ReSense: Mapping Dynamic Workloads of Colocated Multithreaded Applications Using Resource Sensitivity

TANIMA DEY, WEI WANG, JACK W. DAVIDSON, and MARY LOU SOFFA,
University of Virginia

To utilize the full potential of modern chip multiprocessors and obtain scalable performance improvements, it is critical to mitigate resource contention created by multithreaded workloads. In this article, we describe ReSense, the first runtime system that uses application characteristics to dynamically map multithreaded applications from dynamic workloads—workloads where multithreaded applications arrive, execute, and terminate continuously in unpredictable ways. ReSense mitigates contention for the shared resources in the memory hierarchy by applying a novel thread-mapping algorithm that dynamically adjusts the mapping of threads from dynamic workloads using a precalculated sensitivity score. The sensitivity score quantifies an application's sensitivity to sharing a particular memory resource and is calculated by an efficient characterization process that involves running the multithreaded application by itself on the target platform. To measure ReSense's effectiveness, sensitivity scores were determined for 21 benchmarks from PARSEC-2.1 and NPB-OMP-3.3 for the shared resources in the memory hierarchy on four different platforms. Using three different-sized dynamic workloads composed of randomly selected two, four, and eight corunning benchmarks with randomly selected start times, ReSense was able to improve the average response time of the three workloads by up to 27.03%, 20.89%, and 29.34% and throughput by up to 19.97%, 46.56%, and 29.86%, respectively, over the native OS on real hardware. By estimating and comparing ReSense's effectiveness with the optimal thread mapping for two different workloads, we found that the maximum average difference with the experimentally determined optimal performance was 1.49% for average response time and 2.08% for throughput.

Categories and Subject Descriptors: D.4.1 [**Process Management**]: Scheduling, Thread-mapping; D.4.8 [**Performance**]: Measurement

General Terms: Coscheduling, Performance, Measurement, Algorithms

Additional Key Words and Phrases: multicore, multithreaded applications, thread mapping, resource contention, memory hierarchy

ACM Reference Format:
Dey, T., Wang, W., Davidson, J. W., and Soffa, M. L. 2013. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. ACM Trans. Architec. Code Optim. 10, 4, Article 41 (December 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555298

1. INTRODUCTION

With the continuous growth of the number of cores on modern chip multiprocessors (CMPs), the number of simultaneously executing multithreaded applications is increasing. When there are multiple applications executing on CMPs, a key challenge is to determine the thread-to-core mappings to optimize performance [Das et al. 2013;

This work is supported by the National Science Foundation, under grants CCF-0811689 and CNS-0964627.
Authors' addresses: T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa, Department of Computer Science, University of Virginia, Charlottesville, Virginia 22904; email: td8h@virginia.edu.

DOI: http://dx.doi.org/10.1145/2555289.2555298

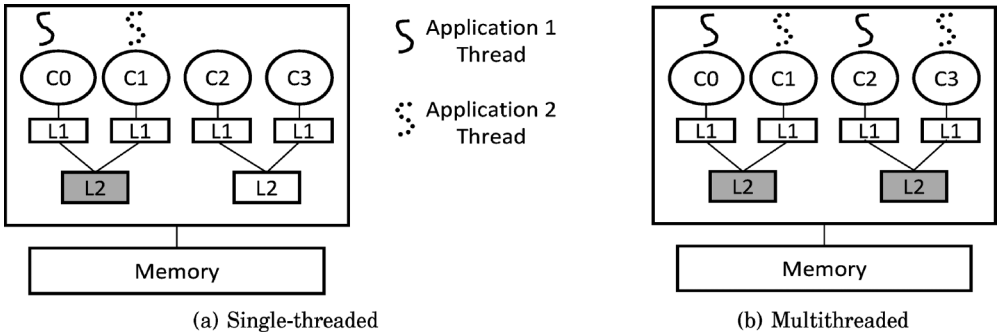


Fig. 1. Difference between a single- and multithreaded application for shared cache contention.

Zhuravlev et al. 2010]. When an application shares any resource with a corunner,[1] contention can occur for that shared resource. Because of resource contention in the memory hierarchy, an application's performance can degrade by more than 50% [Zhuravlev et al. 2010], and thus scalable performance improvement is often not readily achieved on multicore and many-core machines [Dey et al. 2011]. Contention for the shared resources in the memory hierarchy can also lead to inefficient resource usage [Jin and Cho 2009; Knauerhase et al. 2008]. To utilize these resources to their full potential and obtain scalable performance improvements on CMPs, intelligent thread mapping is critical.

A number of techniques have been proposed to address the shared-resource contention problem for *single-threaded* applications and *static* workloads[2] via thread mapping and scheduling [Fedorova et al. 2007; Jiang et al. 2008; Knauerhase et al. 2008; Mars et al. 2011b; Zhuravlev et al. 2010; Mars et al. 2010]. However, there are several differences between the mapping of single- and multithreaded applications as they contend for shared resources [Dey et al. 2011].

A single-threaded application can have one corunning thread on a neighboring core when it shares, for example, a cache (L2), as shown in Figure 1(a). Consequently, it contends for one cache with that corunner. On the other hand, a multithreaded application can share multiple caches with multiple corunning threads, as shown in Figure 1(b), and contend for more than one cache. Moreover, single-threaded applications do not have data sharing, whereas multithreaded applications can have data sharing, as well as contentious behavior, among its threads [Dey et al. 2011]. Existing techniques to address resource contention for single-threaded applications do not consider these differences and are thus not applicable for mapping multithreaded applications.

There are several challenges to effectively map multithreaded applications on CMPs. For workloads with multiple applications, the most effective thread mapping, which minimizes contention in the shared resources, depends on an application's behaviors, the underlying resource topology of the platform, and the behaviors of the corunning applications. One approach is to develop an online thread-mapping algorithm that detects and mitigates contention in the shared resources created by corunning applications. Online contention detection involves performance comparison of different thread-to-core-mapping configurations, which vary the contention for the shared resources. Mitigation involves selecting the thread-to-core mapping that ensures the lowest contention and performance degradation [Tang et al. 2011; Snavely and Tullsen 2000]. However, as

[1]Applications that execute on the same or neighboring cores are corunners.

[2]A static workload is one in which all applications start execution at the same time, and the set of simultaneously executing applications does not change during execution.

the multithreaded applications in a workload can create a varying number of threads, the number of thread-to-core mapping configurations can increase exponentially [Radojković et al. 2012]. As a result, determining the thread mapping that minimizes contention by online contention detection in all possible thread-mapping configurations has exponential complexity and makes the problem of optimally mapping threads an NP-complete problem [Jiang et al. 2008].

Another issue arises when thread-mapping algorithms consider multithreaded applications in realistic *dynamic* workloads, where any number of multithreaded applications arrive, execute, and terminate in unpredictable ways. Online detection and minimization of the contention created by dynamic workloads is very challenging because of the continuous change in the total number of applications and the intensity of contention in the execution environment, resulting in exponentially varying numbers of thread-mapping configurations.

Another approach to mitigate shared-resource contention, and the one presented in this article, is to first determine the inherent characteristics and potential behaviors of each multithreaded application in the workload for how it creates and suffers from the contention on the underlying platform, using an offline technique. After characterization, during execution, the thread-mapping algorithm dynamically maps the application threads using the characterization rather than exploring the exponential number of thread-to-core mapping configurations. Prior work characterized applications in the presence of corunning applications and synthetic workloads [Mars et al. 2011a, 2011b], where the number of characterizations to be performed during execution can increase polynomially depending on the number of applications in the workload.

In this article, we describe **ReSense**, the first runtime system that uses application characteristics to dynamically map threads from dynamic workloads of multithreaded applications to mitigate contention for the shared-memory resources. Each characteristic of a multithreaded application is represented by a *sensitivity score*, which is calculated offline via an independent characterization process. This characterization process measures the potential impact of contention for a specific shared resource in the memory hierarchy, including shared caches, Last-Level Caches (LLCs), Front-Side Bus (FSB), on-chip memory controller, and memory-socket interconnection, on a multithreaded application's performance. The characterization is done by running the application by itself on a particular CMP, which keeps the number of characterizations in linear order. The sensitivity score identifies different behaviors (e.g., data sharing or contentiousness) of a multithreaded application and is precise enough to evaluate the relative importance of a shared resource for an application. The score can be used to compare the contentiousness among corunning applications as well as the contentiousness among sibling threads (threads from the same multithreaded application) with the corunner's threads.

Using the sensitivity score of each application, ReSense applies a novel thread-mapping algorithm that dynamically determines the thread mappings of multithreaded applications in the presence of any number of corunners to maximize the workload's average response time and throughput. Because ReSense uses the precalculated sensitivity scores of the applications, it is capable of handling dynamic workloads and mapping any number of threads from any number of applications arriving and terminating nondeterministically, avoiding the performance overhead of using online contention detection mechanisms.

There have been a few research efforts that address contention for colocated multithreaded applications. Table I summarizes the important differences between ReSense and other state-of-the-art systems for mitigating contention for the shared-memory resources. Some of these systems pursue the goals to independently optimize energy, thread throughput, minimize lock contention, allocate optimal number of cores to the

Table I. Comparison between ReSense and Some State-of-the-Art Systems

Systems	ReSense	Pusukuri et al. TACO'13	Bhadauria et al. PACT'10	Tang et al. ISCA'11	Zhuravlev et al. ASPLOS'10
Resources	All shared resources in memory hierarchy	Last-level cache and shared lock	Shared cache bus	All shared resources in memory hierarchy	All shared resources in memory hierarchy
Primary objective	Maximize workload's average response time and throughput	Maximize workload's average turnaround time and throughput	Minimize system's energy and maximize thread throughput	Satisfy quality of service of one latency-sensitive application	Reduce workload's completion time
Application characteristics detection complexity	Linear, no corunner considered	Polynomial, increases with the number of corunners	Polynomial, increases with the number of corunners	Polynomial, increases with the number of corunners	Polynomial, increases with the number of corunners
Multithreaded application	Yes	Yes	Yes	Yes	No
Dynamic workloads	Yes	Yes	No	No	No
Performance compared with optimal/oracle	Average less than 1%	Unknown	Average less than 1%	Within 3%	Within 2%
Baseline comparison	Native OS	Native OS	Suleman et al. ASPLOS'08	None	Native OS

applications in a workload, or improve only the latency-sensitive application's performance. Our goal is to improve the overall performance of the colocated multithreaded applications from dynamic workloads and mitigate contention for all the shared resources in the memory hierarchy by determining the effective thread mapping. All the systems mentioned in Table I determine characterization in the presence of a corunner. In comparison, ReSense is able to characterize applications without considering corunners. Thus, ReSense operates in linear time to determine application characteristics, while the other systems operate in polynomial time and are able to achieve similar performance. Details of these systems and the comparisons with ReSense are described in Section 4.

