Addressing Shared Resource Contention in Multicore Processors via Scheduling

Sergey Zhuravlev Sergey Blagodurov Alexandra Fedorova
School of Computing Science, Simon Fraser University, Vancouver, Canada
{sergey_zhuravlev, sergey_blagodurov, alexandra_fedorova}@sfu.ca

Abstract

Contention for shared resources on multicore processors remains an unsolved problem in existing systems despite significant research efforts dedicated to this problem in the past. Previous solutions focused primarily on hardware techniques and software page coloring to mitigate this problem. Our goal is to investigate how and to what extent contention for shared resource can be mitigated via thread scheduling. Scheduling is an attractive tool, because it does not require extra hardware and is relatively easy to integrate into the system. Our study is the first to provide a comprehensive analysis of contention-mitigating techniques that use only scheduling. The most difficult part of the problem is to find a classification scheme for threads, which would determine how they affect each other when competing for shared resources. We provide a comprehensive analysis of such classification schemes using a newly proposed methodology that enables to evaluate these schemes separately from the scheduling algorithm itself and to compare them to the optimal. As a result of this analysis we discovered a classification scheme that addresses not only contention for cache space, but contention for other shared resources, such as the memory controller, memory bus and prefetching hardware. To show the applicability of our analysis we design a new scheduling algorithm, which we prototype at user level, and demonstrate that it performs within 2% of the optimal. We also conclude that the highest impact of contention-aware scheduling techniques is not in improving performance of a workload as a whole but in improving quality of service or performance isolation for individual applications.

Categories and Subject Descriptors D.4.1 [*Process Manageent*]: Scheduling

General Terms Performance, Measurement, Algorithms

Keywords Multicore processors, shared resource contention, scheduling

1. Introduction

Multicore processors have become so prevalent in both desktops and servers that they may be considered the norm for modern computing systems. The limitations of techniques focused on extraction of instruction-level parallelism (ILP) and the constraints on power budgets have greatly staggered the development of large single cores and made multicore systems a very likely future of computing, with hundreds to thousands of cores per chip. In operating system scheduling algorithms used on multicore systems, the primary strategy for placing threads on cores is load balancing. The OS scheduler tries to balance the runnable threads across the available resources to ensure fair distribution of CPU time and minimize the idling of cores. There is a fundamental flaw with this strategy which arises from the fact that a core is not an independent processor but rather a part of a larger on-chip system and hence shares resources with other cores. It has been documented in previous studies [6, 14, 15, 17, 22, 24] that the execution time of a thread can vary greatly depending on which threads run on the other cores of the same chip. This is especially true if several cores share the same last-level cache (LLC).

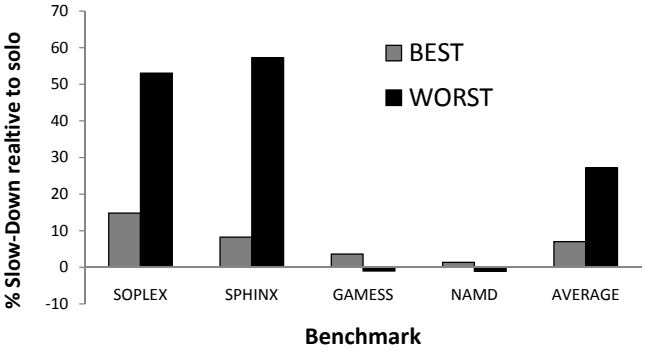


Figure 1. The performance degradation relative to running solo for two different schedules of SPEC CPU2006 applications on an Intel Xeon X3565 quad-core processor (two cores share an LLC).

Figure 1 highlights how the decisions made by the scheduler can affect the performance of an application. This figure shows the results of an experiment where four applications were running simultaneously on a system with four cores and two shared caches. There are three unique ways to distribute the four applications across the four cores, with respect to the pairs of co-runners sharing the cache; this gives us three unique schedules. We ran the threads in each of these schedules, recorded the average completion time for all applications in the workload, and labeled the schedule with the lowest average completion time as the *best* and the one with the highest average completion time as the *worst*. Figure 1 shows the performance degradation that occurs due to sharing an LLC with another application, relative to running solo (contention-free). The best schedule delivers a 20% better average completion time than the worst one. Performance of individual applications improves by as much as 50%.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

Previous work on the topic of improving thread performance in multicore systems focused on the problem of *cache contention* since this was assumed to be the main if not the only cause of performance degradation. In this context cache contention refers to the effect when an application is suffering extra cache misses because its co-runners (threads running on cores that share the same LLC) bring their own data into the LLC evicting the data of others. Methods such as utility cache partitioning (UCP) [17] and page coloring [6, 24, 27] were devised to mitigate cache contention.

Through extensive experimentation on real systems as opposed to simulators we determined that cache contention is not the dominant cause of performance degradation of threads co-scheduled to the same LLC. Along with cache contention other factors like *memory controller contention*, *memory bus contention*, and *prefetching hardware contention* all combine in complex ways to create the performance degradation that threads experience when sharing an LLC.

Our goal is to investigate contention-aware scheduling techniques that are able to mitigate as much as possible the factors that cause performance degradation due to contention for shared resources. Such a scheduler would provide speedier as well as more stable execution times from run to run. Any cache aware scheduler must consist of two parts: a classification scheme for identifying which applications should and should not be scheduled together as well as the scheduling policy which assigns threads to cores given their classification. Since the classification scheme is crucial for an effective algorithm, we focused on the analysis of various classification schemes. We studied the following schemes: Stack Distance Competition (SDC) [5], Animal Classes [26], Solo Miss Rate [12], and the Pain Metric. The best classification scheme was used to design a scheduling algorithm, which was prototyped at user level and tested on two very different systems with a variety of workloads.

