

Characterizing Multi-threaded Applications based on Shared-Resource Contention

Tanima Dey

Wei Wang

Jack W. Davidson

Mary Lou Soffa

Department of Computer Science

University of Virginia

Charlottesville, VA 22904

Email: {td8h, ww6r, jwd, soffa}@virginia.edu

Abstract—For higher processing and computing power, chip multiprocessors (CMPs) have become the new mainstream architecture. This shift to CMPs has created many challenges for fully utilizing the power of multiple execution cores. One of these challenges is managing contention for shared resources. Most of the recent research address contention for shared resources by single-threaded applications. However, as CMPs scale up to many cores, the trend of application design has shifted towards multi-threaded programming and new parallel models to fully utilize the underlying hardware. There are differences between how single- and multi-threaded applications contend for shared resources. Therefore, to develop approaches to reduce shared resource contention for emerging multi-threaded applications, it is crucial to understand how their performances are affected by contention for a particular shared resource. In this research, we propose and evaluate a general methodology for characterizing multi-threaded applications by determining the effect of shared-resource contention on performance. To demonstrate the methodology, we characterize the applications in the widely used PARSEC benchmark suite for shared-memory resource contention. The characterization reveals several interesting aspects of the benchmark suite. Three of twelve PARSEC benchmarks exhibit no contention for cache resources. Nine of the benchmarks exhibit contention for the L2-cache. Of these nine, only three exhibit contention between their own threads—most contention is because of competition with a co-runner. Interestingly, contention for the Front Side Bus is a major factor with all but two of the benchmarks and degrades performance by more than 11%.

I. INTRODUCTION

Large scale multicore processors, known as chip multiprocessors (CMPs), have several advantages over complex uniprocessor systems, including support for both thread-level and instruction-level parallelism. As such, CMPs offer higher overall computational power which is very useful for many types of applications, including parallel and server applications. Consequently, these machines have become the new mainstream architecture and are now being widely used in servers, laptops and desktops. This shift to CMPs has created many challenges for fully utilizing the power of multiple execution cores. One of these challenges is the management of contention for shared resources. As there are multiple cores and separate hardware execution modules for each core, the contention among applications for a computing resource has been reduced. However, there are several resources in CMPs which are shared by several and/or all processing cores, including on-chip shared and last-level caches (LLC),

Front Side Bus (FSB), memory bus, disk, and I/O-devices. In particular, contention for the shared resources in the memory hierarchy can dramatically impact the performance of applications, as shown in several recent studies [13, 17, 21, 22, 30]. Contention is especially critical for real-time, high priority and latency sensitive applications as timing is crucial for these applications.

There are several approaches for addressing contention for shared resources. One approach is to characterize the applications to better understand their interactions and behaviors on CMPs by performance analyses for shared-resource contention. Such understanding can lead to techniques for reducing shared-resource contention, improving system throughput and realizing scalable performance. For example, Xie and Loh characterize and classify applications based on cache miss-rate and propose dynamic cache partitioning policy with this classification [26]. Another approach is to detect shared-resource contention dynamically and take counter measures to mitigate the contention [8, 11, 15, 21, 24, 30]. Several other recent studies address contention for shared LLC by using different hardware techniques [7, 12, 16, 19, 23, 25].

As the hardware scales up to many cores, the trend of application design has shifted towards multi-threaded and parallel programming to fully utilize the underlying hardware. Most of the recent studies address contention for shared resources mainly by single-threaded applications. However, there are differences between how single- and multi-threaded applications contend for shared resources. For example, single-threaded applications suffer from shared cache contention when there is another application (co-runner) on the neighboring core sharing the same cache. In this case, the application suffers from contention for one shared cache. On the other hand, multi-threaded applications have multiple threads running on multiple cores, sharing more than one cache with multiple co-runners. A multi-threaded application can also suffer from shared-resource contention among its own threads even when it does not have any co-runner.

Additionally, when threads from multi-threaded applications run on cores that have separate caches and have true sharing, they suffer from additional cache misses to maintain the cache coherency. When one of the data-sharing threads writes the shared data, all shared copies cached by the other threads are invalidated following the cache coherency protocol. When

one data-sharing thread reads the data again, a cache miss occurs because of the invalid cache-line. These additional cache misses can also lead to performance degradation. Therefore, multi-threaded applications suffer from shared-resource contention differently than single-threaded applications, for both solo-execution and execution with co-runners.

For multi-threaded applications, there are two categories of contention for shared resources. *Intra-application* contention can be defined as the contention for a resource among threads of the same application when the application runs solely (without co-runners). In this situation, application threads compete with each other for the shared resources. *Inter-application* contention can be defined as the contention for shared resources among threads from different applications. In this case, threads from one multi-threaded application compete for shared resources with the threads from its co-running multi- or single-threaded application.

Both types of contention can severely degrade a multi-threaded application's performance [14, 30]. To develop approaches for reducing contention for shared resources, it is crucial to understand how the performance of a multi-threaded application is affected because of such contention. A standard methodology to characterize these applications, based on contention, would facilitate such understanding.

In this research, we propose and evaluate a general methodology for characterizing emerging multi-threaded applications by determining the effect of shared-resource contention on its performance. Using the methodology, we are able to determine the performance implications of multi-threaded applications and characterize them with regard to both intra- and inter-application shared-resource contention. Previous research efforts demonstrate that the contention for shared resources in the memory hierarchy cause significant loss of performance and quality of service [21, 30]. Therefore, we characterize the multi-threaded applications from the PARSEC (Princeton Application Repository for Shared-Memory Computers) benchmark suite [5], based on shared-resource contention in the memory hierarchy to demonstrate the methodology. The PARSEC benchmark suite is especially designed for modern CMPs, incorporating benchmarks having diverse characteristics and using different programming models. From the performance analyses and characterization of the benchmarks, it is possible to determine precisely the sensitivity of a benchmark's performance to a particular resource in the memory hierarchy.

The contributions of the work are:

- We propose a general methodology to characterize multi-threaded applications based on performance analyses for both intra- and inter-application shared-resource contention.
- Using the methodology, we perform thorough performance analyses and characterization of the multi-threaded PARSEC benchmarks based on contention for several resources in the memory hierarchy on real hardware.
- The characterization reveals several interesting aspects of the benchmark suite. Three of twelve PARSEC bench-

marks exhibit no contention for cache resources. Nine of the benchmarks exhibit contention for the L2-cache. Of these nine, only three exhibit contention between their own threads—most contention is because of competition with a co-runner. Interestingly, contention for the FSB is a major factor with all but two of the benchmarks and degrades performance by more than 11%.

The paper is organized as follows: Section II describes the methodology to determine intra- and inter-application contention of multi-threaded applications for shared resources. Section III describes the experiments conducted to characterize the PARSEC benchmarks for shared-resource contention in the memory hierarchy. The experimental results are analyzed in Section IV and discussed in Section V. Section VI discusses related work and Section VII concludes the paper.

