3
significant		N	N	Y	N	N	N

Fig. 11. Statistical analysis of the significance of the differences highlighted in Table 5. The first four graphs correspond to the highlighted configurations of *canneal* from left to right in Table 5; the rest correspond to *Facesim* from left to right.

TABLE 5
Maximal performance differences (%) caused by different bindings of threads to a given set of cores

	Intel				AMD			
	simlarge		native		simlarge		native	
Benchmarks	4-t	8-t	4-t	8-t	4-t	8-t	4-t	8-t
Blackscholes	0.04	0.08	0.02	0.03	0.1	0.3	0.05	0.04
Bodytrack	0.54	0.43	0.10	0.84	0.9	0.3	2.50	1.69
Canneal	3.69	**5.01**	1.94	3.0	2.1	**7.9**	**9.08**	**4.62**
Facesim	0.48	0.11	0.09	1.21	**8.8**	**15**	1.42	2.39
Fluidanimate	1.31	2.04	1.33	3.97	0.7	1.1	0.28	1.21
Streamcluster	0.49	0.22	0.12	0.08	0.1	0.3	1.14	0.04
Swaptions	0.40	0.76	0.09	0.46	0.9	0.9	0.21	0.65
X264	0.52	0.65	0.08	0.31	0.8	3.3	0.54	1.04

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 [23], [24]) when the program runs in different thread-core assignments. The three curves in each graph in Figure 10 represent the temporal traces of the L2-cache miss rates of a program when it runs on the Intel machine with threads placed on two pairs of sibling cores in three different ways, corresponding to three sharing cases. The four programs show different phase patterns, but the three sharing cases all 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.

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 [11], [25] and server programs [28]. However, Section 4 will show that program transformations may lead to an opposite conclusion.

B.3 Pipeline Programs

Unlike data-parallel programs, typical pipeline programs contain several concurrent computation stages, and the interactions among the threads exist both within and between stages. In PARSEC, *ferret* and *dedup* are two such programs. When changing thread-core bindings, we sometimes observe significant performance variations. However, the reason for the difference is not the effects from the shared cache, but the load balance across stages, as detailed in our previous paper [32].

B.4 Effects of Binding vs. Non-Binding of Threads

As many of the measurements shown earlier bind threads with cores, in this part, we examine the effects of the binding. 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. As mentioned in Section 2.2, in both binding and non-binding cases, we explicitly specify the set of cores to use (through the "taskset" command in Linux or the "pthread_setaffinity_np" function) when the number of threads is smaller than the total number of cores in a machine.

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. Furthermore, most running times of non-binding cases are either similar or longer than the corresponding binding cases [32], 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.

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.

APPENDIX C
OPTIMIZING *Ferret*

The program, *ferret*, is a pipeline program. It implements a search engine for finding a set of images that best match a query image by analyzing their contents. The program contains 6 concurrent pipeline stages. The first and final stages are for initialization and completion with only one thread in each. The other four stages have the same number of threads. The number is specified in the program input.

The fourth stage of the pipeline is the most memory intensive phase of the program. During that stage, the features of a query image extracted by the first three stages are compared against the feature database to identify the top K images closest to the query image. In the original program, each thread processes one query each time. Every query will look up some part of the database according to the locality sensitive hashing (LSH) that maps similar items to the same buckets in the hash table. The part of database that is traversed is much larger than the size of the last-level cache we experiment with, not to mention the size of the whole database.

We apply a two-step transformation to examine the potential of shared-cache–aware program optimizations. First, we split the database into m sections evenly (m equals the number of shared caches in the machine). The threads running on the ith shared cache compare the queries against only the ith database section. As a result, each query is processed by m threads (one on each shared cache). A post step is added to merge the results together. For example, there are eight threads with the first four running on the first chip and the second four on the other chip (assuming one shared cache per chip). After the transformation, the database is split evenly into two sections, owned by each chip. At most four queries can be processed at one time with threads 0 and 4 processing the first query, threads 1 and 5 processing the second and so on. In the second step, based on their data addresses, we reorder the queries and assign them to the corresponding threads so that nearby queries for the same database section are processed by sibling threads within a short time interval.