Our methodology allowed us to identify the *solo last-level cache miss rate* as one of the most accurate predictors of the degree to which applications will suffer when co-scheduled. We used it to design and implement a new scheduling algorithm called Distributed Intensity (DI). We show experimentally on two different multicore systems that DI always performs better than the default Linux scheduler, delivers much more stable execution times than the default scheduler, and performs within a few percentage points of the theoretical optimal. DI needs only the real miss rates of applications, which can be easily obtained online. As such we developed an online version of DI, DI Online (DIO), which dynamically reads miss counters online and schedules applications in real time.

The key contribution of our work is the *analysis demonstrating the effectiveness of various classification schemes in aiding the scheduler to mitigate shared resource contention*. Previous studies focusing on contention-aware scheduling did not investigate this issue comprehensively. They attempted isolated techniques, in some cases on a limited number of workloads, but did not analyze a variety of techniques and did not quantify how close they are to optimal. Therefore, understanding what is the *best* we can do in terms of contention-aware scheduling remains an open question. Our analysis, in contrast, explores a variety of possible classification schemes for determining to what extent the threads will affect each other's performance, and we believe to cover most of the schemes previously proposed in literature as well as introducing our own. We compare each classification scheme to the theoretical optimal, and this provides a clear understanding of what is the best we can do in a scheduler. Further, we analyze the extent of performance improvements that can be achieved for different workloads by methodically categorizing the workloads based on the potential speedup they can achieve via cache-aware scheduling. This enables us to evaluate the applicability of cache-aware scheduling techniques for a wide variety of workloads. We believe that our work is the first to comprehensively evaluate the potential of scheduling to mitigate contention for shared resources.

The primary application of our analysis is for building new scheduling algorithms that mitigate the effects of shared resource contention. We demonstrate this by designing and evaluating a new algorithm Distributed Intensity Online. Our evaluation of this algorithm leads us to a few interesting and often unexpected findings. First of all, we were surprised to learn that if one is trying to improve average workload performance, the default cache-unaware scheduler already does a rather good job if we measure performance of a workload over a large number of trials. The reason is that for a given workload there typically exists a number of "good" and "bad" scheduling assignments. In some workloads each of these assignments can picked be with a roughly equal probability if selecting uniformly at random, but in other workloads a good assignment is far more likely to occur than the bad one. A cache-unaware default scheduler runs into good and bad assignments according to their respective probabilities, so over time it achieves performance that is not much worse than a contention-aware algorithm that always picks the good assignment. However, when one is interested in improving performance of individual applications, for example to deliver quality of service guarantees or to accomplish performance isolation, a contention-aware scheduler can offer significant improvements over default, because this scheduler can ensure to never select a bad scheduling assignment for the prioritized application.

The rest of the paper is organized as follows: Section 2 describes the classification schemes and policies that we evaluated, the methodology for evaluating the classification schemes separately from the policies and provides the evaluation results for the classification schemes. Section 3 attempts to quantify the effects of different factors resulting from contention for shared on-chip resources on performance on multicore CPUs in order to better explain the results of Section 2. Section 4 describes the scheduling algorithms which were implemented and tested on real systems. Section 5 provides the experimental results. Section 6 discusses the related work and Section 7 offers conclusions and discusses our plans for future work.

2. Classification Schemes

2.1 Methodology

A conventional approach to evaluate new scheduling algorithms is to compare the speedup they deliver relative to a default scheduler. This approach, however, has two potential flaws. First, the schedule chosen by the default scheduler varies greatly based on stochastic events, such as thread spawning order. Second, this approach does not necessarily provide the needed insight into the quality of the algorithms. A scheduling algorithm consists of two components: the information (classification scheme, in our case) used for scheduling decisions and the policy that makes the decisions based on this information. The most challenging part of a cache-aware scheduling algorithm is to select the right classification scheme, because the classification scheme enables the scheduler to predict the performance effects of co-scheduling any group of threads in a shared cache. Our goal was to evaluate the quality of classification schemes separately from any scheduling policies, and only then evaluate the algorithm as a whole. To evaluate classification schemes independently of scheduling policies, we have to use the classification schemes in conjunction with a "perfect" policy. In this way, we are confident that any differences in the performance between the different algorithms are due to classification schemes, and not to the policy.

2.1.1 A "perfect" scheduling policy

As a perfect scheduling policy, we use an algorithm proposed by Jiang et al. [11]. This algorithm is guaranteed to find an optimal scheduling assignment, i.e., the mapping of threads to cores, on a machine with several clusters of cores sharing a cache as long as the *co-run degradations* for applications are known. A co-run degradation is an increase in the execution time of an application when it shares a cache with a co-runner, relative to running solo.

Jiang's methodology uses the co-run degradations to construct a graph theoretic representation of the problem, where threads are represented as nodes connected by edges, and the weights of the edges are given by the sum of the mutual co-run degradations between the two threads. The optimal scheduling assignment can be found by solving a min-weight perfect matching problem. For instance, given the co-run degradations in Table 1, Figure 2 demonstrates how Jiang's method would be used to find the best and the worst scheduling assignment. In Table 1, the value on the intersection of row i and column j indicates the performance degradation that application i experiences when co-scheduled with application j. In Figure 2, edge weights show the sum of mutual co-run degradations of the corresponding nodes. For example, the weight of 90.4% on the edge between MCF and MILC is the sum of 65.63% (the degradation of MCF when co-scheduled with MILC) and 24.75% (the degradation of MILC co-scheduled with MCF).

Table 1. Co-run degradations of four obtained on a real system. Small negative degradations for some benchmarks occur as a result of sharing of certain libraries. The value on the intersection of row i and column j indicates the performance degradation that application i experiences when co-scheduled with application j.

	mcf	milc	gamess	namd
mcf	48.01%	65.63%	2.0%	2.11%
milc	24.75%	45.39%	1.23%	1.11%
gamess	2.67%	4.48%	-1.01%	-1.21%
namd	1.48%	3.45%	-1.19%	-0.93%

Although Jiang's methodology and the corresponding algorithms would be too expensive to use online (the complexity of the algorithm is polynomial in the number of threads on systems with two cores per shared cache and the problem is NP-complete on systems where the degree of sharing is larger), it is acceptable for offline evaluation of the quality of classification schemes.