II. METHODOLOGY

In this section, we describe the methodology to measure intra- and inter-application contention along with the choices of machine architecture.

A. Measuring intra-application contention

To measure intra-application contention of a multi-threaded application for a shared (targeted) resource, we need to analyze how sharing the targeted resource among threads from the same multi-threaded application affects its performance, compared to when they do not share. To accomplish this measurement, the application is run solely with at least two threads in two configurations. The first or *baseline* configuration maps the threads such that the threads do not share the targeted resource and run using two separate dedicated resources. The second or *contention* configuration maps the application threads such that the threads do share the targeted resource and execute while using the same resource. Because the contention configuration maps the threads to use the same resource, it creates the possibility that the threads compete with each other for that resource causing intra-application contention that degrades the application's performance. In both configurations, the mapping of threads keeps the effect on other related resources the same. For example, if we measure intra-application contention for L1-caches, the mapping of threads in both configurations must maintain the same effect on the rest of the memory hierarchy, including L2/L3-cache and the FSB. When we compare the performances of the two configurations, the difference indicates the effect of contention for that particular resource on the application's performance. If the performance difference between the baseline and the contention configuration is negative (degradation), then there is intra-application contention for that resource among the application threads. However, if the performance difference is positive (improvement), there is no intra-application contention for the targeted resource.

B. Measuring inter-application contention

To measure the inter-application contention for a shared (targeted) resource, we need to analyze the performance differences when threads from different applications run together

sharing that resource, compared to when the resource is not shared. To accomplish this measurement, multi-threaded applications are run with a co-runner which can be another multi- or single-threaded application. Similar to the approach for characterizing intra-application contention, pairs of applications are run in two configurations. The *baseline* configuration maps the application threads such that each application has exclusive access to the targeted resource. In this configuration, the applications do not share the resource and there is no interference or contention for that resource from the co-running application. The *contention* configuration maps the application threads such that threads from one application share the targeted resource with the co-runner's thread creating the possibility of inter-application contention. Similar to the intra-application contention, both configurations map the application threads such that the thread-mapping to the other related shared resources remains the same so the effect of contention for the targeted resource can be precisely determined. When we compare the performance of both configurations, the difference indicates the effect of contention for the targeted resource on each application's performance. If the performance difference between the baseline and contention configuration is negative (degradation), then the application suffers from inter-application contention for that resource. If the performance difference is positive (improvement), the intra-application contention for the targeted resource dominates the performance more as performance does not degrade due to contention caused by the co-runner's threads.

C. Additional methodology requirements

Choosing machine architecture: To determine the effect of contention for a particular shared resource on an application's performance, we use a machine architecture that supports the necessary experiments according to the methodology. The methodology is flexible enough that either a simulator or real hardware can be used. In general, when we consider a resource for contention analysis, the experiments are run on such platforms which have a multiple number of that targeted resource with the same parameter values. For example, if we plan to measure the contention for L3-cache, we need a system which has three levels of caches and at least two L3-caches so that we can apply the methodology. In this case, parameters values are cache parameter, including cache size, number of ways, and cache block size. Additionally, we have to ensure that the mapping and capacity sizes of the other related shared resources are the same. For the above example, the sizes of L1-, L2-cache and how they share the L3-cache must be the same.

Collecting performance information: For both intra- and inter-application contention analyses, we need to compare application's performance of both configurations. This information can be collected using any appropriate techniques, for example, reading hardware performance counters, the *time* command, the real-time clock, counting cycles or clock-ticks etc. In our experiments, we use hardware performance counters to gather this information, as described in Section III.

III. CHARACTERIZATION OF PARSEC BENCHMARKS

According to the methodology, to characterize multi-threaded benchmarks based on both types of contention, we perform two categories of experiments: (1) we run the benchmarks solely, and (2) we run each benchmark with a co-runner which is another multi-threaded benchmark. Each category contains three sets of experiments in which each set is designed to target a specific resource in the memory hierarchy and measures the impact of contention on performance for that resource. The resources in the memory hierarchy that are considered in the experiments are: L1-cache, L2-cache and FSB.

The multi-threaded workloads that we use in our experiments are from the latest PARSEC2.1 benchmark suite. It consists of thirteen benchmarks. For our experiments, we use twelve benchmarks which use the POSIX-thread (*pthread*) library for creating threads. We keep profiling overhead as low as possible and employ a simple technique to instrument the benchmarks by identifying *pthread-create* system calls. By detecting new thread creation, we gather each thread's *threadID* information which is necessary to get the per-thread profile information. We do not include the benchmark *freqmine* in our analysis because it uses OpenMP to create threads. We can intercept thread creation for OpenMP, but require using software dynamic translators to detect the thread creation, which causes higher run-time overhead.

To collect different run-time statistics, as each benchmark in each experiment is run, profile information is collected by reading hardware performance counters using the Perfmon2 tool [10]. The interfaces defined in the tool's library allow user-space programs to read hardware performance counters on thread-basis (per-thread) and system-basis (per-core). These interfaces enable users to access the counters' values with very low run-time overhead. The initial set up for the counters takes $318\mu\text{sec}$ and reading one counter's value takes $3.5\mu\text{sec}$, on average. After the initialization of the performance counters for each thread, the values of the counters are read via signal handlers when periodic signals are sent (every second) to each thread by using the *threadID* information. As performance is a direct measure of contention [22], we collect the counter, UNHALTED_CORE_CYCLES's sampling values for each thread in both configurations in all experiments to determine both intra- and inter-application contention.

We use statically linked binaries for our experiments, compiled with GCC 4.2.4. We do not include any additional compiler optimization flag other than the ones used in the benchmarks' original makefiles. The threads are affinityized by pinning them to cores via Linux *sched_setaffinity()* system call. When we are co-scheduling two applications and one application finishes before the rest, we immediately restart it. We perform this restart until the longest running application completes five iterations. We collect profile information for five iterations to ensure low variability in the collected values. In modern processors, the prefetchers use very sophisticated techniques (e.g., stride pattern detection) to fetch data into