This work makes the following contributions:

—The first dynamic thread-mapping algorithm, ReSensor, which determines the thread-to-core mappings of the multithreaded applications in dynamic workloads using sensitivity scores to optimize performance. These scores are based on an application's solo characterization, without considering any corunning applications, for each shared resource in the memory hierarchy on a particular CMP.
—A low-overhead runtime system, ReSense, which uses the ReSensor algorithm and employs the precalculated sensitivity score of a multithreaded application for a specific shared resource to effectively determine and dynamically adjust its thread mapping in the presence of any number of multithreaded applications from dynamic workloads for mitigating contention and optimizing performance.
—A comprehensive empirical evaluation of the effectiveness of ReSense, which demonstrates that it can improve the average response time and throughput of dynamic

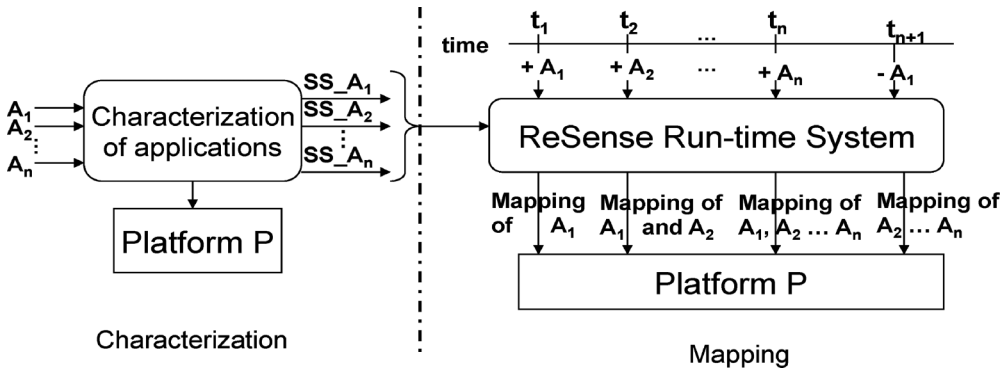


Fig. 2. Overview of the ReSense approach.

workloads consisting of multiple multithreaded applications by up to 29.34% and 46.56%, respectively, over the native Operating System (OS) using real hardware.

—A performance comparison of ReSense's effectiveness in mitigating contention and improving application performance with that of the optimal thread mapping, which demonstrates that the maximum average differences with the experimentally determined optimal performance is 1.49% for average response time and 2.08% for throughput for two different workloads.

This article is organized as follows: Section 2 describes the ReSense system and ReSensor algorithm. Section 3 describes experimental methodology and evaluation metrics and discusses the results. Section 4 describes related work and Section 5 concludes.

2. THE RESENSE APPROACH

In this section, we describe the ReSense runtime system, characterization of the multithreaded applications, sensitivity scores, and ReSensor algorithm.

2.1. The ReSense Runtime System

Figure 2 shows a high-level overview of the ReSense approach. Consider a dynamic workload consisting of n multithreaded applications, $\{A_1, A_2, \ldots, A_n\}$ arriving (represented by + sign) at time $\{t_1, t_2, \ldots, t_n\}$, respectively, on platform P. To map these applications, the ReSense approach has an offline and online phase. The offline *characterization* phase identifies the potential behaviors (e.g., data sharing or contentiousness) of each multithreaded application and determines each application's sensitivity scores (SS_A_1, SS_A_2, ... , SS_A_n) for each shared resources in the memory hierarchy on P. This characterization is done for each application in isolation and consequently needs to be done only once for a particular shared resource on the target platform. Section 2.2 discusses the characterization and the calculation of the sensitivity score for a multithreaded application in more detail.

In the online *mapping* phase, the ReSense runtime uses the ReSensor algorithm to determine the thread-to-core mappings of the multithreaded applications using the sensitivity scores for each application on P. The ReSense runtime invokes this algorithm when there is a change in the number of threads or applications in the system. Therefore, as the execution of each application *starts* or *terminates* or any thread is *created* or *destroyed*, ReSense dynamically adjusts the thread mappings of the applications. For example, in Figure 2, at time t_1, there is only one application A_1, and ReSense maps A_1's threads on P. At time t_2, A_2 starts execution and ReSense

determines and adjusts the thread mapping of A_1 and A_2 dynamically. At time t_n, there are n multithreaded applications running, and ReSense maps all n applications on P using the ReSensor algorithm. If at time t_{n+1} any application A_i (e.g., A_1 in Figure 2) terminates, ReSense adjusts the mappings of the remaining executing applications.

The next two sections describe the two phases of the ReSense approach in detail.

2.2. Characterization: Sensitivity Score

Multithreaded applications demonstrate different behaviors for different resources on CMPs, which we can determine from their characterization. In this work, we consider the shared resources in the memory hierarchy, for example, shared caches, LLCs, memory controller, FSB, and memory socket connection. We apply our prior methodology for characterizing multithreaded applications for a particular shared resource in the memory hierarchy [Dey et al. 2011].

We defined two types of contention created by multithreaded applications. *Intra-application* contention is defined as the contention for a resource among threads of the same application when the application runs solely (without corunners). In this situation, application threads compete with each other for the shared resources. *Interapplication* contention is defined as the contention for shared resources among threads from different applications. In this case, threads from one multithreaded application compete for shared resources with the threads from its corunning multi- or single-threaded application. Our methodology characterizes multithreaded applications based on both intra-application and interapplication contention.

Because it is infeasible to perform the necessary characterization with corunners for large dynamic workloads, we characterize a multithread application as it runs solely (without any corunners). For example, for one corunning application, there are $O(n^2)$ pair-wise characterizations, and for $(r-1)$ corunning applications, there are $O(n^r)$ characterizations for n applications for each shared resource. On the other hand, the characterization of a multithreaded application when it runs solely has linear $O(n)$ complexity for each shared resource. As a multithreaded application's characteristics vary depending on the underlying platform, offline characteristics for different resources can be determined *only once* for each target platform. This type of offline characterization is useful for scientific workloads where the input size does not change often. This one-time offline characterization has much less overhead than considering corunners and less performance overhead than determining the contention online in the presence of an unknown number of corunners.

Our methodology measures how sharing the targeted resource among threads from the same application affects its performance, compared to when they do not share. To accomplish this measurement, a multithreaded 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 and dedicated resources. The second or *contention* configuration maps the application threads such that the threads 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 will keep the effect on other related resources the same. The number of threads is chosen to be equal to the number of cores that use the same targeted resource.

For example, to characterize a multithreaded application for the shared L2 cache for the Intel-Yorkfield platform shown in Figure 3, the application is run solely in two configurations. Each configuration runs the number of threads equal to the number of cores sharing one L2 cache, which is two in this case. In the *baseline* configuration,

(a) Baseline Configuration (b) Contention Configuration

Fig. 3. Configurations for characterizing a multithreaded application for shared L2 cache contention.

two application threads are mapped onto the two cores that use a separate L2 cache, for example, C0, C2 (shown in Figure 3(a)) or C1, C3. In the *contention* configuration, the threads are mapped to the cores that use the same cache so that the threads have potential cache contention, for example, C0, C1 (shown in Figure 3(b)) or C2, C3. As the methodology characterizes the application for shared L2 cache contention, intra-application L1 cache contention is avoided by allowing only one thread to access one L1 cache and using same number of L1 caches in both configurations. The FSB contention is kept unchanged between configurations by choosing Intel-Yorkfield, which has one FSB.

The application's performance is measured in each of the configurations and the characteristics of the multithreaded application is then determined by comparing the performance. If the application performance improves in the second configuration, the application is characterized to have data sharing in the cache. If the performance degrades, the application is characterized to contend for the shared cache. Thus, the methodology identifies the key characteristics and potential behaviors of a multithreaded application for a specific shared resource. These characteristics are later used in the online phase to determine the effective thread mapping to improve an application's performance. The characterization configurations for the other targeted resource of the experimental platforms are summarized in Table III in Section 3.

We use the performance difference in the characterization phase to compute a *sensitivity score* of a multithreaded application for each shared resource in the memory hierarchy on a particular CMP. The sensitivity scores are calculated using the following equation:

$$Sensitivity\_Score = \frac{(Number\_of\_Cycles\_Base - Number\_of\_Cycles\_Contend) * 100}{Number\_of\_Cycles\_Base}. \quad (1)$$

Here, $Number\_of\_Cycles\_Base$ and $Number\_of\_Cycles\_Contend$ are the total number of cycles (by reading hardware performance counter) in the baseline and contention configuration, respectively, which represent the application performance.