Using Jiang's algorithm as the perfect policy implies that the classification schemes we are evaluating must be suitable for estimating co-run degradations. All of our chosen classification schemes answered this requirement: they can be used to estimate co-run degradations in absolute or in relative terms.

2.1.2 An optimal classification scheme

To determine the quality of various classification schemes we not only need to compare them with each other, but also to evaluate how they measure up to the optimal classification scheme. All of our evaluated classification schemes attempt to approximate relative performance degradation that arbitrary tuples of threads experience when sharing a cache relative to running solo. An optimal classification scheme would therefore have the knowledge of *actual* such degradation, as measured on a real system. To obtain these measured degradations, we selected ten representative benchmarks from the SPEC CPU2006 benchmark suite (the methodology for selection is described later in this section), ran them solo on our experimental system (described in detail in Section 5), ran all possible pairs of these applications and recorded their performance degradation relative to solo performance. In order to make the analysis tractable it was performed based on pairwise degradations, assuming that only two threads may share a cache, but the resultant scheduling algorithms are evaluated on systems with four cores per shared cache as well.

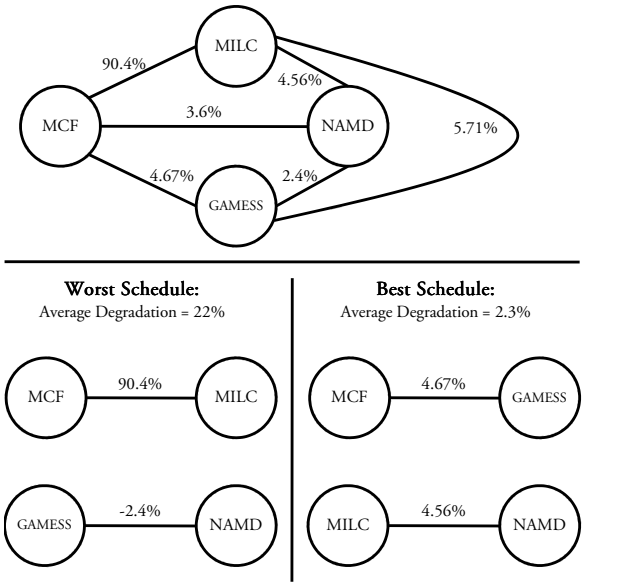


Figure 2. An overview of using Jiang's method for determining the optimal and the worst thread schedule. Edges connecting nodes are labeled with mutual co-run degradations, i.e., the sum of individual degradations for a given pair. The average degradation for a schedule is computed by summing up all mutual degradations and dividing by the total number of applications (four in our case).

2.1.3 Evaluating classification schemes

To evaluate a classification scheme on a particular set of applications, we follow these steps:

1. Find the *optimal schedule* using Jiang's method and the optimal classification scheme, i.e., relying on measured degradations. Record its average performance degradation (see Figure 2).
2. Find the *estimated best schedule* using Jiang's method and the evaluated classification scheme, i.e., relying on estimated degradations. Record its average performance degradation.
3. Compute the difference between the degradation of the optimal schedule and of the estimated best schedule. The smaller the difference, the better the evaluated classification scheme.

To perform a rigorous evaluation, we construct a large number of workloads consisting of four, eight and ten applications. We evaluate all classification schemes using this method, and for each classification scheme report the average degradation above the optimal scheme across all workloads.

2.1.4 Benchmarks and workloads

We selected ten benchmarks from the SPEC2006 benchmark suite to represent a wide range of cache access behaviors. The cache miss rates and access rates for every application in the SPEC2006 benchmark suite were obtained from a third party characterization report [10] and a clustering technique was employed to select the ten representative applications.

From these ten applications we constructed workloads for a four-core, six-core, eight-core, and ten-core processor with two cores per LLC. With the ten benchmarks we selected, there are 210 unique four-application workloads, 210 unique six-application

workloads, 45 unique eight-application workloads, and 1 unique ten-application workload to be constructed on each system. There are three unique ways to schedule a four-application workload on a machine with four cores and two shared caches. The number of unique schedules grows to 15, 105, and 945 for the six, eight, and ten-core systems respectively. Using Jiang's methodology, as opposed to running all these 9450 schedules, saves a considerable amount of time and actually makes it feasible to evaluate such a large number of workloads.

2.2 The Classification Schemes

For any classification scheme to work it must first obtain some "raw" data about the applications it will classify. This raw data may be obtained online via performance counters, embedded into an application's binary as a signature, or furnished by the compiler. Where this data comes from has a lot to do with what kind of data is required by the classification scheme. The SDC algorithm proposed by Chandra et al. [5] is one of the most well known methods for determining how threads will interact with each other when sharing the same cache. The SDC algorithm requires the memory reuse patterns of applications, known as stack distance profiles, as input. Likewise all but one of our classification schemes require stack distance profiles. The one exception is the Miss Rate classification scheme which requires only miss rates as input. The simplicity of the Miss Rate Scheme allowed us to adapt it to gather the miss rates dynamically online making it a far more attractive option than the other classification schemes. However, in order to understand why such a simple classification scheme is so effective as well as to evaluate it against more complex and better established classification schemes we need to explore a wide variety of classification schemes and we need to use stack distance profiles to do so.

A stack distance profile is a compact summary of the application's cache-line reuse patterns. It is obtained by monitoring (or simulating) cache accesses on a system with an LRU cache. A stack distance profile is a histogram with a "bucket" or a position corresponding to each LRU stack position (the total number of positions is equal to the number of cache ways) plus an additional position for recording cache misses. Each position in the stack-distance profile counts the number of hits to the lines in the corresponding LRU stack position. For example, whenever an application reuses a cache line that is at the top of the LRU stack, the number of "hits" in the first position of the stack-distance profile is incremented. If an application experiences a cache miss, the number of items in the miss position is incremented. The shape of the stack-distance profile captures the nature of the application's cache behavior: an application with a large number of hits in top LRU positions has a good locality of reference. An application with a low number of hits in top positions and/or a large number of misses has a poor locality of reference. For our study we obtained the stack-distance profiles using the Pin binary instrumentation tool [16]; an initial profiling run of an application under Pin was required for that. If stack-distance profiles were to be used online in a live scheduler, they could be approximated online using hardware performance counters [24].