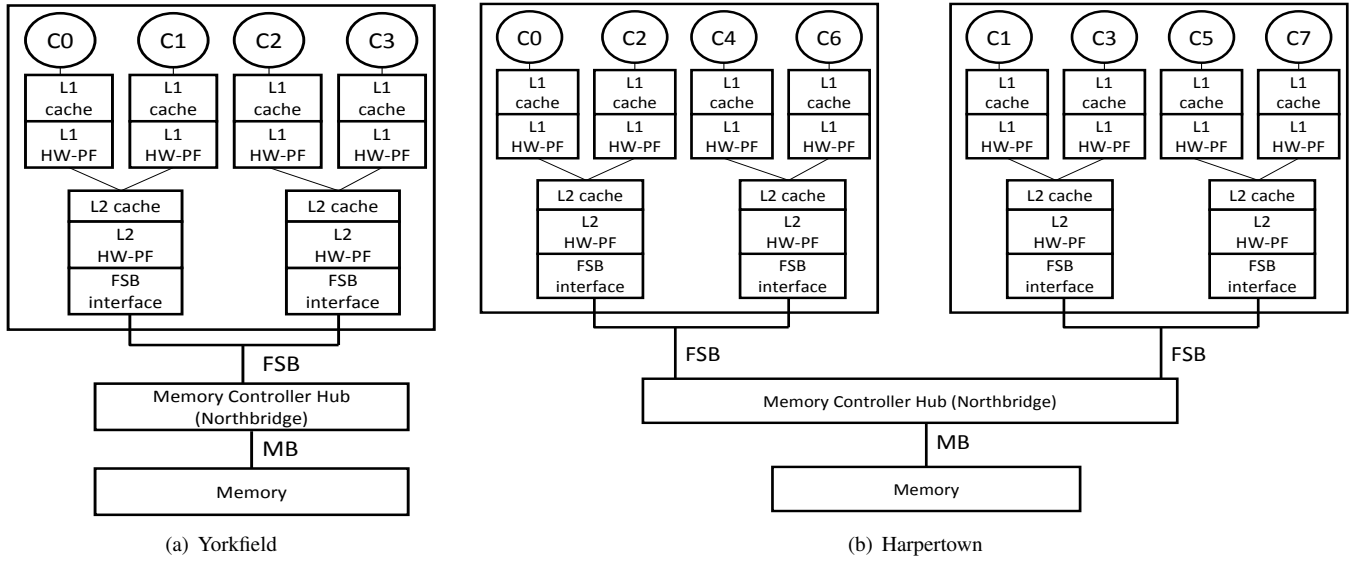


Fig. 1. Experimental Platforms (*CX* stands for the processor core, *L1 HW-PF* and *L2 HW-PF* stand for hardware prefetcher for L1- and L2-caches, respectively and *FSB* and *MB* stand for Front Side Bus and Memory Bus, respectively.)

the cache in advance which can affect the count of cache misses. Therefore, we also disable prefetching to eliminate any effects that it might cause so we could precisely measure cache contention.

A. Experimental platforms

To measure the contention among threads for different levels of caches, we require a machine which has at least two levels of cache. Because we need to map the application threads to the cores sharing one cache to determine the effect of cache contention, the platform must have both private (per-core) L1- and shared L2-caches. It must have single socket memory connection, so the contention for the FSB is expected to be the same and we are able to measure only contention for the caches. Intel Q9550 (“Yorkfield”), shown in Figure 1(a), satisfies the requirements for such experiments and we use this platform to measure cache contention. This platform has four cores and each core has private L1-data and L1-instruction cache, each of size 32KB. It has two 6MB 24-way L2-caches and each L2-cache is shared by two cores. It has 2 GB of memory connected by single socket to the L2-cache, so there is one FSB. It runs Linux kernel 2.6.25.

Similarly, to measure contention for the FSB among threads, we need a platform which has multiple socket connections to memory, i.e., multiple FSBs. The cache hierarchy in each FSB connection must be the same so that we can isolate contention for the FSB by keeping the other factors (L1-/L2-cache contention) unchanged. Dual Intel Xeon E5462 (“Harpertown”), shown in Figure 1(b), fulfills these requirements. Therefore, we choose this platform to measure contention for FSB. It has two processors each having four cores. Each core has private L1-data and L1-instruction cache each of size 32 KB. Each pair of cores share one of the four 6MB 24-way L2-caches. This platform has dual sockets and each processor

has a separate bus connected to 32GB memory. It runs Linux kernel 2.6.30.

B. Measuring intra-application contention

To measure intra-application contention for the memory hierarchy resources, we run each PARSEC benchmark solely with native input set. The experiments are described below.

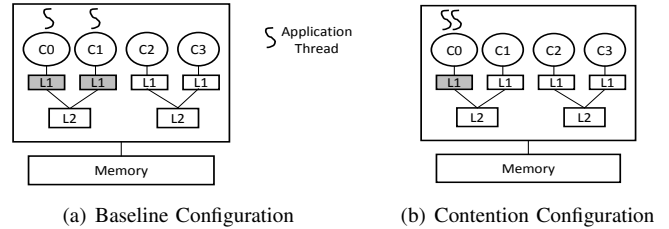


Fig. 2. Configurations for measuring intra-application L1-cache contention

L1-cache: According to the methodology, to measure the effect of intra-application contention for L1-cache on performance, we run each PARSEC benchmark in two configurations. In each configuration, the number of threads to run equals the number of cores sharing one L2-cache. In the baseline configuration, two threads from a benchmark use their own private L1-caches and there is no intra-application L1-cache contention. These threads are mapped onto the two cores which share one L2-cache, e.g., C0 and C1 (shown in Figure 2(a)). In the contention configuration, two threads from the benchmark share one L1-cache compared to the exclusive access. In the presence of intra-application L1-cache contention, the threads compete for L1-cache space when they share the L1-cache and access conflicting cache-lines. Here, these threads are mapped onto one core, e.g., C0 (shown in

Figure 2(b)) or C1. As we measure contention for the L1-cache, we keep the effect of L2-cache contention the same by mapping threads to the cores that share the same L2-cache. Furthermore, we make sure that contention for FSB remains unchanged and choose Yorkfield which has one FSB, for this experiment. The only difference between these two configurations is the way L1-cache is shared between the threads and we are able to measure how L1-cache contention affects the benchmark’s performance.