This transformation creates a non-uniform relation among threads: Sibling threads share similar data accesses in the same database section, but non-sibling ones do not. Besides the synergistic prefetching among sibling threads, an additional benefit is that shared cache can be used more efficiently: Rather than keeping multiple copies of a database entry in multiple shared caches as the original program entails, the transformed program ensures that different shared caches store different parts of the database. The transformation is insensitive to database size as data reuses are enhanced in an inter-thread level within short time intervals. As shown in the right part of Figure 4, the transformation eliminates most shared-cache misses, and yields a speedup of as much as 1.53.

```
float pFL(Points *points, int *feasible, int
numfeasible, float z, long *k, double cost, long
iter, float e, int pid, pthread_barrier_t* barrier){
   ... ...
   for (i=0; i< iter; i++) { // outer loop
      x = i%numfeasible; // a possible center
      change += pgain (feasible[x], points, z, k,
                              pid, barrier);
   }
     ... ...
}
```

```
float dist(Point p1, Point p2, int dim){
  float result=0.0;
  for (i=0;i<dim;i++)
    result += (p1.coord[i] - p2.coord[i])*
              (p1.coord[i] -p2.coord[i]);
  return result;
}
```

```
double pgain(long x, Points *points, double z, long int *numcenters, int pid, pthread_barrier_t* barrier){
  // i ∈ [k1, k2], the range of points the current thread operates on
  R: is_center[i], *numCenters;   W: center_table[i];
  P0:  W: gl_cost_of_openning_x, gl_number_of_centers_to_close;
  R: is_center[i];   W: center_table[i], switch_memship[i];
  for ( i = k1; i < k2; i++ ) { // inner loop
     float x_cost = dist(points->p[i], points->p[x], points->dim)* points->p[i].weight; //candidate references
    R: points->p[i].assign, points->p[i].cost, center_table[i];   W: switch_membership[i];
  }
  R: center_table[i], is_center[i];
  P0: R, W: gl_cost_of_openning_x,  gl_number_of_centers_to_close;
  R: points->p[i].assign, points->p[i].corrd, points->p[i].weight, center_table[i], is_center[i],
    gl_cost_of_openning_x, switch_membership[i];   W: points->p[i].cost, points->p[i].assign, is_center[i];
  P0: W: *numberCenters, gl_cost_of_openning_x,  gl_number_of_centers_to_close;
  R: gl_cost_of_openning_x;
}
```

Fig. 12. The sketch of part of the actual code of *streamcluster*. All procedures shown are executed by every thread. (W: the data that are written; R: the data that are read; P0: thread 0; line segments: barriers.) The two loops in boldface correspond to the nested loop shown in Figure 1(a).

APPENDIX D
DISCUSSION ON THE AUTOMATION OF CACHE-SHARING-AWARE TRANSFORMATIONS

How to automate cache-sharing-aware transformations is a challenging problem beyond the scope of this paper. This section examines the manual transformations we have conducted in a further depth to expose some insights for the development of such optimizers.

Cache-sharing-aware transformation consists of two steps. The first is to identify data references to optimize based on the frequency, and the potential influence on cache performance of all major data objects in a program. Both static and profiling approches may facilitate this process.

The second step, code transformation, is more complicated than the first. The principle of the transformation is to change the way the candidate data structures are referenced so that a large amount of data are shared among sibling threads, but little among non-siblings. The main approach to such changes is loop transformations as exemplified in Figure 1.

However, automating such loop transformations in real code is much more complex than what the simplified code in Figure 1 shows. Figure 12 outlines the sketch of the real code of some functions in *streamcluster*. The two levels of the nested loop shown in Figure 1(a) correspond to the two loops in boldface in Figure 12.

Compared to the simplified code in Figure 1, the code in Figure 12 has some complexities worth mentioning. First, the two levels of the nested loop lie in two procedures, *pFL()* and *pgain()* respectively, with a considerable amount of code existing both before and after the inner loop. In addition, the procedure *pgain()* uses some pointers whose targets are not easy to resolve through existing pointer analysis techniques. The implication to automatic optimizers is that they must have the capability for loop analysis across procedures, and for handling the inaccuracy in the analysis of aliases and pointers.

Second, some data references, both outside and inside the inner loop, carry true dependences across the iterations of the outer loop. For example, the value of *points->p[i].assign* loaded through a statement inside the inner loop may come from an update by the previous iteration of the outer loop. Similar dependences exist on the references to **numCenters*, *is_center*, and *points->p[i].cost*. These dependences make it infeasible to directly conduct the unrolling transformation of the outer loop, an important step in the optimization illustrated in Figure 1(b). In our manual transformation, we address this problem based on the observation that despite all those dependences, the candidate references in the inner loop (highlighted by boldface comments) carry no true dependences across the iterations of the outer loop. Therefore, if we can extract the related statements into a new procedure and invoke it before *pgain()*, the transformations shown in Figure 1(b) would become possible. To do so, the optimizer must be able to determine the validity of the transformation by checking two properties: No dependences in *pgain()* are invalidated, and no errors are introduced by the resulting changes to the timing that is originally set by the many barriers in *pgain()*. These requirements suggest another two capabilities that the automatic optimizer must have: They must be able to detect data dependences both inside and cross procedures, and handle various parallel-programming constructs, including barriers, synchronizations, locks, and so on.

It is worth mentioning that between the two extremes, complete manual transformations and fully automatic optimizations, there is a spectrum of degrees of combination between them. Manual transformations are powerful, but time consuming, and error-prone. More importantly, they are subject to hardware changes: In

many cases, the transformations have to hard code the thread-core assignments to attain a deterministic sibling relations among threads, so that they can realize the suitable non-uniform data sharing. The optimized program would be difficult to adapt to other systems that have different core layouts. On the other hand, automatic optimizations are free of those drawbacks but are constrained by the various program complexities as mentioned in the previous paragraphs.

An appropriate integration of programmer's knowledge in the automatic optimizer may help get best of both worlds. The design of the interface between them is an interesting research problem. Ideally, it should allow easy input from the programmer to help resolve the program complexities, but impose no hardware-dependent constraints to the transformation. Detailed studies are out of the scope of this paper.