We now discuss four classification schemes which are based on the information provided in the stack distance profiles.

2.2.1 SDC

The SDC[1] classification scheme was the first that we evaluated, since this is a well known method for predicting the effects of cache contention among threads [5]. The idea behind the SDC method is to model how two applications compete for the LRU stack positions in the shared cache and estimate the extra misses incurred by each application as a result of this competition. The sum of the extra misses from the co-runners is the proxy for the performance degradation of this co-schedule.

The main idea of the SDC algorithm is in constructing a new stack distance profile that merges individual stack distance profiles of threads that run together. On initialization, each individual profile is assigned a current pointer that is initialized to point to the first stack distance position. Then the algorithm iterates over each position in the profile, determining which of the co-runners will be the "winner" for this stack-distance position. The co-runner with the highest number of hits in the current position is selected as the winner. The winner's counter is copied into the merged profile, and its current pointer is advanced. After the Ath iteration (A is the associativity of the LLC), *the effective cache space for each thread is computed proportionally to the number of its stack distance counters that are included in the merged profile*. Then, the cache miss rate with the new effective cache space is estimated for each co-runner, and the degradation of these miss rates relative to solo miss rates is used as a proxy for the co-run degradation. Note that miss rate degradations do not approximate absolute performance degradations, but they provide an approximation of relative performance in different schedules.

2.2.2 Animal Classes

This classification scheme is based on the animalistic classification of applications introduced by Xie et al. [26]. It allows classifying applications in terms of their influence on each other when co-scheduled in the same shared cache. Each application can belong to one of the four different classes: *turtle* (low use of the shared cache), *sheep* (low miss rate, insensitive to the number of cache ways allocated to it), *rabbit* (low miss rate, sensitive to the number of allocated cache ways) and *devil* (high miss rate, tends to thrash the cache thus hurting co-scheduled applications).

We attempted to use this classification scheme to predict contention among applications of different classes, but found an important shortcoming of the original animalistic model. The authors of the animalistic classification proposed that *devils* (applications with a high miss rate but a low rate of reuse of cached data) must be insensitive to contention for shared resources. On the contrary, we found this not to be the case. According to our experiments, *devils* were some of the most *sensitive* applications – i.e., their performance degraded the most when they shared the on-chip resources with other applications. Since devils have a high miss rate they issue a large number of memory and prefetch requests. Therefore, they compete for shared resources other than cache: memory controller, memory bus, and prefetching hardware. As will be shown in Section 3, contention for these resources dominates performance, and that is why *devils* turn out to be sensitive.

To use the animalistic classification scheme for finding the optimal schedule as well as to account for our findings about "sensitive" *devils* we use a *symbiosis table* to approximate relative performance degradations for applications that fall within different animal classes. The symbiosis table provides estimates of how well various classes co-exist with each other on the same shared cache. For example, the highest estimated degradation (with the experimentally chosen value of 8) will be for two sensitive devils co-scheduled in the same shared cache, because the high miss rate of one of them will hurt the performance of the other one. Two turtles, on the other hand, will not suffer at all. Hence, their mutual degradation is estimated as 0. All other class combinations have their estimates in the interval between 0 and 8.

[1] Chandra suggested three algorithms for calculating the extra miss rates. However, only two of them (FOA and SDC) are computationally fast enough to be used in the robust scheduling algorithm. We chose SDC as it is slightly more efficient than FOA.

The information for classification of applications, as described by Xie et al. [26], is obtained from stack-distance profiles.

2.2.3 Miss Rate

Our findings about "sensitive" devils caused us to consider the miss rate as the heuristic for contention. Although another group of researchers previously proposed a contention-aware scheduling algorithm based on miss rates [12], the hypothesis that the miss rate should explain contention contradicted the models based on stack-distance profiles, which emphasized cache reuse patterns, and thus it needed a thorough validation.

We hypothesized that identifying applications with high miss rates is very beneficial for the scheduler, because these applications exacerbate the performance degradation due to memory controller contention, memory bus contention, and prefetching hardware contention. To attempt an approximation of the "best" schedule using the miss rate heuristic, the scheduler will identify high miss rate applications and separate them into different caches, such that no one cache will have a much higher total miss rate than any other cache. Since no cache will experience a significantly higher miss rate than any other cache the performance degradation factors will be stressed evenly throughout the system.

In addition to evaluating a metric based on miss rates, we also experimented with other metrics, which can be obtained online using hardware counters, such as cache access rate and IPC. Miss rate, however, turned out to perform the best among them.

2.2.4 Pain

The Pain Classification Scheme is based on two new concepts that we introduce in this work: *cache sensitivity* and *cache intensity*. Sensitivity is a measure of how much an application will suffer when cache space is taken away from it due to contention. Intensity is a measure of how much an application will hurt others by taking away their space in a shared cache. By combining the sensitivity and intensity of two applications, we estimate the "pain" of the given co-schedule. Combining a sensitive application with an intensive co-runner should result in a high level of pain, and combining an insensitive application with any type of co-runner should result in a low level of pain. We obtain sensitivity and intensity from stack distance profiles and we then combine them to measure the resulting pain.