The sensitivity score captures the characteristics of the application for shared caches and memory bandwidth. The score is represented as a floating-point number, which has both a sign and magnitude. The sign indicates whether the application's performance improves (positive sign) or degrades (negative sign) as its threads share a particular resource. For example, in Table IV, *canneal* has a positive sensitivity score for the L2 cache on Intel-Yorkfield, indicating that *canneal*'s performance improves in the contention configuration. *Streamcluster* has a negative sensitivity score for FSB on Intel-Harpertown, which indicates its performance degrades when its threads are mapped to use the same FSB and require more FSB bandwidth. Therefore, from the sign of the sensitivity score, we can identify the key characteristics of an application as

to whether it benefits from certain resource sharing. On the other hand, the magnitude indicates the degree of the application's sensitivity for a specific shared resource. The higher the magnitude, the more sensitive the application is to sharing that resource. For example, *canneal* has a higher magnitude of sensitivity score and is more sensitive to L2 cache sharing than *dedup* because *canneal* accesses more shared data. From the magnitude of the sensitivity score, we determine how much an application benefits or is penalized from certain resource sharing.

A *sensitivity vector* of a multithreaded application is a vector containing the sensitivity score for each shared resource considered on a particular CMP. For example, if the platform has N types of shared resources, then the sensitivity vector of an application is an N-element vector. This vector is used as an input to the ReSensor thread-mapping algorithm, described in the next section.

2.3. Mapping: The ReSensor Algorithm

To maximize the performance of a workload and mitigate contention for the shared resources, it is critical to determine the thread mapping, considering the characteristics of the applications and the underlying architecture of the platform. Existing thread-mapping techniques map an application by considering its characteristics in the presence of a corunner [Mars et al. 2010; Zhuravlev et al. 2010]. On the other hand, we determine the thread mapping that maps a multithreaded application's threads in the presence of *any* corunner(s) by utilizing the characteristics determined without considering the presence of the corunner(s). As sensitivity scores identify the key characteristics of multithreaded applications for the shared resources on a particular platform, the scores are used to determine the effective thread mapping of each application in a workload to optimize performance by mitigating contention.

Depending on the number of applications in a workload and number of shared resources at a particular memory hierarchical level on a platform, there are two scenarios. The scenarios and intuition behind the algorithm are described as follows.

Scenario 1: There are the same or more shared resources at a particular memory hierarchical level than the number of applications in the workload at a particular time. As the platform has enough shared resources such that each application can use separate shared resources, the thread-mapping algorithm should allocate the shared resources to *one application* at a time by considering its sensitivity score. For example, if an application has contentious behavior and is very sensitive for a particular resource, the application threads should be mapped on to the cores that use separate resources.

Scenario 2: There are more applications in the workload than the number of shared resources at a particular memory hierarchical level. In this case, as the platform does *not* have enough shared resources such that each application can use separate shared resources, the thread-mapping algorithm should prioritize the applications and map *multiple applications* with complementary behavior together to use the same resource, favoring the more sensitive application. The more sensitive application should be prioritized because its performance has a higher impact on the workload's overall performance. The magnitude of the sensitivity score represents the extent of how an application's performance benefits or is penalized from certain thread mapping and sensitivity for a particular resource sharing. Therefore, the prioritization can be determined by considering the magnitude of the applications' sensitivity scores. For example, when there are more applications that have to be mapped than the number of shared caches, the application threads should be mapped such that the most cache-contentious application shares the same cache with the least cache-contentious or cache-sensitive application so that the most contentious application does not degrade the least cache-contentious/sensitive application's performance significantly.

ALGORITHM 1: ReSensor Algorithm: Mapping application threads based on resource sensitivity

```
1: INPUT: Workload WL, Topology of the experimental platform P, Sensitivity vector SV of
   the applications on P
2: nApps ← number of multithreaded applications in WL
3: [Apps] ← multithreaded applications in WL
4: for each level MHL in the memory hierarchy of P do
5:    R ← shared resource at MHL
6:    NR ← number of R at MHL
7:    [C+] ← set of cores that use or share the same R on P
8:    [C−] ← set of cores that do not use or share the same R on P
9:    [SS] ← SV[R] of the applications in [Apps]
10:   sort [SS] array in descending order of the magnitude of the sensitivity scores and
      rearrange [Apps] accordingly
11:   if NR >= nApps then
12:      /* Case 1: equal or more shared resources than the number of applications */
13:      for ( i=0; i < nApps ; i++ ) do
14:         if SS[i] > 0 AND [C+] has available core(s) then
15:            map Apps[i]-threads on the available cores from [C+]
16:         else if SS[i] < 0 AND [C−] has available core(s) then
17:            map Apps[i]-threads on the available cores from [C−]
18:         else
19:            /* [C+] or [C−] does not have available core(s) */
20:            map Apps[i]-threads on any core on P
21:         end if
22:      end for
23:   else
24:      /* Case 2: fewer shared resources than the number of applications */
25:      for ( i=0; i < nApps/2 ; i++ ) do
26:         if SS[i] > 0 AND [C+] has available core(s) then
27:            map Apps[i]- and Apps[nApps − i − 1]-threads on the available cores from [C+]
28:         else if SS[i] < 0 AND [C−] has available core(s) then
29:            map Apps[i]- and Apps[nApps − i − 1]-threads on the available cores from [C−]
30:         else
31:            /* [C+] or [C−] does not have available core(s) */
32:            map Apps[i]- and Apps[nApps − i − 1]-threads on any core on P
33:         end if
34:      end for
35:   end if
36: end for
```

Considering these scenarios, we developed an algorithm, ReSensor, that determines the thread mappings of multithreaded applications by utilizing their characteristics for a particular shared resource. The algorithm maps the application threads in the presence of any number of corunning multithreaded applications to maximize performance by exploiting the application's sensitivity scores on a particular platform.

Algorithm 1 contains the pseudo-code of the ReSensor algorithm for mapping threads from a workload consisting of any number of multithreaded applications on a particular platform P. Platform P can have shared resources in multiple levels of the memory hierarchy. In most platforms, a resource lower in the memory hierarchy contains multiple numbers of resources that are at a higher level in the memory hierarchy. Once the mapping with respect to the resources lower in the memory hierarchy is determined, the mapping with respect to resources at a higher memory hierarchy can be easily determined because the number of thread-mapping configurations reduces to half. In

addition, if we consider the resources from the top of the memory hierarchy to the bottom, the mapping determined on the basis of a resource at a higher level can violate the mapping that will be determined based on the characterization of the shared resource in the lower level. Therefore, ReSensor considers the shared resources from the bottom to the top of the memory hierarchy, that is, from memory bus or memory controller to the shared caches, to determine the thread mappings.

For example, Intel-Harpertown (shown in Figure 5(b)) has four L2 caches and two FSB connections as the shared resources. Assume we have to map an application A with two threads on this platform. A has a negative sensitivity score for L2 cache and positive sensitivity score for FSB. From the sensitivity scores, we can conclude that A's performance improves when A is mapped on the cores that use the same FSB and separate L2 caches. If ReSensor determines the thread mapping of the application by considering the characteristic of the L2 cache (higher at the memory hierarchy) first, then A can be mapped on the cores that use separate cache, any one from the 24 possible thread-mapping configurations. If the mapping $\{C0, C1\}$ is chosen, then this mapping uses a separate cache to avoid cache contention. But at the same time, this mapping causes A to use separate FSB connections, which violates the mapping that leverages A's FSB characterization to use the same FSB. Therefore, the mapping $\{C0, C1\}$ can degrade A's performance. On the other hand, if ReSensor determines the thread mapping of the applications by considering the characteristic of FSB (lower in the memory hierarchy) first, then A can be mapped on cores that use same the FSB connection, which reduce the number of mapping configurations for the L2 cache to eight. Therefore, ReSensor considers each shared resource R from the bottom of the memory hierarchy to the top (line 4). For the Intel-Harpertown example, ReSensor considers the FSB first and then L2 cache characteristics to determine the final thread mapping.

Next, ReSensor counts the number of shared resources, NR, at each memory hierarchical level (MHL) (line 6). It computes two arrays: the set of cores that share or use the same R, $[C_+]$, and the set of cores that do not share the same R, $[C_-]$ (lines 7, 8). These two arrays are later used to look up the cores on which the threads will be mapped. It collects the sensitivity scores of the applications for R into $[SS]$ array (line 9). To maximize the workload's performance in terms of the average response time and throughput (defined in Section 3), the application that has the highest sensitivity for R should be prioritized (as described in Scenario 2 previously) and should have its thread mapping earlier than the least sensitive applications. To ensure the prioritization, the algorithm sorts the $[SS]$ array in the descending order of the sensitivity score's magnitude and rearranges the applications in the $[Apps]$ array accordingly (line 10). There are two cases to consider by ReSensor, corresponding to the two scenarios described earlier.

Case 1: There is the same or more shared resources than the number of applications in the workload (line 11). In this case, P has enough resources to be allocated to each application for isolated execution. The sign of the sensitivity score represents if sharing R will improve the application performance. Therefore, ReSensor determines the thread mapping of each application by considering the sign of the sensitivity score for the shared resource at that level. If the application's sensitivity score is positive (line 14), it has sharing behavior and its performance improves when its threads execute while using or sharing the same resource R. ReSensor maps the application threads on the available cores (the cores on which any thread is not mapped yet) from $[C_+]$ considering its sharing characteristics (line 15). Sharing the same resource, especially the caches, also reduces the number of memory transactions for maintaining the cache coherency and results in better performance. If the application's sensitivity score is negative (line 16), its performance degrades when the application threads use

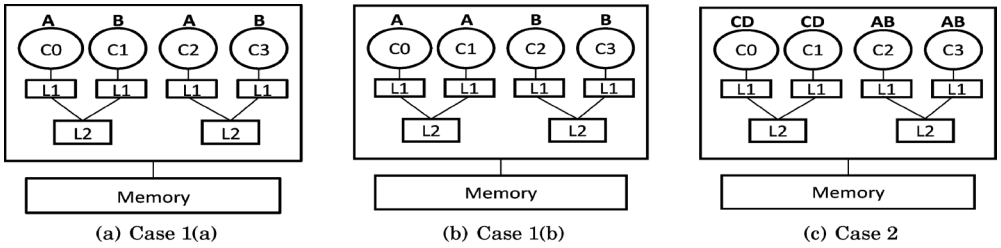


Fig. 4. Mapping decision for the two cases.