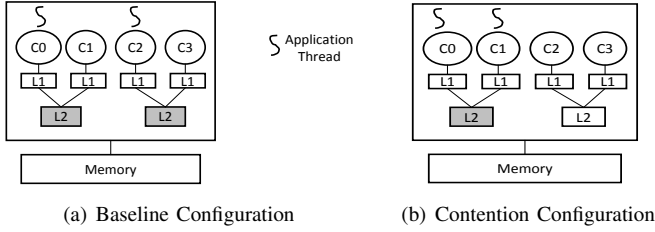


Fig. 3. Configurations for measuring intra-application L2-cache contention

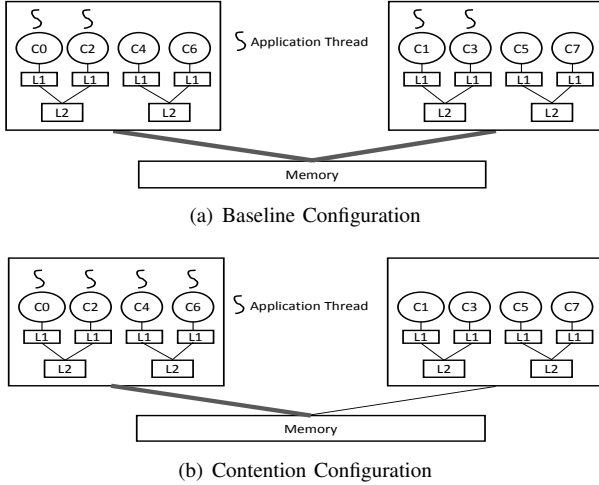


Fig. 4. Configurations for measuring intra-application FSB contention

L2-cache: Similar to L1-cache contention, to measure the effect of intra-application contention for L2-cache on performance, we run each PARSEC benchmark in two configurations. Each configuration runs threads equal to the number of L2-caches sharing one FSB. In the baseline configuration, two threads from a benchmark use their own L2-cache, avoiding intra-application contention for L2-cache. The threads are mapped onto the two cores which have separate L2-cache, e.g., C0, C2 (shown in Figure 3(a)) or C1, C3. In the contention configuration, two threads from the benchmark share one L2-cache and contend for L2-cache with each other. Here, the threads are mapped onto the two cores sharing one L2-cache, e.g., C0, C1 (shown in Figure 3(b)) or C2, C3. As we measure contention for L2-caches, we avoid intra-application L1-cache contention by allowing only one thread to access one L1-cache and keep the FSB contention unchanged between configurations by choosing Yorkfield which has one FSB.

Front side bus: To measure the effect of intra-application contention for the FSB on performance, we need to understand how sharing the FSB among application/benchmark threads affects its performance compared to using separate FSB. For this experiment, we use Harpertown as it has more than one FSB. According to the methodology, we run each PARSEC benchmark in two configurations. In each configuration, the number of threads equals the number of cores sharing one FSB to fully utilize its bandwidth. In the baseline configuration, four threads from a benchmark use separate FSB equally and do not compete for this resource. Four threads are mapped onto the four cores which have separate socket connections (separate bus) to memory (via shared L2-cache), e.g., C0, C2, C1 and C3 (shown in Figure 4(a)). In the contention configuration, four threads use only one FSB and there is potential contention among them for this resource. In this case, four threads are mapped onto the four cores sharing one socket connection to memory, e.g., C0, C2, C4 and C6 (shown in Figure 4(b)). As both configurations use the same number of threads as cores and L1-caches are private to each core, there is no intra-application contention for L1-cache. Similarly, as both configurations use two L2-caches shared by an equal number of threads, the contention for L2-cache remains the same. So comparing the performance output of the configurations, we are able to determine how bus bandwidth and FSB contention affect the performance of each benchmark.

For the performance analysis of a multi-threaded application for intra-application contention for a particular resource, we use the following formula:

$$\text{Percent_Performance_Difference} = \frac{(\text{Sum_Cycles_Base} - \text{Sum_Cycles_Contend}) * 100}{\text{Sum_Cycles_Base}} \quad (1)$$

Here, *Sum_Cycles_Base* and *Sum_Cycles_Contend* are the sum of the sampling values of the hardware performance counter, UNHALTED_CORE_CYCLES, in the baseline and contention configuration, respectively.

C. Measuring inter-application contention

To understand the effect of inter-application contention for a particular resource in the memory hierarchy on performance, each PARSEC benchmark is run with another PARSEC benchmark (co-runner) in two configurations with native input set. As we consider twelve benchmarks, there are 66 distinct benchmark pairs in each experiment. The experiments for each shared resource are described below.

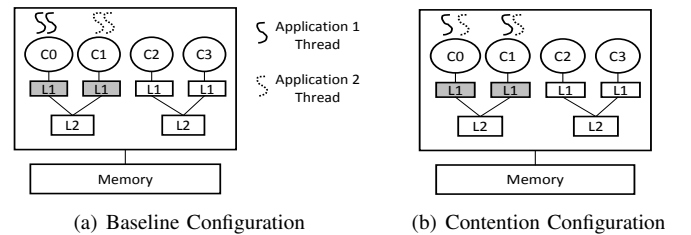


Fig. 5. Configurations for measuring inter-application L1-cache contention

L1-cache: To measure the effect of inter-application contention for L1-cache on performance, we run pairs of PARSEC benchmarks each with two threads in two configurations. In the baseline configuration, two threads of each benchmark get exclusive L1-cache access. There is no inter-application contention for L1-cache between them because L1-cache is not shared with the co-runner’s threads. Two threads of one benchmark are mapped onto one core, e.g., C0 and two threads of the co-running benchmark are mapped onto the other core, e.g., C1 (shown in Figure 5(a)). In the contention configuration, two threads from both benchmarks share the L1-caches and there is potential contention for L1-cache among them. Here, two threads from both benchmarks are mapped onto the two cores which share the same L2-cache, e.g., C0 and C1 (shown in Figure 5(b)). As we measure contention only for L1-caches, we keep the effect of L2-cache contention the same by using the same L2-cache and choose Yorkfield, having one FSB, to make sure that the contention for the FSB remains unchanged.

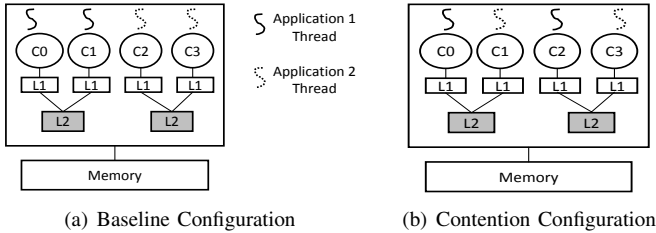


Fig. 6. Configurations for measuring inter-application L2-cache contention

L2-cache: Similar to L1-caches, to determine the effect of inter-application contention for L2-caches, we run pairs of PARSEC benchmarks each with two threads in two configurations. In the baseline configuration, two threads of each benchmark get exclusive L2-cache access. There is no inter-application L2-cache contention among them as L2-cache is not shared with the co-runner’s threads. On Yorkfield, two threads of one benchmark are mapped onto two cores, e.g., C0, C1 (shown in Figure 6(a)) or C2, C3 and two threads of the co-running benchmark are mapped onto the remaining cores, e.g., C2, C3 (shown in Figure 6(a)) or C0, C1. In the contention configuration, one thread each from both benchmarks shares the L2-caches and there is potential contention for L2-cache between them. As shown in Figure 6(b), one thread from both benchmarks are mapped onto the two cores which share one L2-cache, e.g., C0, C1 and the second threads from both benchmarks are mapped onto the remaining two cores which share the second L2-cache, e.g., C2, C3. Both configurations use the same L1-cache size and single socket memory connection. So we are able to measure how L2-cache contention affects each benchmark’s performance because the only difference between these configurations is how the L2-cache is shared.