To calculate sensitivity S, we examine the number of cache hits that will most likely turn into misses when the cache is shared. To that end, we assign to the positions in the stack-distance profile *loss probabilities* describing the likelihood that the hits will be lost from each position. Intuitively hits to the Most Recently Used (MRU) position are less likely to become misses than hits to the LRU position when the cache is shared. Entries that are accessed less frequently are more likely to be evicted as the other thread brings its data into the cache; thus we scale the number of hits in each position by the corresponding probability and add them up to obtain the likely extra misses. The resulting measure is the sensitivity value which is shown in equation (Eq. 1). Here $h(i)$ is the number of hits to the i-th position in the stack, where $i = 1$ is the MRU and $i = n$ is the LRU for an n-way set associative cache. We use a linear loss probability distribution. As such the probability of a hit in the i-th position becoming a miss is $\frac{i}{n+1}$.

$$S = (\frac{1}{1+n}) \sum_{i=0}^{n} i * h(i) \tag{1}$$

Intensity Z is a measure of how aggressively an application uses cache. As such, it approximates how much space the application will take away from its co-runner(s). Our approach to measuring intensity is to use the number of last-level cache accesses per one million instructions.

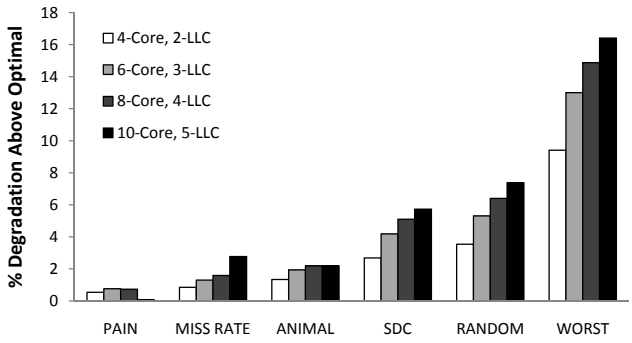


Figure 3. Degradation relative to optimal experienced by each classification scheme on systems with different numbers of cores.

We combine sensitivity S and intensity Z into the Pain metric, which is then used to approximate the co-run degradations required by our evaluation methodology. Suppose we have applications A and B sharing the same cache. Then the Pain of A due to B approximates the relative performance degradation that A is expected to experience due to B and is calculated as the intensity of B multiplied by the sensitivity of A (Eq. 2). The degradation of co-scheduling A and B together is the sum of the Pain of A due to B and the Pain of B due to A (Eq. 3).

$$Pain(A_B) = S(A) * Z(B) \tag{2}$$

$$Pain(A, B) = Pain(A_B) + Pain(B_A) \tag{3}$$

2.2.5 Classification Schemes Evaluation

For the purposes of this work we collected stack distances profiles offline using Intel's binary instrumentation tool Pin [10], an add-on module to Pin MICA [19], and our own module extending the functionality of MICA. The stack distance profiles were converted into the four classification schemes described above: SDC, Pain, Miss rates, and Animal. We estimate the extra degradation above the optimal schedule that each classification scheme produces for the four-core, six-core, eight-core and ten-core systems.Additionally, we present the degradations for the worst and random schedules. A random schedule picks each of the possible assignment for a workload with equal probability.

Figure 3 shows the results of the evaluation. Lower numbers are better. The Pain, Miss Rate and Animal schemes performed relatively well, but SDC surprisingly did only slightly better than random. Pain performed the best, delivering only 1% worse performance than the optimal classification scheme for all the systems. Interestingly we see that all classification schemes except Pain and Animal do worse as the number of cores in the system grows. In systems with more cores the number of possible schedules grows, and so imperfect classification schemes are less able to make a *lucky* choice.

The Animal scheme did worse than Pain. Animal classes are a rather rough estimation of relative co-run degradations (a lot of programs will fall to the same class), and so the Animal scheme simply cannot achieve the same precision as Pain which takes into account absolute values. The Miss Rate scheme performs almost as well as Pain and Animal scheme and yet is by far the easiest to compute either online or offline.

SDC performed worse than Pain and Animal for the following reasons. The first reason is that SDC does not take into account miss rates in its stack distance competition model. So it only works well in those scenarios where the co-running threads have roughly equal miss rates (this observation is made by the authors themselves [5]). When the miss rates of co-running threads are very different, the thread with the higher miss rate will "win" more cache real estate – this fact is not accounted for by the SDC model. Authors of SDC [5] offer a more advanced (but at the same time computationally more expensive) model for predicting extra miss rates which takes into account different miss rates of co-running applications. We did not consider this model in the current work, because we deemed it too computationally expensive to use in an online scheduler.

The second reason has to do with the fact that SDC models the performance effects of cache contention, but as the next section shows, this is not the dominant cause for performance degradation and so other factors must be considered as well. We were initially surprised to find that SDC, a model extensively validated in the past, failed to outperform even such a coarse classification heuristic as the miss rate. In addition to SDC, many other studies of cache contention used stack-distance or reuse-distance profiles for managing contention [5, 17, 22, 24]. The theory behind stack-distance based models seems to suggest that the miss rate should be a *poor* heuristic for predicting contention, since applications with a high cache miss rate may actually have a very poor reuse of their cached data, and so they would be indifferent to contention. Our analysis, however, showed the opposite: miss rate turned out to be an excellent heuristic for contention.

We discovered that the reason for these seemingly unintuitive results had to do with the causes of performance degradation on multicore systems. SDC, and other solutions relying on stack distance profiles such as cache partitioning [17, 22, 26], assumed that the dominant cause of performance degradation is contention for the space in the shared cache, i.e., when co-scheduled threads evict each other data from the shared cache. We found, however, that cache contention is by far not the dominant cause of performance degradation. Other factors, such as contention for memory controllers, memory bus, and resources involved in prefetching, dominate performance degradation for most applications. A high miss rate exacerbates the contention for all of these resources, since a high-miss-rate application will issue a large number of requests to a memory controller and the memory bus, and will also be typically characterized by a large number of prefetch requests.

In the next section we attempt to quantify the causes for performance degradation resulting from multiple factors, showing that contention for cache space is not dominant; these results provide the explanation why a simple heuristic such as the miss rate turns out to be such a good predictor for contention.