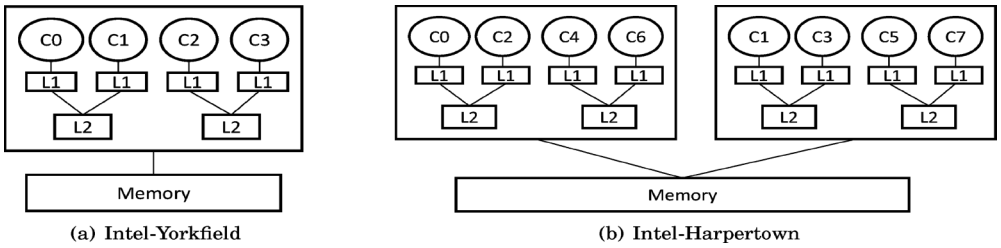


Fig. 5. Topology of experimental platforms.

the same resource R because of intra-application contention for R. Therefore, to avoid and mitigate the contention, ReSensor maps the threads on the available cores from $[C_-]$ (line 17) so that the threads use separate R. If there are no available cores from $[C_+]$ or $[C_-]$, ReSensor maps threads on any core on P (line 20). As ReSensor maps the application threads considering its performance characterization for each shared resource, it always guarantees the mapping that improves the workload's performance.

For example, consider a workload that has two multithreaded applications, A and B. Assume both applications have negative sensitivity scores of $-a$ and $-b$, respectively, and contentious behavior for the shared L2 cache on the quad-core platform (e.g., Intel-Yorkfield), shown in Figure 4. Here, both *nApp* and *NR* equal 2 and $[C_+]$ = [{C0,C1}, {C2,C3}] and $[C_-]$ = [{C0,C2}, {C1,C3}]. As both applications have contentious behavior for the shared L2 cache, they are both mapped to the cores from $[C_-]$. A is mapped on $C0$ and $C2$, and B is mapped on the two remaining cores from $[C_-]$, as shown in Figure 4(a). These applications, with negative *sensitivity scores*, suffer from relatively higher intra-application contention than interapplication contention for cache resources [Dey et al. 2011], and the performance degrades more when the sibling threads are colocated with each other compared to when threads are colocated with the threads from the corunning application to use the same cache. Therefore, for such workloads with negative sensitivity scores, it is beneficial to share the resource with the corunner's threads than sharing the resource with its sibling threads to improve the workload's overall performance.

Let us now consider application A, which has a positive sensitivity score $+a$ and sharing behavior, and application B, which has a negative sensitivity score $-b$ and contentious behavior for the shared L2 cache on the same platform. Similar to the last example, both *nApp* and *NR* equal 2 and $[C_+]$ = [{C0,C1}, {C2,C3}] and $[C_-]$ = [{C0,C2}, {C1,C3}]. After sorting, if $|a| > |b|$, ReSensor maps A's threads on the cores from $[C_+]$ (C0 and C1) to take advantage of the sharing characteristics and B's threads on the available cores from $[C_-]$ (C2 and C3), shown in Figure 4(b). This mapping may degrade B's performance as it forces B's contentious threads to use the same cache. Because B is comparatively less sensitive for the L2 cache, the degradation is less

significant than A's degradation if the opposite thread mapping was selected. As ReSensor prioritizes A's characteristic, A has better performance improvement compared to the alternative mapping when it shares the same L2 cache with B's threads, and the overall performance of the workload improves.

Case 2: There are more applications than the number of resources at a particular level of the memory hierarchy. In this case, P does *not* have enough resources to be allocated to each application for isolated execution. Therefore, ReSensor needs to select more than one application to use the same resource R. ReSensor selects the most sensitive application with the least sensitive application to share the same resource and chooses the mapping that benefits the most sensitive application. ReSensor prioritizes the most sensitive applications because its performance has a higher impact on the workload's overall performance than that of the least sensitive applications, to maximize the workload's overall performance. Lines 25–34 contain the pseudo-code for mapping applications in such cases. After sorting $[SS]$ in descending order, the algorithm maps the most sensitive (highest magnitude) application from the first half of [*Apps*] with the least sensitive ones (lowest magnitude) from the second half of [*Apps*], favoring the characteristics of the most sensitive application. If the sensitivity score of the most sensitive application is positive (line 26), it maps its threads and least sensitive application threads to the available cores from $[C_+]$ (line 27) favoring the sharing characteristics of the most sensitive application for R. If the sensitivity score of the most sensitive application is negative (line 28), ReSensor maps its threads and the least sensitive application threads to the available cores from $[C_-]$ (line 29) to avoid the intra-application contention among sibling threads of the most sensitive applications. If there are no available cores from $[C_+]$ or $[C_-]$, ReSensor maps threads on to any core on P (line 32). The mapping prioritization toward the most sensitive application does not affect the least sensitive application's performance significantly and results in overall performance improvement of the workload.

For example, let us consider a workload that has four applications [*A, B, C, D*] to be mapped on the same platform from the previous example. The sensitivity scores of the four applications for the L2 cache after sorting is $[+c, -a, +b, -d]$, where $|c| > |a| > |b| > |d|$. ReSensor selects the most sensitive application C and the least sensitive application D to map them together. As C has a positive sensitivity score, ReSensor maps C and D on the cores from $[C_+]$ (C0 and C1), prioritizing C's sharing behavior for the L2 cache. Then ReSensor maps A and B on the remaining cores (C2 and C3). The final mapping is shown in Figure 4(c). Because C is the most sensitive application, this mapping favors C's characteristic to ensure its improved performance. D being the least sensitive, the mapping does not degrade D's performance significantly and improves the workload's overall performance.

3. EXPERIMENTS AND RESULTS

To evaluate ReSense's effectiveness in mapping the threads of applications from a workload, we chose two multithreaded benchmark suites, PARSEC [Bienia et al. 2008] and NAS Parallel Benchmarks (NPBs) [Bailey et al. 1991]. We used the PARSEC benchmark suite as it is composed of multithreaded applications designed to be representative of next-generation shared-memory programs for multicore architectures. We used the NAS benchmark suite as it consists of representative applications for high-performance computing. Both benchmark suites have multithreaded applications of diverse characteristics. We used 12 benchmarks from PARSEC-2.1 shown in Table IV. We used the largest *native* input set for the PARSEC benchmarks. We used nine benchmarks from NPB-OMP-3.3 described in Table V. We used the Input B for the benchmark DC and input D for all other benchmarks as these were the largest inputs for the experimental platforms. We did not use BT from NPB as it did not run

Table II. Configuration of the Experimental Platforms

Platform	Topology	Linux kernel /GCC version	Number of cores/contexts	Memory system	Base of characterization
Intel-Yorkfield	Figure 5(a)	2.6.25/4.2.4	4/4	private 32KB L1 2 shared 6MB L2 2GB memory	L2 cache
Intel-Harper-town	Figure 5(b)	2.6.30/4.2.4	8/8	private 32KB L1 4 shared 6MB L2 32GB memory	L2 cache, FSB
Intel-Xeon	Figure 6(a)	2.6.32/4.4.3	32/64	private 32KB L1 private 256KB L2 4 shared 18MB L3 250GB memory	L3 cache + Memory controller (MC)
AMD-Opteron	Figure 6(b)	2.6.32/4.4.3	48/48	private 64KB L1 private 512KB L2 8 shared 5MB L3 95GB memory	L3 cache, Memory socket

multiple threads. All the applications were statically compiled using the GCC compiler with optimization level 3, and the OpenMP benchmarks were compiled using *fopenmp* flag.

We chose four different experimental platforms: Intel-Yorkfield, Intel-Harpertown, Intel-Xeon, and AMD-Opteron. Table II describes the configurations. We chose Intel-Yorkfield to characterize benchmarks for the L2 cache or last-level cache (LLC) contention. We selected Intel-Harpertown to characterize benchmarks based on contention for the FSB bandwidth. Intel-Xeon is chosen to characterize benchmarks based on contention for the L3 cache and on-chip memory controller (MC). We chose AMD-Opteron to characterize benchmarks on a different microarchitecture and measure contention for LLC and memory socket connection. The selected machines represent a range of different microarchitectures, topologies, and types of shared resources in the memory hierarchy and provide evidence of the generality by ReSense.

ReSense is implemented as a user-level virtual execution manager using REEact's framework [Wang et al. 2012]. We chose this framework because it is customizable, is especially designed for CMPs, and has very low (less than 3%) runtime overhead. ReSense uses several services provided by REEact to detect the creation and termination of an application thread, including detecting the start and finish of an application and pinning application threads on specific cores.