Front side bus: We run two PARSEC benchmarks each with four threads on Harpertown in two configurations to measure the effect of inter-application contention for FSB. In

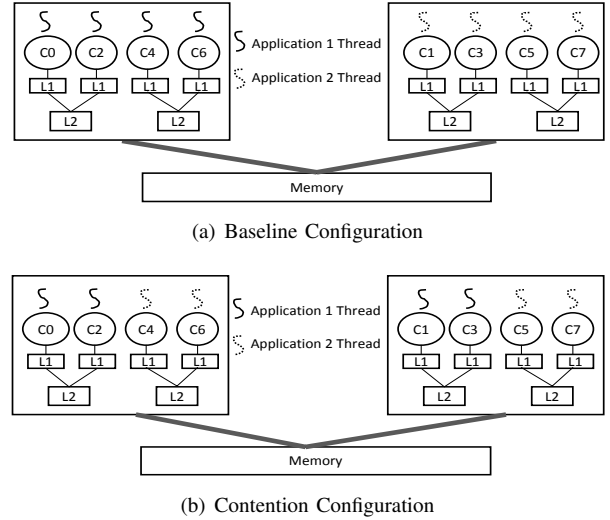


Fig. 7. Configurations for measuring inter-application FSB contention

the baseline configuration, each benchmark gets its exclusive FSB access and there is no FSB interference/contention from the co-running benchmark. Four threads from one benchmark are mapped onto four cores sharing one socket connection to memory, e.g., C0, C2, C4, C6 and four threads from the other benchmark are mapped onto the remaining four cores sharing the second socket connection to memory, e.g., C1, C3, C5, C7 (shown in Figure 7(a)). In the contention configuration, both benchmarks share both FSB and there is potential contention for this resource between them. Here, four threads from one benchmark are mapped equally onto the four cores which have separate socket connections (separate bus) to memory, e.g., C0, C2, C1 and C3 and the remaining threads on the remaining cores (shown in Figure 7(b)). The only difference between these two configurations is how applications share the FSB connected from the memory controller to the chipset, i.e., whether it is connected by the same or separate bus. As both configurations use the same sized L1- and L2-cache, the contention for L1- and L2-cache remains unchanged and we are able to determine how separate FSB usage affects the performance of each benchmark.

For the performance analysis for inter-application contention for a particular resource, we use Equation ?? to calculate the percentage performance difference between the application’s performances in two configurations with each of its 11 co-runners.

IV. EXPERIMENTAL RESULTS AND ANALYSES

In this section, we present and analyze the experimental results for intra- and inter-application contention of the PARSEC benchmarks. The twelve benchmarks used in the experiments are: *blackscholes* (BS), *bodytrack* (BT), *cannal* (CN), *dedup* (DD), *facesim* (FA), *ferret* (FE), *fluidanimate* (FL), *raytrace* (RT), *streamcluster* (SC), *swaptions* (SW), *vips* (VP) and *x264* (X2).

A. Intra-application contention analyses

Figure 8-10 gives the results of measuring intra-application contention for L1-, L2-cache and FSB. The positive and negative performance difference indicates performance improvement and degradation respectively.

L1-cache: The results of intra-application contention of the PARSEC benchmarks for L1-caches are shown in Figure 8. We observe in the figure that all the benchmarks except *blackscholes*, *ferret* and *vips* show performance improvement. This improvement is because when two threads from the same benchmark are mapped to use one L1-cache, data sharing among the threads causes fewer cache misses as they share more than they contend for L1-cache. The fewer number of cache misses reduces the number of cycles to complete execution, improving the performance. Additionally, as the benchmark's threads share data, when they are mapped to use the same L1-cache, there are up to 99% reduced cache-snooping operations, decreasing the cache coherency protocol's overhead. These benchmarks, showing performance improvements, do not suffer from intra-application contention for L1-cache. As *facesim* and *streamcluster* show performance improvements of 8.6% and 11% respectively, they have a large amount of data sharing. *Dedup* and *fluidanimate* show performance improvement of 0.9% and 1.6% respectively. *Canneal*, *raytrace* and *x264* show very small performance improvements of 0.5%, 0.28% and 0.31% respectively. Although *bodytrack* and *swaptions* show performance improvement, the magnitude of improvement is very close to zero, less than 0.1%. So the data sharing in these benchmarks yields minimal improvements. On the other hand, *ferret* and *vips* have fewer number of sharers compared to other benchmarks [5], causing contention when the threads share L1-cache. This characteristic results in more cache-misses and performance degradation by approximately 1% and 7%, respectively. Because these benchmarks show performance degradation when the threads only share L1-cache, they suffer from intra-application contention for L1-caches. *Blackscholes* shows the lowest and almost negligible performance degradation of 0.4% and is not much affected by L1-cache contention.

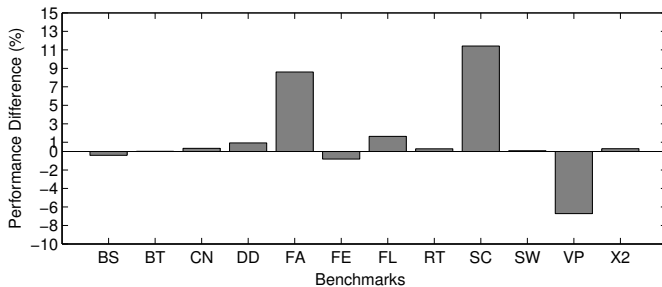


Fig. 8. Experimental results of measuring intra-application L1-cache contention

L2-cache: The results of intra-application contention of the PARSEC benchmarks for L2-caches are shown in Figure 9. In the figure we observe that performance degrades for *dedup*,

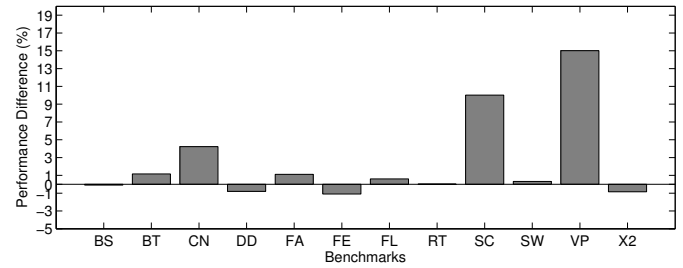


Fig. 9. Experimental results of measuring intra-application L2-cache contention

ferret and *x264*. The rest of the benchmarks show performance improvements because of sharing and do not suffer from intra-application L2-cache contention. Among these, *canneal*, *streamcluster* and *vips*' performance improvements are among the highest, respectively 4%, 10% and 15%. Although *vips* does not show performance improvement due to sharing in L1-caches, it shows better sharing in L2-cache, reducing the cache misses up to 43%. *Bodytrack*, *facesim*, *fluidanimate* and *swaptions* show small performance improvements of 1.16%, 1.11%, 0.60% and 0.34% respectively. *Raytrace*'s performance improvement is negligible, approximately 0.02% showing very small amount of sharing. *Dedup*, *ferret* and *x264* show performance degradation close to 1%, on average and suffer from some intra-application contention for L2-caches. *Blackscholes* has the lowest and almost negligible performance degradation of 0.09% for intra-application L2-cache contention.