3. Factors Causing Performance Degradation

Recent work on the topic of performance degradation in multicore systems focused on contention for cache space and the resulting data evictions when applications share the LLC. However, it is well known that cache contention is far from being the only factor that contributes to performance degradation when threads share an LLC. Sharing of other resources, such as the memory bus, memory controllers and prefetching hardware also plays an important role. Through extensive analysis of data collected on real hardware we have determined that contention for space in the shared cache explains only a part of the performance degradation when applications share an LLC. In this section we attempt to quantify how much performance degradation can be attributed to contention for each shared resource.

Estimating the contribution that each factor has on the overall performance degradation is difficult, since all the degradation factors work in conjunction with each other in complicated and practically inseparable ways. Nevertheless, we desired a rough estimate of the degree to which each factor affects overall performance degradation to identify if any factor in particular should be the focus of our attention since mitigating it will yield the greatest improvements.

We now describe the process we used to estimate the contributions of each factor to the overall degradation. Our experimental system is a two-sockets server with two Intel X5365 "Clovertown" quad-core processors. The two sockets share the memory controller hub, which includes the *DRAM controller*. On each socket there are four cores sharing a *front-side bus (FSB)*. There are two L2 caches on each socket, one per pair of cores. Each pair of cores also shares prefetching hardware, as described below. So when two threads run on different sockets, they compete for the DRAM controller. When they run on the same socket, but on different caches, they compete for the FSB, in addition to the DRAM controller. Finally, when they run on cores sharing the same cache, they also compete for the L2 cache and the prefetching hardware, in addition to the FSB and the DRAM controller. To estimate how contention for each of these resources contributes to the total degradation, we measured the execution times of several benchmarks under the following eight conditions:

Solo_PF_ON: Running SOLO and prefetching is ENABLED
Solo_PF_OFF: Running SOLO and prefetching is DISABLED
SameCache_PF_ON: Sharing the LLC with an interfering benchmark and prefetching is ENABLED
SameCache_PF_OFF: Sharing the LLC with an interfering benchmark and prefetching is DISABLED
DiffCache_PF_ON: An interfering benchmark runs on a different LLC but on the same socket and prefetching is ENABLED
DiffCache_PF_OFF: An interfering benchmark runs on a different LLC but on the same socket and prefetching is DISABLED
DiffSocket_PF_ON: An interfering benchmark runs on a different socket and prefetching is ENABLED
DiffSocket_PF_OFF: An interfering benchmark runs on a different socket and prefetching is DISABLED

As an interfering benchmark for this experiment we used MILC. MILC was chosen for several reasons. First, it has a very high solo miss rate which allows us to estimate one of the worst-case contention scenarios. Second, MILC suffers a negligible increase in its own miss rate due to cache contention (we determined this via experiments and also by tracing MILC's memory reuse patterns, which showed that MILC hardly ever reuses its cached data) and hence will not introduce extra misses of its own when co-run with other applications. We refer to MILC as the *interfering benchmark* and we refer to the test application simply as the *application*. In the experiment where MILC is the test application, SPHINX is used as the interfering benchmark.

Estimating Performance Degradation due to DRAM Controller Contention We look at the difference between the solo run and the run when the interfering benchmark is on a different socket. When the interfering benchmark is on a different socket any performance degradation it causes can only be due to DRAM controller contention since no other resources are shared. Equation 4 shows how we estimate the performance degradation due to DRAM controller contention.

$$DRAM\_contention = \frac{DiffSocket\_PF\_OFF - Solo\_PF\_OFF}{Solo\_PF\_OFF} \quad (4)$$

There are several complications with this approach, which make it a rough estimate as opposed to an accurate measure of DRAM controller contention. First, when the LLC is shared by two applications, extra evictions from cache cause the total number of misses to go up. These extra misses contribute to the DRAM controller contention. In our experimental technique the two applications are in different LLCs and hence there are no extra misses. As a result, we are underestimating the DRAM controller contention. Second, we chose to disable prefetching for this experiment. If we enabled prefetching and put two applications into different LLC then they would each have access to a complete set of prefetching hardware. This would have greatly increased the total number of requests issued to the memory system from the prefetching hardware as compared to the number of requests that can be issued from only one LLC. By disabling the prefetching we are once again underestimating the DRAM controller contention. As such the values that we measure should be considered a lower bound on DRAM controller contention.

Estimating Performance Degradation due to FSB Contention Next, we estimate the degree of performance degradation due to contention for the FSB. To that end, we run the application and the interfering benchmark on the same socket, but on different LLCs. This is done with prefetching disabled, so as not to increase the bus traffic. Equation 5 shows how we estimate the degradation due to FSB contention.

$$FSB\_Contention = \frac{DiffCache\_PF\_OFF - DiffSocket\_PF\_OFF}{Solo\_PF\_OFF} \quad (5)$$

Estimating Performance Degradation due to Cache Contention To estimate the performance degradation due to cache contention we take the execution time when an application is run with an interfering co-runner in the same LLC and subtract from it the execution time of the application running with the interfering benchmark in a different LLC of the same socket. This is done with prefetching disabled so as not to increase bus traffic or contend for prefetching hardware. The difference in the execution times between the two runs can be attributed to the extra misses that resulted due to cache contention. Equation 6 demonstrates how we estimate performance degradation due to cache contention.

$$Cache\_Contention = \frac{SameCache\_PF\_OFF - DiffCache\_PF\_OFF}{Solo\_PF\_OFF} \quad (6)$$

Estimating Performance Degradation due to Contention for Resources Involved in Prefetching Contention for resources involved in prefetching has received less attention in literature than contention for other resources. We were compelled to investigate this type of contention when we observed that some applications experienced a decreased prefetching rate (up to 30%) when sharing an LLC with a memory-intensive co-runner. Broadly speaking, prefetching resources include all the hardware that might contribute to the speed and quality of prefetching. For example, our experimental processor has two types of hardware that prefetches into the L2 cache. The first is the Data Prefetching Logic (DPL) which is activated when an application has two consecutive misses in the LLC and a stride pattern is detected. In this case, the rest of the addresses up to the page boundary are prefetched. The second is the adjacent cache line prefetcher, or the streaming prefetcher. The L2 prefetching hardware is dynamically shared by the two cores using the LLC. The memory controller and the FSB are also involved in prefetching, since they determine how aggressively these requests can be issued to memory. It is difficult to tease apart the latencies attributable to contention for each resource, so our estimation of contention for prefetching resources includes contention for prefetching hardware as well as additional contention for these two other resources. This is an upper bound on the contention for the prefetching hardware itself.