We compare our experimental results with the native OS, as after an extensive search for similar work, we found it is the only viable option. Most of the previous thread-mapping or scheduling work focused on single-threaded applications, and extensions to accommodate multithreaded applications were not obvious. The prior research on multithreaded applications, on the other hand, focused on optimizing energy, choosing the thread count, minimizing lock contention, or optimal core allocation, goals and techniques that are different than mitigating shared-resource contention and improving application performance by determining the thread mapping. This work is the first to have management of contention for shared-memory resources for multiple multithreaded applications from dynamic workloads via thread mapping as the goal. We believe comparing to the native OS is a fair comparison as recent operating systems, including the one we used, consider an application's cache and memory behavior in scheduling [Siddha and Mallick 2005]. Therefore, to evaluate ReSense's effectiveness, we ran the experiments in two configurations. In the first or baseline configuration, we ran the workloads under the OS's control where the native OS determines the thread

Table III. Characterization Configurations on the Experimental Platforms

Platform: Targeted resource	Number of threads	Baseline configuration	Contention configuration
Intel-York field: L2 cache	2	Maps on cores C0 and C2	Maps on cores C0 and C1
Intel-Harper-town: L2 cache	2	Maps on cores C0, C4	Maps on cores C0, C2
Intel-Harper-town: FSB	4	Maps on cores C0, C2, C1, C3	Maps on cores C0, C2, C4, C6
Intel-Xeon: L3 cache+MC	8	Maps on cores C0-C3 and C8-C11	Maps on cores C0-C7
AMD-Opteron: L3 cache	6	Maps on cores C0-C2 and C6-C8	Maps on cores C0-C5
AMD-Opteron: Memory socket	12	Maps on cores C0-C5 and C12-C17	Maps on cores C0-C11

mapping (called *OS mapping*). In the second configuration, we ran the workloads under ReSense's control, using the mapping determined by ReSensor (called ReSense mapping). In all experiments, the number of workloads were chosen to ensure statistical significance for *t-test* (see Section 3.5). The evaluation metrics, *average response time* and *throughput,* are computed according to the following equations, and the results are normalized with respect to the native OS. Here, n is the total number of applications in a workload and *Execution Time* is the average wall-clock execution time of an application.

$$Average\ Response\ Time = \frac{\sum_{i=1}^{n} ExecutionTime_i}{n} Throughput = \sum_{i=1}^{n} \frac{1}{ExecutionTime_i}$$

3.1. Sensitivity Score

To characterize the benchmarks and determine their sensitivity scores, we ran each benchmark in two configurations for each shared resource on the four experimental platforms, as described in Section 3.1. The characterization configurations for each targeted resources on the platforms are described in detail in Table III. The number of threads for each benchmark was chosen to be equal to the number of cores that shared the targeted resource on the platform. The baseline configuration mapped the threads to the cores such that the benchmark used two shared targeted resources. The contention configuration mapped the threads such that the benchmark used the same targeted resource. Both configurations kept the effect of contention on the other resources the same so that only the effect of contention for the targeted resource on an application's performance could be isolated.

The application's sensitivity scores for the targeted resources were calculated using Equation 1. Tables IV and V show the sensitivity scores of the PARSEC and NPB benchmarks, respectively. We did not determine sensitivity scores of the NPB benchmarks on Intel-Yorkfield and Intel-Harpertown as these platforms are too resource constrained for these long-running applications.

In all our experiments described in the following sections, we configured each benchmark to run with two, four, eight, and six threads on Intel-Yorkfield, Intel-Harpertown, Intel-Xeon, and AMD-Opteron, respectively. We represent each workload, WL_n, that consists of n applications, as $\{ts_1(BM_1,k_1)\ ts_2(BM_2,k_2)\ ...\ ts_n(BM_n,k_n)\}$, which means at time-stamp ts_i, BM_i arrives and executes for k_i iterations. Depending on the size of the workload, the time stamps are randomly chosen between 0 and 400 seconds, and the benchmarks are randomly picked from PARSEC and NPB benchmark suites. The

Table IV. Sensitivity Scores of the PARSEC Benchmarks

Platform	Intel-Yorkfield	Intel-Harper-town	Intel-Harper-town	Intel-Xeon	AMD-Opteron	AMD-Opteron
Benchmarks	L2 cache	L2 cache	FSB	L3 cache +MC	L3 cache	Memory socket
blackscholes (BS)	0.0354	−0.0922	−0.1277	−7.5441	−1.1348	0.1289
bodytrack (BT)	2.2210	3.412	0.2922	−4.6291	0.5649	0.0558
canneal (CN)	4.4049	4.6012	−3.3664	8.1034	6.7756	8.5197
dedup (DD)	2.4294	1.7702	−0.2925	−5.4493	2.2727	1.1811
facesim (FA)	−0.5406	−6.6455	−1.1869	−6.0272	3.8579	24.148
ferret (FE)	0.2665	−0.4081	−0.6787	−3.0600	4.0712	1.1431
fluidanimate (FL)	0.4920	1.9047	−1.7727	−7.8056	−1.0396	0.3836
freqmine (FQ)	2.065	−0.4574	−0.3188	−8.6898	−0.2647	0.0623
raytrace (RT)	0.0196	0.7365	−0.0036	−0.4280	1.1278	−0.0463
streamcluster (SC)	9.9298	9.1566	−11.3983	14.4905	10.9319	36.148
swaptions (SW)	0.2762	0.2570	0.0061	−6.3446	0.6461	0.2931
vips (VP)	0.2803	1.1600	0.4082	−11.2819	0.0897	0.1766

Table V. Sensitivity Scores of the NPB Benchmarks

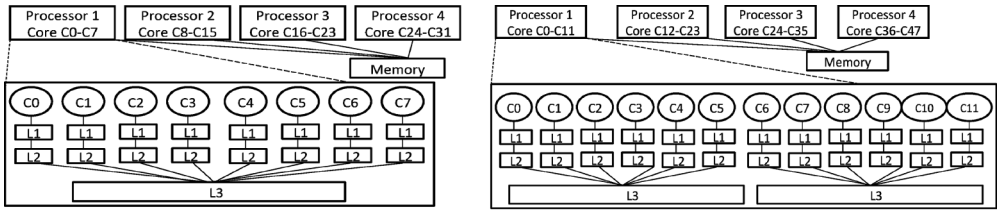
Platform	Intel-Xeon	AMD-Opteron	AMD-Opteron
Benchmarks	L3 cache +MC	L3 cache	Memory socket
IS.D	−12.4783	−52.0424	−10.4446
DC.B	−7.1677	−11.0202	1.9157
SP.D	−21.5190	−10.1729	21.6309
LU.D	−25.8932	−22.3693	1.1384
FT.D	−31.2067	−233.2549	−38.6669
CG.D	1.5424	−107.5569	17.9871
MG.D	−11.9599	−66.5676	−43.0097
EP.D	−11.4738	0.0476	0.2832
UA.D	−23.3012	−19.5132	0.9321

number of iterations is randomly chosen between 1 and 10. These parameters for the different-sized workloads are described in the corresponding experiments.

3.2. Evaluation: Small Dynamic Workloads

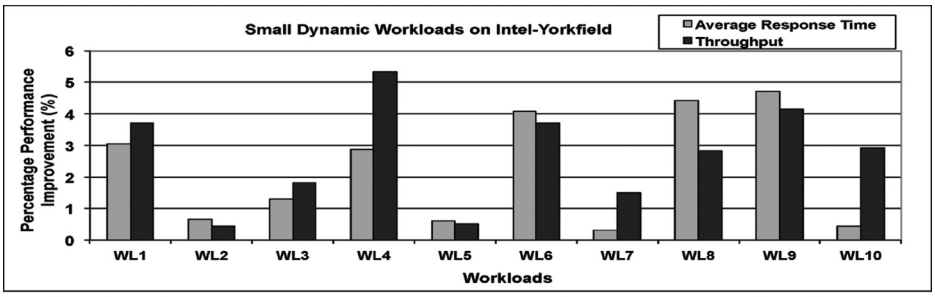
To evaluate if ReSense determines effective thread-to-core mappings of the multi-threaded applications by using the sensitivity scores and improves the workload's overall performance, we first used *Small* dynamic workloads. We randomly selected three PARSEC benchmarks. The first two benchmarks started execution simultaneously. After the second benchmark finished, the third benchmark executed for a random i_3 iterations. We chose to run the third benchmark to evaluate ReSense's effectiveness at dynamically adjusting the mapping based on the new benchmark's sensitivity scores. Each workload had simultaneously executing two or one benchmark at some point in time.

In Figure 7(a), we observe that the average response time and throughput of most workloads improve by up to 4.75% and 5.32% on Intel-Yorkfield. The improvement indicates that ReSense adjusts the benchmarks' thread mappings dynamically in the presence of a corunner and the corresponding sensitivity scores. The workloads, *WL2*,

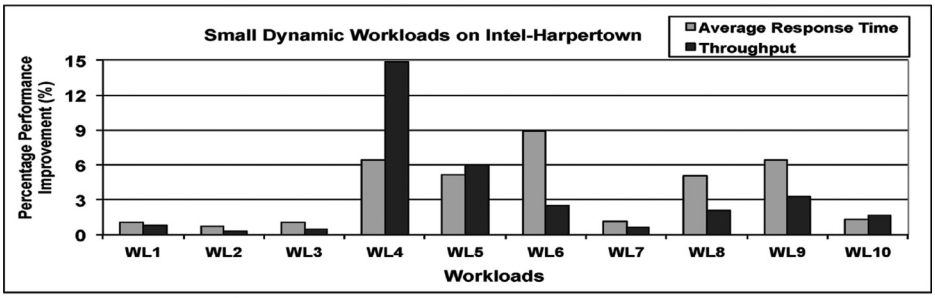


(a) Intel-Xeon (b) AMD-Opteron

Fig. 6. Topology of experimental platforms.



(a) Results on Intel-Yorkfield



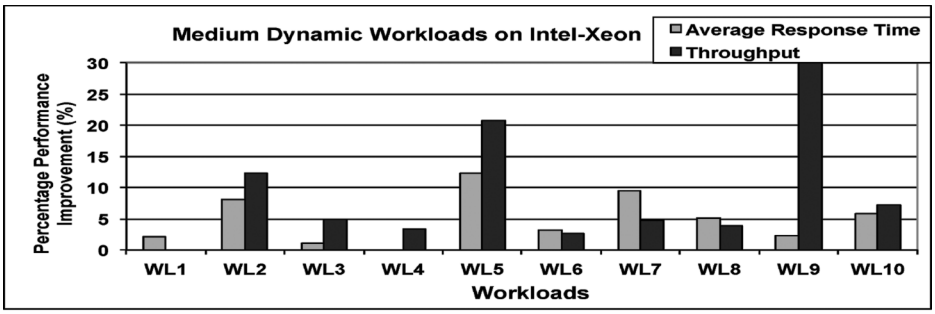
(b) Results on Intel-Harpertown

Fig. 7. Performance results of *Small* dynamic workloads, normalized to the native OS (ReSense performs better than the OS).