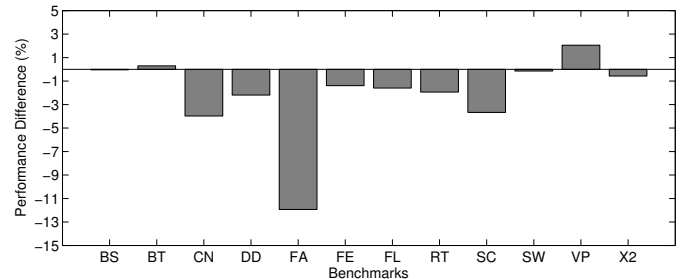


Fig. 10. Experimental results of measuring intra-application FSB contention

Front side bus: The results of intra-application contention of the PARSEC benchmarks for the FSB are shown in Figure 10. We observe from the graph that the performances of most of the benchmarks except *bodytrack* and *vips* degrade when we map the threads to use one FSB. Because there is performance degradation for the reduced bus bandwidth, we conclude that there is intra-application contention for the FSB among the threads of these benchmarks. *Facesim* suffers the most performance degradation (nearly 12%), thereby showing the highest intra-application contention for the FSB. *Canneal* and *streamcluster* suffer nearly 4% performance degradation. The performances of *dedup*, *ferret*, *fluidanimate* and *raytrace* degrade on average by 2%. The performance degradation of *swaptions* and *blackscholes* is negligible, less than 0.05%. In contrast, *vips* and *bodytrack* show performance improve-

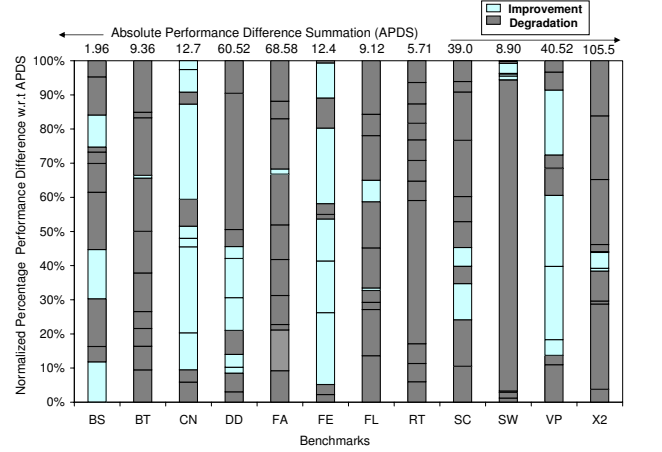
ments of 2% and 0.15% respectively and do not show intra-application contention for the FSB.

B. Inter-application contention analyses

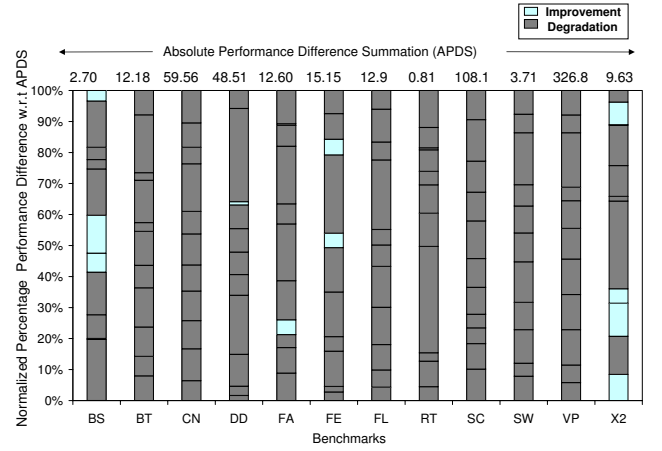
The experimental results of measuring the inter-application contention for L1-, L2-cache and the FSB are given in Figure 11(a)- 11(c). In each figure, the X-axis corresponds to each benchmark in alphabetical order. Each column is a percentage stacked graph, where the stacks or segments show the performance results of a benchmark with each of its 11 co-runners in alphabetical order from the bottom to the top. The lighter shade segments represents performance improvement and darker shade represents performance degradation. For example, in Figure 11(a) for *BS*, the first segment from the bottom shows performance improvement with *BT*, the next segment shows performance degradation with *CN* while measuring inter-application L1-cache contention. Similarly, for *BT*, the first and second segment from the bottom shows performance degradation respectively with *BS* and *CN*.

If a particular segment in a benchmark’s column is in the lighter shade, it means that the benchmark’s performance improves in the contention configuration. A performance improvement results when the benchmark’s threads show lower contention for the resource with its co-runner’s threads compared to the contention among its own threads for that resource. For example, in Figure 11(b), *FE*’s (Ferret) performance improves when running with *RT* (Raytrace) as its co-runner which means *FE*’s threads do not have much sharing among themselves and have more L2-cache contention among themselves than the contention with the co-running *RT*’s threads. On the other hand, if a particular segment in a benchmark’s column is in the darker shade, it means that the benchmark’s performance degrades in the contention configuration and the benchmark’s threads suffer from higher contention with its co-runner’s threads than the contention among its own threads. For example, in Figure 11(b), *FE*’s performance degrades when running with *SC* (Streamcluster) as its co-runner which means *FE*’s threads have more L2-cache contention with the co-running *SC*’s threads than the contention among its own threads.

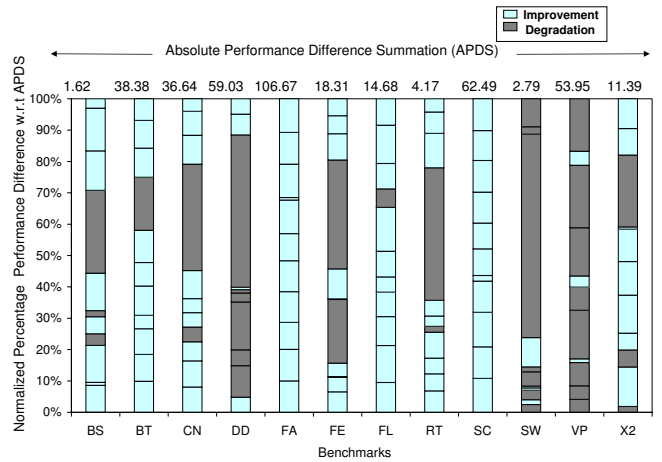
The number on top of each column (*Absolute Performance Difference Summation (APDS)*) is sum of the absolute percentage performance differences of each benchmark with each of its 11 co-runners. The height of each segment in the columns represents percentage performance difference of a benchmark with one of its co-runners, normalized with respect to this summation to keep the total height of its column at 100%. To get the actual percentage performance difference for a benchmark with any co-runner, we multiply the height of the appropriate segment in the benchmark’s column with the APDS value above the column. For example, to get the actual percentage performance improvement for *BS* with *BT* for L1-cache contention, we multiply the height of first segment of first column in Figure 11(a) with 1.96, which is $0.1173 \cdot 1.96 = 0.23$.