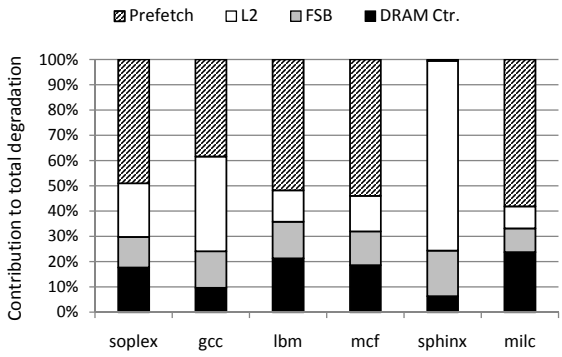


Figure 4. Percent contribution that each of the factors have on the total degradation.

We can measure the performance degradation due to prefetching-related resources as the difference between the total degradation and the degradation caused by cache contention, FSB, and DRAM controller contention. Equation 7 calculates the total degradation of an application when the LLC is shared by looking at the difference when the interfering benchmark shares the LLC and when the application runs alone. Equation 8 shows the calculation of the prefetching degradation.

$$Total\_Degradation = \frac{SameCache\_PF\_ON - Solo\_PF\_ON}{Solo\_PF\_ON} \quad (7)$$

$$Prefetching\_Contention = Eq.(7) - Eq.(6) - Eq.(5) - Eq.(4) \quad (8)$$

Finally, we calculate the degradation contribution of each factor as the ratio of its degradation compared to the total degradation. Figure 4 shows the percent contribution of each factor (DRAM controller contention, FSB contention, L2 cache contention, and prefetching resource contention) to the total degradation for six SPEC2006 benchmarks.

The six applications shown in Figure 4 are the applications that experience a performance degradation of at least 45% chosen from the ten representative benchmarks. We see from Figure 4 that for all applications except SPHINX contention for resources *other than shared cache* is the dominant factor in performance degradation, accounting for more than 50% of the total degradation.

While cache contention does have an effect on performance degradation, any strategy that caters to reducing cache contention exclusively cannot and will not have a major impact on performance. The fact that contention for resources other than cache is dominant, explains why the miss rate turns out to be such a good heuristic for predicting contention. The miss rate highly correlates with the amount of DRAM controller, FSB, and prefetch requests, and thus is indicative of both the sensitivity of an application as well as its intensity.

4. Scheduling Algorithms

A scheduling algorithm is the combination of a classification scheme and a scheduling policy. We considered and evaluated several scheduling algorithms that combined different classification schemes and policies, and in this section we present those that

showed the best performance and were also the simplest to implement. In particular, we evaluate two algorithms based on the Miss Rate classification schemes, because the miss rate is very easy to obtain online via hardware performance counters. The scheduling policy we used was the *Centralized Sort*. It examines the list of applications, sorted by their miss rates, and distributes them across cores, such that the total miss rate of all threads sharing a cache is equalized across all caches. For the evaluation results of other heuristics and policies, we refer the reader to our technical report [3].

While the Pain classification scheme gave the best performance, we chose not to use it in an online algorithm, instead opting to implement one using the miss rate heuristic. This was done to make the scheduler simpler, thus making more likely that it will be adopted in general-purpose operating systems. Using Pain would require more changes to the operating system than using the miss rate for the following reason. Pain requires stack distance profiles. Obtaining a stack distance profile online requires periodic sampling of data addresses associated with last-level cache accesses using advanced capabilities of hardware performance monitoring counters, as in RapidMRC [24]. Although RapidMRC can generate accurate stack-distance profiles online with low overhead, there is certain complexity associated with its implementation. If a scheduler uses the miss rate heuristic, all it has to do is periodically measure the miss rates of the running threads, which is simpler than collecting stack-distance profiles. Given that the miss rate heuristic had a much lower implementation complexity but almost the same performance as Pain, we thought it would be the preferred choice in future OS schedulers.

In the rest of this section we describe two algorithms based on the Miss Rate classification scheme: *Distributed Intensity* (DI) and *Distributed Intensity Online* (DIO). DIO does not rely on stack-distance profiles, but on the miss rates measured online. DI estimates the miss rate based on the stack-distance profiles; it was evaluated in order to determine if any accuracy is lost when the miss rates are measured online as opposed to estimated from the profiles.

4.1 Distributed Intensity (DI)

In the Distributed Intensity (DI) algorithm all threads are assigned a value which is their solo miss rate (misses per one million instructions) as determined from the stack distance profile. The goal is then to distribute the threads across caches such that the miss rates are distributed as evenly as possible. The idea is that the performance degradation factors identified in Section 3 are all exacerbated by a high miss rate and so we avoid the situation where any cache has a much higher cumulative miss rate than any other cache such that performance bottlenecks are not created in that cache.

To further justify the validity of this method we performed a study on all possible 4-thread workloads that can be constructed from the 10 representative SPEC2006 benchmarks. We computed the percent difference between average co-run degradations for each workload achieved with the optimal solution relative to the worst solution (we refer to this value as the speedup). The highest average speedup relative to the worst schedule was 25%, and the lowest was less than 1%. We broke up the workloads based on the speedup range into which they fit: (25%-15%), (15%-10%), (10%-5%), and (5%-0%), and studied which types of applications are present in each range. For simplicity we define two categories of applications: intensive (above average miss rate), and non-intensive (below average miss rate).