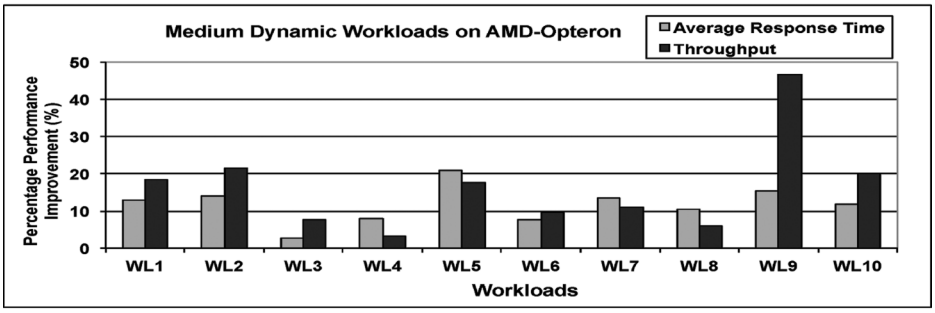
WL3, *WL5,* and *WL7*, consist of at least two benchmarks that are not L2 cache sensitive, and thus the performance difference between the OS and ReSense is small.

In Figure 7(b), we observe that the average response time and throughput improve by up to 8.89% and 14.88% for most workloads on Intel-Harpertown, especially those containing *SC*, *FA*, *FL,* and *CN*. These benchmarks have the highest sensitivity scores for FSB and are more memory intensive than the other benchmarks. *SC*, *FL,* and *CN* also have data sharing in the L2 cache. Because ReSensor considers the benchmarks' sensitivity scores for both L2 cache and FSB, ReSense maps the threads to use both FSB's bandwidth and the same cache, resulting in performance improvement. As the benchmarks in other workloads do not have high sensitivity scores for FSB, their performance differences between OS and ReSense are small.

ReSense improved both average response time and throughput by up to 16.52% and 13.70% on Intel-Xeon and by up to 27.03% and 19.97% on AMD-Opteron, respectively, which indicates that ReSense effectively adjusts the thread mappings depending on the benchmarks' sensitivity scores and the underlying platform's topology. The performance improvements on the more powerful machines (Intel-Xeon and AMD-Opteron)



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Fig. 8. Performance results for *Medium* dynamic workloads, normalized to the native OS (ReSense performs better than the OS).

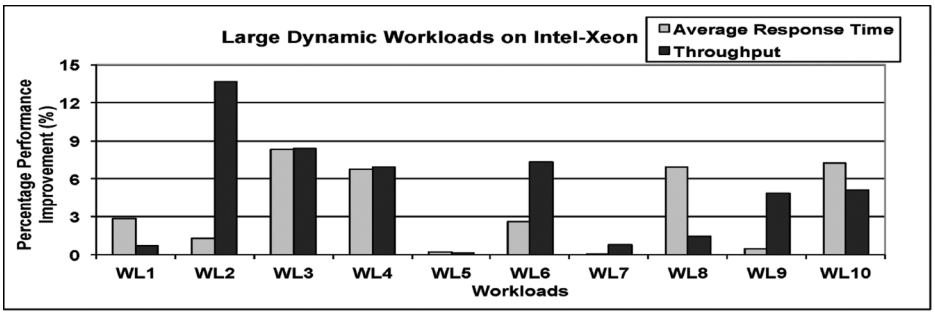
are much higher than that of the less powerful machines (Intel-Yorkfield and Intel-Harpertown). This discrepancy between the performance gains of the more and less powerful machine is caused by the benchmarks, which are more sensitive to the shared resources on the more powerful machines.

To summarize, by utilizing an application's sensitivity score, ReSense effectively maps threads from dynamic pairs of multithreaded applications and improves response time and throughput.

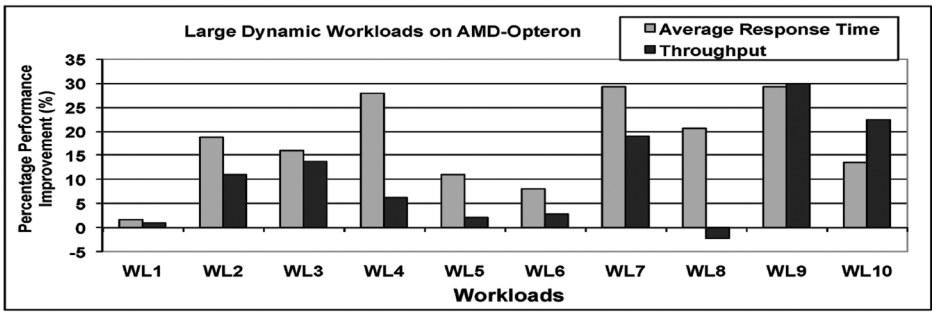
3.3. Evaluation: Medium Dynamic Workloads

To demonstrate that ReSense effectively maps threads from workloads consisting of multithreaded applications, we ran experiments with *Medium* workloads. The workloads consisted of four randomly selected benchmarks from both PARSEC and NPB to have a diverse set of applications. Two benchmarks started simultaneous execution at the beginning, and the third and fourth benchmark arrived after random intervals. A benchmark in the workload continued to execute and re-execute for a number of times. Thus, even when the third and fourth benchmarks arrived and executed, the first and second benchmarks were still executing. If any benchmark finished, it restarted immediately without any intermediate delay. Therefore, on average more than 50% of the time there were four simultaneously executing multithreaded applications in the system for *Medium* workloads.

In Figure 8, we observe that ReSense improves the average response time and throughput by up to 12.38% and 30% on Intel-Xeon and 20.89% and 46.56% on AMD-Opteron, respectively, over the native OS. From the improvements in both metrics for every workload, we conclude that ReSense effectively maps multithreaded applications



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Fig. 9. Performance results for *Large* dynamic workloads, normalized to the native OS (ReSense performs better than the OS).

from very diverse dynamic workloads and dynamically adjusts the thread mappings as the benchmarks arrive and execute nondeterministically.

On Intel-Xeon, all the benchmarks in *WL*1, *WL*3, *WL*4, and *WL*8 have negative sensitivity scores for (L3 cache+MC). Therefore, ReSense maps the sibling threads on separate L3 caches to reduce the cache contention among threads. Under OS mapping, the sibling threads are randomly mapped on the cores using separate L3 caches and the mapping determined by ReSense and OS is similar. Therefore, the performance difference between ReSense and the OS is small.

To summarize, ReSense improves the workload's average response time and throughput by dynamically adjusting the thread mappings of the multithreaded applications in the presence of multiple dynamic corunners using sensitivity scores.

3.4. Evaluation: Large Dynamic Workloads

To evaluate ReSense's scalability and the capability of handling more multithreaded applications and threads in a more dynamic environment, we ran experiments with *Large* dynamic workloads. The workloads were composed of randomly selected eight benchmarks from both PARSEC and NPB to create diversity. Each workload had one to eight benchmarks simultaneously executing at some point in time.

In Figure 9, we observe that ReSense improves the average response time and throughput of the workloads up to 8.29% and 13.65%, respectively, on Intel-Xeon and 29.34% and 29.86%, respectively, on AMD-Opteron, over the native OS. On Intel-Xeon, all workloads show improvements for both metrics. The mapping decision of ReSense is similar to the OS mapping for *WL*1, *WL*5, and *WL*7. Therefore, the performance differences between the OS and ReSense are small. On AMD-Opteron, most workloads have high performance improvements for both metrics. The benchmarks in *WL*1,

Table VI. Confidence Interval of Performance Improvements for Three Workloads

Dynamic workloads	Metrics	Intel-Yorkfield	Intel-Harpertown	Intel-Xeon	AMD-Opteron
Small	Average response time	2.25 ± 1.58 p-val = 0.0005	3.73 ± 2.68 p-val = 0.005	14.15 ± 13.4 p-val = 0.005	7.45 ± 6.18 p-val = 0.0005
	Throughput	2.70 ± 1.64 p-val = 0.0005	3.27 ± 3.95 p-val = 0.025	4.89 ± 4.03 p-val = 0.005	11.70 ± 6.39 p-val = 0.0005
Medium	Average response time	–	–	4.94 ± 4.18 p-val = 0.005	11.70 ± 7.45 p-val = 0.0005
	Throughput	–	–	9.005 ± 9.74 p-val = 0.01	16.17 ± 14.80 p-val = 0.005
Large	Average response time	–	–	3.67 ± 3.37 p-val = 0.005	17.60 ± 14.19 p-val = 0.005
	Throughput	–	–	4.95 ± 4.45 p-val = 0.0005	10.58 ± 10.8 p-val = 0.01

particularly *RT*, *SW,* and *BS,* are not very sensitive to the shared resources. ReSense maps *IS*'s threads on separate processors and L3 caches, and OS mapping also maps the threads randomly on any processor. Because both OS and ReSense map the threads similarly on the cores, the performance difference between the OS and ReSense for *WL*1 is small. The throughput degrades by 2% for *WL*8. ReSense colocates one of the benchmarks in the workload, *FE*, with the long-running NAS benchmarks, causing *FE*'s performance degradation. As *FE* has a very low execution time, it has high impact on the throughput equation. Therefore, even if the average response time improved for this workload, the throughput did not improve over the native OS.

In summary, ReSense manages and determines effective thread mapping for dynamic workloads, consisting of eight randomly selected benchmarks, and improves both average response time and throughput.

3.5. Discussion and Statistical Analysis

Because we used workloads having randomly selected benchmarks in our experiments, we performed a significance test for the reported average response time and throughput. We assumed the null hypothesis that ReSense does not improve the average response time and throughput of the workloads over the native OS. As we performed each experiment in two configurations, using the native OS and ReSense runtime, each experiment has two distributions, *OS* and *ReSense*. We performed a *t-test* to compare these distributions to determine if *OS* is better than *ReSense* in terms of average response time and throughput [Milton and Arnold 2003]. For each dynamic workload on the experimental platforms, we observed that the null hypothesis is rejected with a p-value of at most 0.005 for average response time and 0.025 for throughput. It indicates that the probability of ReSense improving a workload's average response time and throughput over the OS is very high, at least 99.5% and 97.5%, respectively.