(a) Inter-application contention for L1-cache



(b) Inter-application contention for L2-cache



(c) Inter-application contention for the FSB

Fig. 11. Experimental results of inter-application contention. APDS is sum of the absolute percentage performance differences of each benchmark with its 11 co-runners. The segments in each column shows performance results for a benchmark with its 11 co-runners in alphabetical order and the actual percentage performance difference is obtained by multiplying the height of the appropriate segment with its APDS value.

L1-cache: From the results of inter-application contention of the PARSEC benchmarks for L1-cache (shown in Figure 11(a)), we can categorize the benchmarks into three classes. This classification depends on how much they suffer from inter-application contention for L1-cache resulting in performance degradation or number of the darker shaded segments in each column. The first class includes the benchmarks whose column has most of its segments in the darker shade and shows highest inter-application contention for L1-cache. This class includes *bodytrack*, *facesim*, *fluidanimate*, *raytrace*, *streamcluster*, *swaptions* and *x264*. Among these benchmarks, *facesim*, *streamcluster*, *x264* and *raytrace*, *swaptions* show, respectively, the most and least contention as the APDS values are among the highest and lowest of all benchmarks in this class. The next class includes the benchmarks whose columns have almost half of its height in the lighter and the other half in the darker shade. This class includes *blackscholes* and *dedup*. From the magnitude of APDS, we infer that *dedup* has more performance impact for L1-cache contention compared to *blackscholes*. The third category includes the benchmarks whose columns have most segments in the lighter shade. This class includes *canneal*, *ferret* and *vips* which suffer more from intra-application than inter-application contention for L1-cache as their performance improve with most of the co-runners. *Vips* suffers the most due to intra-application contention as it has the highest APDS value among these benchmarks (also validated by the Figure 8 results).

L2-cache: Figure 11(b) shows the experimental results of the inter-application contention of the PARSEC benchmarks for L2-cache. Similar to L1-cache contention results, we can categorize the benchmarks in three classes. In this case, we categorize them based on the APDS values as we observe in the figure that most of the benchmarks have all the column-segments in the darker shade, denoting performance degradation due to inter-application L2-cache contention. The first class includes the benchmarks which have the highest APDS values representing greater impact on the performance. This class includes *canneal*, *dedup*, *streamcluster* and *vips*. All the segments of these benchmarks' columns are in the darker shade showing performance degradation due to high inter-application contention for L2-cache. The next category includes the benchmarks which have lower APDS than that of the previous class. This class includes *bodytrack*, *facesim*, *ferret*, *fluidanimate* and *x264*. These benchmarks have most column segment in the darker shade showing performance degradation for L2-cache contention except *x264* and *ferret*. *X264* shows more intra-application contention for L2-cache with *blackscholes*, *canneal*, *dedup* and *swaptions* as co-runner. *Ferret* shows more intra-application L2-cache contention with *raytrace* and *swaptions* as co-runner. The last class includes the rest of the benchmarks which show very small APDS values. This class includes *blackscholes*, *raytrace* and *swaptions*. For each co-runner, these three benchmarks show on average 0.24%, 0.07% and 0.31% performance differences respectively, which is very small compared to those of the other benchmarks. So we can conclude that these three benchmarks'

performances are not much affected by the inter-application L2-cache contention.

Front side bus: From the results of inter-application contention of the PARSEC benchmarks for the FSB (shown in Figure 11(c)), we can categorize the benchmarks into three classes. Similar to L1-cache, the classification depends on how much they suffer from inter-application contention for the FSB resulting in performance degradation (i.e., the total length of darker segments in each column). The first class includes the benchmarks *dedup*, *swaptions* and *vips* which have the most column area in the darker shade. *Dedup* and *vips* show the highest APDS values in this class and suffer more from inter-application contention for the FSB. The second class includes benchmarks which have both the lighter and darker shaded segments of almost equal length. This class includes *blackscholes*, *ferret*, *raytrace* and *x264*. Among these benchmarks, *blackscholes* and *raytrace* have very small APDS values, so their performance is not much affected because of the FSB contention. The third class includes the benchmarks whose columns have most of the segments in the lighter shade. These benchmarks' performances improve because of the increased bandwidth and they have more intra-application than inter-application contention for the FSB. This class includes *bodytrack*, *facesim*, *fluidanimate*, *canneal* and *streamcluster*. Among these benchmarks, *facesim* and *streamcluster* have the highest APDS values which indicate they have higher intra-application contention for the FSB which is also validated by the results in Figure 10. We include *canneal* in this category as for most of its co-runners, it improves performance when it uses increased bandwidth and it also has high APDS value. All benchmarks suffer from inter-application contention for the FSB when they run with *streamcluster* as co-runner in contention configuration. From this we infer that *streamcluster* has a higher memory requirement for which its co-runners suffer. Only *facesim* does not degrade performance with *streamcluster* as it suffers more due to intra-application contention.

V. DISCUSSION

Table I summarizes the performance sensitivity of the PARSEC benchmarks by contention for L1-, L2-cache and FSB. Each entry of the table indicates the type of contention for the corresponding resources that affect the benchmarks' performance the most, considering all the results in Figure 8-11. Analyzing the percentage performance difference and APDS values respectively in all intra- and inter-application contention results, we infer that the performances of *blackscholes* and *swaptions* are not affected much and they do not have any performance sensitivity to contention for the considered resources in the memory hierarchy.

In the inter-application FSB contention results (Figure 11(c)), the white segments in *bodytrack*'s column represents that there is performance improvement (on average $31.9/10 = 3.19\%$) with its co-runners when the threads share the FSB in the contention configuration. Observing this improvement, we can infer that *bodytrack* has more contention among its own threads than the contention with its co-runner's threads

Benchmarks	L1-cache	L2-cache	FSB
<i>blackscholes</i>	none	none	none
<i>bodytrack</i>	inter	inter	intra
<i>canneal</i>	intra	inter	intra
<i>dedup</i>	inter	intra, inter	intra, inter
<i>facesim</i>	inter	inter	intra
<i>ferret</i>	intra	intra, inter	intra
<i>fluidanimate</i>	inter	inter	intra
<i>raytrace</i>	none	none	intra
<i>streamcluster</i>	inter	inter	intra
<i>swaptions</i>	none	none	none
<i>vips</i>	intra	inter	inter
<i>x264</i>	inter	intra, inter	intra

TABLE I
PERFORMANCE SENSITIVITY OF THE PARSEC BENCHMARKS DUE TO
CONTENTION IN THE MEMORY HIERARCHY RESOURCES

for the FSB. *Bodytrack*'s performance does not degrade in the intra-application FSB contention experiment. However, when we compare the magnitude of the performance results between intra- and inter-application FSB contention, the performance result for inter-application (3.19%) is higher than that for intra-application contention (0.14%). Therefore, *bodytrack* is affected by intra-application FSB contention the most, especially in the presence of the co-runners. Similarly, *canneal* suffers from intra-application L1-cache contention (on average by $10.04/7=1.43\%$) when it runs with co-runners. But its performance does not degrade in intra-application L1-cache contention experiment. When we compare the magnitude of the performance results between intra- and inter-application L1-cache contention, we observe that the performance result for inter-application (1.43%) is higher than that for intra-application contention (0.34%). Therefore, *canneal* is affected by intra-application L1-cache contention in the presence of the co-runners. *Canneal*'s performance is degraded the most when it shares L2-cache with its co-runners.