We determined the confidence intervals of the mean improvement of the average response time and throughput provided by ReSense over the native OS, shown in Table VI. If the confidence interval for any metric is x ± y with $p - val = z$, it means that at $(1 - z) * 100\%$ confidence interval the mean improvement will range from $x - y$ to $x + y$. From the table, we observe that the lower interval for the average response time is always greater than 0, which indicates that ReSense always improves the average response time of the three dynamic workload sets on the four platforms. For throughput, the lower interval is always greater than 0 except the negligible –0.68% for *Small* on Intel-Harpertown, –0.69% for *Medium* on Intel-Xeon, and –0.22% for *Large* on AMD-Opteron. These negligible negative values are caused by the very small

Table VII. Average Performance Improvements (Positive Values) or Degradation (Negative Values) over the Native OS, for Thread Mappings Using Fixed Positive, Fixed Negative, and Characterization-based Sensitivity Scores

Thread mappings	2-application workload		4-application workload	
	Average response time	Throughput	Average response time	Throughput
C_+	−1.70%	5.57%	14.1%	8.28%
C_-	0.51%	−3.77%	−2.5%	−6.51%
ReSense	5.07%	7.90%	16.77%	11.82%

performance degradation of the benchmark having very small execution time. From the higher interval, we observe that the maximum average response time improvement is 31.79% for *Large* and the maximum throughput improvement is 30.97% for *Medium* on AMD-Opteron.

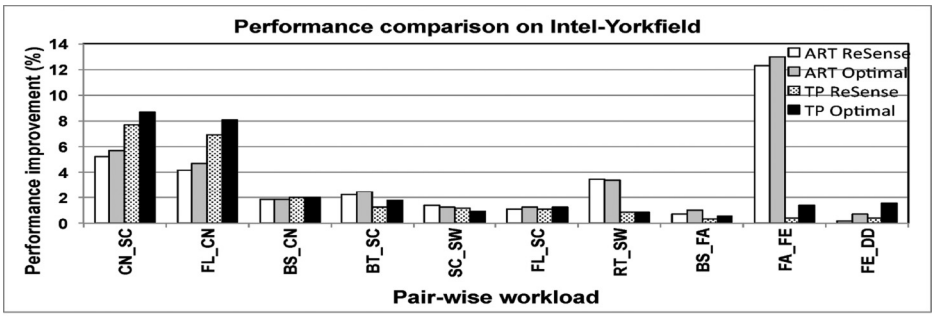
To summarize, the statistical analysis validates ReSense's effectiveness in mapping and improving the performance of multithreaded applications over the native OS.

The improvements over the native OS by ReSense are mainly due to the characterization and use of these characterizations by ReSense. If ReSense does not consider any sensitivity scores (sensitivity scores are 0) to determine the thread mappings, the performance of the workload will be the same as the native OS. To further isolate the benefits of the sensitivity scores, we explored the performance of workloads when the sensitivity scores were set to the same magnitude with a positive or negative sign and ran our algorithm for these two cases. Table VII summarizes these experimental results for the two-application and four-application workloads used in Section 3.6 on Intel-Harpertown and AMD-Opteron, respectively, relative to the native OS. The rows C_+ and C_- show the performance results of the thread mappings determined using the fixed positive and negative sensitivity scores (same magnitude with positive or negative sign), respectively, for all applications in the workloads. The row ReSense shows the performance results of the thread mappings determined using an application's sensitivity scores from the characterization. From the table we observe that C_+ thread mapping degraded application performance for the two-application workloads, and C_- thread mapping degraded application performance for both the two-application and four-application workloads. Therefore, thread mappings determined using fixed positive or negative sensitivity scores do not ensure application performance improvements.

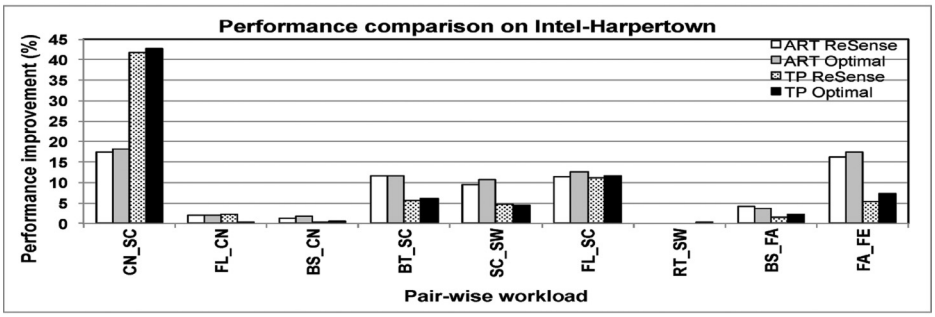
In contrast, with the computed sensitivity scores from the characterization, for the two-application workloads ReSense improved both the workloads' average response time by 5.07% and throughput by 7.9% on average, and for the four-application workloads ReSense improved both the workloads' average response time by 16.77% and throughput by 11.82%, on average, relative to the native OS. Therefore, these results show that the computed sensitivity scores were essential for the improved performance.

3.6. Performance Comparison with Experimentally Determined Optimal Thread Mapping

To evaluate ReSense's effectiveness in contention mitigation, we compared application performance obtained from ReSense and the experimentally determined optimal thread mapping. We experimentally determined the optimal performance of a workload by choosing the minimum average response time and maximum throughput of the optimal thread mapping among all possible thread-to-core mapping configurations. Throughout this section, when we use "optimal," we mean the experimentally determined optimal. Dynamic workloads have $O(r^{n_1} * r^{n_2} *, \ldots * r^{n_n})$ numbers of different thread-mapping configurations, where r is the number of a particular shared resource on a platform, and $n_1, n_2, \ldots, n_n$ are the numbers of executing applications at time $t_1, t_2, \ldots, t_n$, respectively. As dynamic workloads have such a large number of configurations, it is unfeasible to experimentally determine the optimal performance. Therefore, we



(a) Results on Intel-Yorkfield



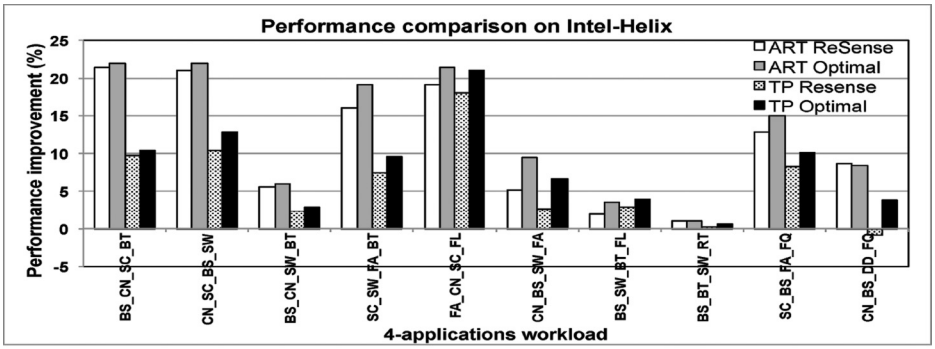
(b) Results on Intel-Harpertown

Fig. 10. Performance comparison between ReSense and experimentally determined optimal thread mapping for pair-wise workloads, normalized to the native OS.

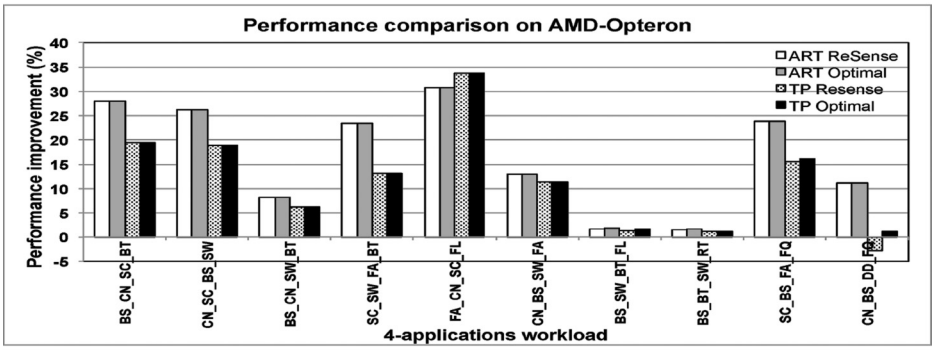
chose to use workloads that have all the benchmarks start at the same time and execute for the same number of iterations so that it was feasible to determine the optimal performance.

For performance comparison on Intel-Yorkfield and Intel-Harpertown, we ran experiments with randomly selected 10 pair-wise workloads from the PARSEC benchmark suites. We did not include the NAS benchmarks in the random selections because those benchmarks have execution times that are too long to finish the experiments for all possible thread-mapping configurations. Figure 10 shows the experimental results relative to the native OS, where the x-axis labels the initials of the benchmarks used as the workload. For each workload, the first two bars show average response times (ARTs) of ReSense and experimentally determined optimal mapping, and the last two bars show throughput (TP) of ReSense and experimentally determined optimal mapping. In both Figures 10(a) and 10(b), we observe that ReSense's performance improvements are very close to that of the optimal. The average ART difference between ReSense and the experimentally determined optimal is 0.27% and 0.18% on Intel-Yorkfield and Intel-Harpertown, respectively. The average TP difference between ReSense and the experimentally determined optimal is 0.49% and 0.10% on Intel-Yorkfield and Intel-Harpertown, respectively. Both the ART and TP differences between ReSense and optimal are negligible on both machines, and we conclude that ReSense always ensures near-optimal performance on these machines.