Dedup is the only benchmark which suffers from inter-application contention for all the resources considered in the memory hierarchy. *Fluidanimate* and *ferret* suffer the most performance degradation from intra-application FSB contention. *Facesim* and *streamcluster* show the highest performance sensitivity to intra-application FSB contention among all the benchmarks. Although *raytrace* does not show significant performance degradation due to any cache contention, shows intra-application contention for the FSB. *Vips* suffers the most due to intra-application contention for L1-cache and inter-application contention for L2-cache. *X264* suffers the most performance degradation due to inter-application L1-cache contention.

VI. RELATED WORK

There has been prior work addressing shared-resource contention. Zhuravlev et al. analyzes the effect of cache contention created by co-runners and provides a comprehensive analy-

sis of different cache-contention classification scheme [30]. Chandra et al. proposes analytical probabilistic models to analyze inter-thread contention in the shared L2-cache in hyper-threaded CMPs for the SPEC benchmarks [6]. Mars et al. synthesizes and analyzes cross-core performance interference for LLC on two architectures [20, 22]. Zhao et al. investigates low overhead mechanisms for fine-grained monitoring of shared cache, along with the usage, interference and sharing [29]. Xu et al. proposes a shared cache aware performance model [27]. These works mainly consider cache contention for single-threaded applications. In contrast, we propose a general methodology to characterize any multi-threaded application for not only LLC contention, but also contention for private cache and the FSB. Jin et al. characterizes parallel workload for resource contention in the memory hierarchy, but they mainly focus on comparing the systems and run applications solely [14]. Whereas we mainly focus on characterizing multi-threaded applications, both when running alone and with a co-runner and determine its sensitivity to contention for a particular resource-contention.

There also has been prior work on characterizing PARSEC. In the original PARSEC paper [5], the authors provide several characteristics of the benchmarks including working set, locality, effects of different cache block size, degree of parallelization, off-chip traffic, and programming models. A more recent paper [4] includes the description of a new benchmark (*raytrace*), the revisions and improvements made in the newer version, PARSEC2.0. Bhaduria et al. describes cache performance, sensitivity with respect to DRAM speed and bandwidth, thread scalability and micro-architectural design choices for the benchmarks over a wide variety of real machines [2]. The authors describe temporal and spatial behavior of communication characteristics among the PARSEC benchmarks' threads [1]. Bhattacharjee et al. describes TLB behavior of these benchmarks and provides many useful insights about redundant and predictable inter-core I- and D-TLB misses, useful for better and novel TLB designs for emerging parallel workloads [3]. The authors transform the PARSEC benchmarks in cache-sharing-aware manner during compilation time [28]. Lakshminarayana and Kim categorize these benchmarks into three classes based on execution time variability and analyze synchronization barrier and critical sections affect [18]. Performance portability is analyzed for a subset of the PARSEC benchmarks by thread building block environment [9]. Most of the work analyze and characterize each benchmark for solo-execution. Our work complements the above research as we characterize the benchmarks both for solo-execution and execution with co-runners on real hardware in contrast to simulators.

VII. CONCLUSION

As CMPs have become the default computing fabric, characterization of multi-threaded applications' contention for shared resources has become very important. We believe that our proposed methodology for intra- and inter-application contention analyses will be useful for providing insight and understand-

ing of multi-threaded applications' execution behaviors. The provided insights and understanding will help designer build more efficient systems as well as execution environments for improved performance and throughput.

VIII. ACKNOWLEDGEMENTS

This research is supported by National Science Foundation grant number CCF-0811689. We appreciate the insightful comments and constructive suggestions from the anonymous reviewers.

REFERENCES

- [1] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterization of SPLASH-2 and PARSEC. In *Int'l Symp. on Workload Characterization (IISWC)*, 2009.
- [2] M. Bhaduria, V. M. Weaver, and S. A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Int'l Symp. on Workload Characterization (IISWC)*, 2009.
- [3] A. Bhattacharjee and M. Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2009.
- [4] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009.
- [5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Int'l Conf. on Parallel architectures and compilation techniques (PACT)*, 2008.
- [6] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2005.
- [7] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Int'l. Conf. on Supercomputing*, 2007.
- [8] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Haravellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Symp. on Parallel Algorithms and Architectures (SPAA)*, 2007.
- [9] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Int'l Symp. on Workload Characterization (IISWC)*, 2008.
- [10] S. Eranian. Perfmon2: A flexible performance monitoring interface for Linux. In *Linux Symp.*, 2006.
- [11] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2007.
- [12] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Int'l Conference on Supercomputing (ICS)*, 2004.
- [13] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2008.
- [14] H. Jin, R. Hood, J. Chang, J. Djomehri, D. Jespersen, and K. Taylor. Characterizing application performance sensitivity to resource contention in multicore architectures. Technical report, NASA Ames Research Center, 2009.
- [15] L. Jin and S. Cho. SOS: A software-oriented distributed shared cache management approach for chip multiprocessors. In *Int'l. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2009.
- [16] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2004.
- [17] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, 2008.
- [18] N. Lakshminarayana and H. Kim. Understanding performance, power and energy behavior in asymmetric multiprocessors. In *Int'l Conf. on Computer Design (ICCD)*, 2008.
- [19] H. Lee, S. Cho, and B. R. Childers. StimulusCache: Boosting performance of chip multiprocessors with excess cache. In *Int'l. Symp. on High-Performance Computer Architecture (HPCA)*, 2010.
- [20] J. Mars and M. L. Soffa. Synthesizing contention. In *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.
- [21] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *Int'l Symp. on Code Generation and Optimization (CGO)*, 2010.
- [22] J. Mars, L. Tang, and M. L. Soffa. Directly characterizing cross core interference through contention synthesis. In *Int'l Conf. on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2011.
- [23] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Int'l. Symp. on Microarchitecture (MICRO)*, 2006.
- [24] A. Snavey and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.
- [25] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
- [26] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *In Proc. of CMP-MSI, held in conjunction with ISCA-35*, 2008.
- [27] C. Xu, X. Chen, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *Int'l Symp. on Performance Analysis of Systems Software (ISPASS)*, 2010.
- [28] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.
- [29] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2007.
- [30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.