For performance comparison on Intel-Xeon and AMD-Opteron, we ran experiments with randomly selected 10 four-application workloads from the PARSEC benchmark suites. Figure 11 shows the experimental results relative to the native OS. In both Figures 11(a) and 11(b), we observe that ReSense's performance improvements are very close to that of the experimentally determined optimal. The average ART



(a) Results on Intel-Xeon



(b) Results on AMD-Opteron

Fig. 11. Performance comparison between ReSense and experimentally determined optimal thread mapping for four-application workloads, normalized to the native OS.

difference between ReSense and the experimentally determined optimal is 1.49% and 0.05% on Intel-Xeon and AMD-Opteron, respectively. The average TP difference between ReSense and the experimentally determined optimal is 2.08% and 0.49% on Intel-Xeon and AMD-Opteron, respectively. Both the ART and TP differences between ReSense and the experimentally determined optimal are very small on both machines. Therefore, we can conclude that ReSense performs very competitively with the optimal mapping.

4. RELATED WORK

There have been a few works on scheduling multiple multithreaded applications. [Bhadauria and McKee 2010] scheduled threads from multiple multithreaded applications at a time quantum to optimize thread throughput and energy. At a particular time quantum, their algorithm selected a number of threads from each application in the workload and the applications to run together by considering an application's cache miss rates (FAIRMIS policy) or bus occupancy (FAIRCOM policy). They did not consider an application's characteristics for both cache and bus in the same policy. On the other hand, ReSense mitigates contention for all the shared resources in the memory hierarchy, considering applications cache and bus characteristics at the same time. [Pusukuri et al. 2013] allocated cores to multithreaded applications by using an application's cache and lock contention characteristics that were determined in the presence of a corunner. In the presence of r corunners, the number of characterizations grows polynomially, $O(n^r)$, for n applications. Their system determined the number

of cores to be allocated to each application by using a supervised learning technique, which requires more offline analyses compared to ReSense. On the other hand, ReSense does not have any training phase and characterizes application without considering any corunner. This solo characterization has linear complexity of $O(n)$, which is much lower than $O(n^r)$.

Pusukuri et al. [2011] described scheduling policy for minimizing lock contention for multithreaded applications. Emani et al. [2013] determined the thread count to improve an application's performance in the presence of external workloads. Das et al. [2013] described application-to-core mapping for NoC systems. Garcia et al. [2012] investigated dynamic scheduling for "embarrassingly" parallel applications for CMPs. Broquedis et al. [2010] described a scheduling framework for OMP applications. These works primarily pursued the goals to minimize lock contention, optimize a single application's performance, minimize communication overhead, or focus on core allocation, NoC, and data-parallel application. These goals are different than the goal pursued by ReSense, which targets optimizing the average response time and throughput of *every* multithreaded application from a dynamic workload by determining the thread mappings. Chen et al. [2007] proposed scheduling threads that share data to use the same cache to improve performance for multithreaded applications. We consider both contention and data sharing in caches to map multithreaded applications.

Jin et al. [2009] characterized parallel applications, and Dey et al. [2011] characterized multithreaded PARSEC applications, when they run alone and with corunners, for shared-resource contention in the memory hierarchy. Kambadur et al. [2012] described methodology to measure interference between data-center applications while they are executing. However, these works have not focused on mitigating contention.

Some research efforts proposed the idea of mapping application threads or managing shared resources based on prior characterization. Mars et al. [2011a, 2011b] and Tang et al. [2011] scheduled and determined the mapping of colocated applications by characterizing the application in the presence of corunners or synthetic workloads. Xie and Loh [2008] classified applications by measuring cache miss rates for a dynamic cache partitioning scheme. Jaleel et al. [2012] mapped applications by using a runtime classification based on cache replacement policy. In this work, ReSense utilizes an application's prior performance characterization for individual shared resources by running it solely, without using any corunning applications or special hardware policies, and is capable of determining the thread mapping for dynamic workloads consisting of multiple multithreaded applications.

There have been several efforts at designing different models to map application threads, including analytical probabilistic models for shared L2 cache [Chandra et al. 2005] and a prediction model for shared resources in simultaneous multithreaded processors [Moseley et al. 2005]. ReSense uses an application's solo performance characterization for each shared resource in the memory hierarchy to predict the effective thread-to-core mapping in the presence of corunners. Several works addressed shared resource management in CMPs via hardware cache partitioning to mitigate contention [Qureshi and Patt 2006; Xie and Loh 2009] and software methods to partition the cache and allocate memory pages [Cho and Jin 2006; Jin and Cho 2009; Tam et al. 2009; Zhang et al. 2009; Soares et al. 2008]. Our approach is compatible and can be combined with both hardware and software cache partitioning techniques to further reduce cache contention and improve performance.

5. CONCLUSION

In this article, we describe the ReSense runtime system, which demonstrates that sensitivity scores for the shared resources in the memory hierarchy are useful in determining the effective thread mappings for multithreaded applications. ReSense does not require

the application's source code modifications. ReSense uses a novel thread-mapping algorithm, ReSensor, to dynamically adjust the thread mappings of three different-sized dynamic workloads and improves the average response time and throughput up to 29.34% and 46.56%, respectively, over the native OS using real hardware.

REFERENCES

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. 1991. The NAS parallel benchmarks. Tech. rep., The International Journal of Supercomputer Applications.

Bhadauria, M. and McKee, S. A. 2010. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the International Conference on Supercomputing*.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. 2010. hwloc: A generic framework for managing hardware affinities in HPC applications. In *Proceedings of the Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'10)*.

Chandra, D., Guo, F., Kim, S., and Solihin, Y. 2005. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

Chen, S., Gibbons, P. B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T. C., and Wilkerson, C. 2007. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*.

Cho, S. and Jin, L. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the International Symposium on Microarchitecture*.

Das, R., Ausavarungnirun, R., Mutlu, O., Kumar, A., and Azimi, M. 2013. Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture*.

Dey, T., Wang, W., Davidson, J. W., and Soffa, M. L. 2011. Characterizing multi-threaded applications based on shared-resource contention. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

Emani, M. K., Wang, Z., and O'Boyle, M. F. 2013. Smart, adaptive mapping of parallelism in the presence of external workload. In *Proceedings of the International Symposium on Code Generation and Optimization*.

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

Garcia, E., Orozco, D., Pavel, R., and Gao, G. R. 2012. A discussion in favor of dynamic scheduling for regular applications in many-core architectures. In *Proceedings of the Workshop on Multithreaded Architectures and Applications*.

Jaleel, A., Najaf-Abadi, H. H., Subramaniam, S., Steely, Jr., S. C., and Emer, J. 2012. CRUS: cache replacement and utility-aware scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

Jiang, Y., Shen, X., Chen, J., and Tripathi, R. 2008. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

Jin, H., Hood, R., Chang, J., Djomehri, J., Jespersen, D., and Taylor, K. 2009. Characterizing application performance sensitivity to resource contention in multicore architectures. Tech. rep., NASA Ames Research Center.

Jin, L. and Cho, S. 2009. SOS: A software-oriented distributed shared cache management approach for chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

Kambadur, M., Moseley, T., Hank, R., and Kim, M. A. 2012. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

Knauerhase, R., Brett, P., Hohlt, B., Li, T., and Hahn, S. 2008. Using OS observations to improve performance in multicore systems. *IEEE Micro 28,* 3.

MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. 2011a. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the International Symposium on Microarchitecture*.

MARS, J., TANG, L., AND SOFFA, M. L. 2011b. Directly characterizing cross core interference through contention synthesis. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*.

MARS, J., VACHHARAJANI, N., HUNDT, R., AND SOFFA, M. L. 2010. Contention aware execution: online contention detection and response. In *Proceedings of the International Symposium on Code Generation and Optimization*.

MILTON, J. S. AND ARNOLD, J. C. 2003. *Introduction to Probability and Statistics* 4th Ed. Tata McGraw Hill Publishing Company, New Delhi, India.

MOSELEY, T., GRUNWALD, D., KIHM, J. L., AND CONNORS, D. A. 2005. Methods for modeling resource contention on simultaneous multithreading processors. In *Proceedings of the International Conference on Computer Design*.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2011. No more backstabbing ... a faithful scheduling policy for multithreaded programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

PUSUKURI, K. K., GUPTA, R., AND BHUYAN, L. N. 2013. ADAPT: A framework for coscheduling multithreaded programs. *ACM Transactions on Architecture and Code Optimization*. 9, 4, Article 45.

QURESHI, M. K. AND PATT, Y. N. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*.

RADOJKOVIĆ, P., ČAKAREVIĆ, V., MORETÓ, M., VERDÚ, J., PAJUELO, A., CAZORLA, F. J., NEMIROVSKY, M., AND VALERO, M. 2012. Optimal task assignment in multithreaded processors: a statistical approach. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

SIDDHA, V. P. S. AND MALLICK, A. 2005. Chip multi processing (CMP) aware Linux kernel scheduler. In *Proceedings of the Ottawa Linux Symposium*.

SNAVELY, A. AND TULLSEN, D. M. 2000. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

SOARES, L., TAM, D., AND STUMM, M. 2008. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proceedings of the International Symposium on Microarchitecture (MICRO'08)*.

TAM, D. K., AZIMI, R., SOARES, L. B., AND STUMM, M. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

TANG, L., MARS, J., VACHHARAJANI, N., HUNDT, R., AND SOFFA, M. L. 2011. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the International Symposium on Computer Architecture*.

WANG, W., DEY, T., MOORE, R., AKTASOGLU, M., CHILDERS, B. R., DAVIDSON, J., IRWIN, M. J., KANDEMIR, M., AND SOFFA, M. L. 2012. REEact: A customizable virtual execution manager for multicore platforms. In *Proceedings of the International Conference on Virtual Execution Environments*.

XIE, Y. AND LOH, G. 2008. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of the CMP-MSI, Held in Conjunction with ISCA-35*.

XIE, Y. AND LOH, G. H. 2009. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA'09)*.

ZHANG, X., DWARKADAS, S., AND SHEN, K. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the European Conference on Computer Systems*.

ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*

Received xxx; revised xxx; accepted xxx Q1

QUERY

Q1: AU: Please provide history date.