We note that according to the ideas of DI, workloads consisting of two intensive and two non-intensive applications should achieve the highest speedups. This is because in the worst case these workloads can be scheduled in such a way as to put both intensive applications in the same cache, creating a cache with a large miss rate, and a performance bottleneck. Alternatively, in the best case these workloads can be scheduled to spread the two intensive applications across caches thus minimizing the miss rate from each cache and avoiding bottlenecks. The difference in performance between these two cases (the one with a bottleneck and the one without) should account for the significant speedup of the workload. We further note that workloads consisting of more than two or fewer than two intensive applications can also benefit from distribution of miss rates but the speedup will be smaller, since the various scheduling solutions do not offer such a stark contrast between creating a major bottleneck and almost entirely eliminating it.

Figure 5 shows the makeup of workloads (intensive vs. non-intensive) applications and the range of speedups they offer. The (unlabeled) x-axis identifies all the workloads falling into the given speedup range. We see that the distribution of applications validates the claims of DI. The other claim that we make to justify why DI should work is that miss rates of applications are relatively stable. What we mean by stability is that when an application shares the LLC with a co-runner its miss rate will not increase so dramatically as to make the solution found by DI invalid.

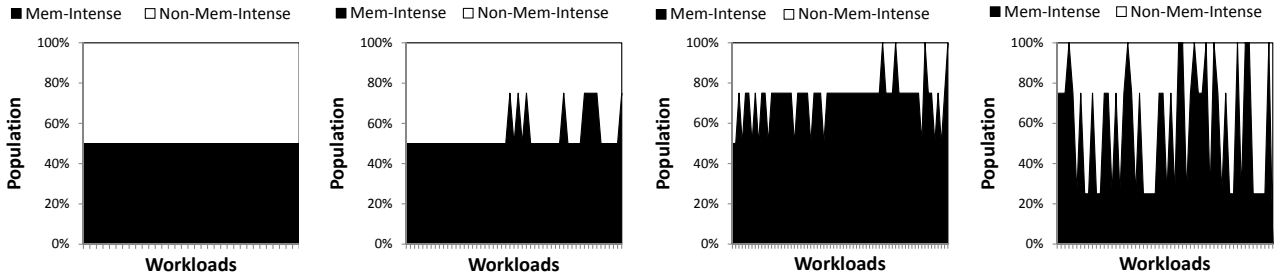
DI assigns threads to caches to even out the miss rate across all the caches. This assignment is done based on the solo miss rates of applications. The real miss rate of applications will change when they share a cache with a co-runner, but we claim that these changes will be relatively small such that the miss rates are still rather even across caches. Consider Figure 6 which shows the solo miss rates of the 10 SPEC2006 benchmarks as well as the largest miss rate observed for each application as it was co-scheduled to share a cache with all other applications in the set.

We see that if the applications were sorted based on their miss rates their order would be nearly identical if we used solo miss rates, maximum miss rates, or anything in between. Only the applications MILC and SOPLEX may exchange positions with each other or GCC and SPHINX may exchange positions depending on the miss rate used. The DI algorithm makes scheduling decisions based on the sorted order of applications. If the order of the sorted applications remains nearly unchanged as the miss rates changes then the solutions found by DI would also be very similar. Hence the solution found by DI with solo miss rates should also be very good if the miss rates change slightly. Through an extensive search of all the SPEC2006 benchmark suite and the PARSEC benchmark suite we have not found any applications whose miss rate change due to LLC contention would violate the claim made above.

The DI scheduler is implemented as a user level scheduler running on top of Linux. It enforces all scheduling decisions via system calls which allow it to bind threads to cores. The scheduler also has access to files containing the solo miss rates. For all the applications it uses solo miss rate estimated using stack distance profiles as the input to the classification scheme.

4.2 Distributed Intensity Online (DIO)

DIO is based on the same classification scheme and scheduling policies as DI except that it obtains the miss rates of applications dynamically online via performance counters. This makes DIO more attractive since the stack distance profiles, which require extra work to obtain online, are not required. The miss rate of applications can be obtained dynamically online on almost any machine with minimal effort. Furthermore, the dynamic nature of the obtained miss rates makes DIO more resilient to applications that have a change in the miss rate due to LLC contention. DIO continuously monitors the miss rates of applications and thus accounts for phase changes. To minimize migrations due to phase changes of applications we collect miss rate data not more frequently than once every



(a) The makeup of workloads which deliver speedups from 25%-15%.

(b) The makeup of workloads which deliver speedups from 15%-10%.

(c) The makeup of workloads which deliver speedups from 10%-5%.

(d) The makeup of workloads which deliver speedups from 5%-0%.

Figure 5. The makeup of workloads

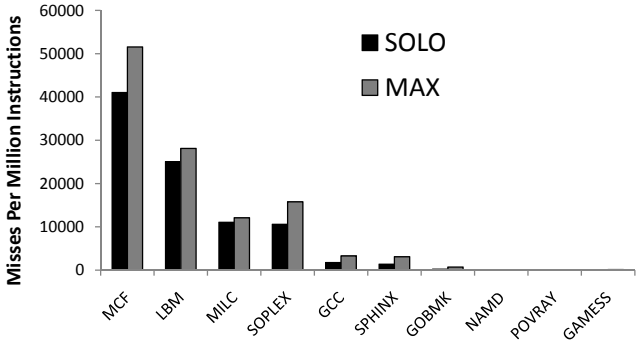


Figure 6. The solo and maximum miss rate recorded for each of the 10 SPEC2006 benchmarks.

billion cycles and we use a running average for scheduling decisions.

The DIO scheduler, like DI, manages the assignment of threads to cores using affinity interfaces provided in Linux. As such, it mirrors the actions that would be performed by a kernel scheduler. The key difference is that the kernel scheduler would directly manipulate the runqueues in order to place a thread on a particular core, but a user-level prototype of the scheduler uses affinity-related system calls for that purpose. For example, to swap thread A on core i with thread B on core j we set affinity of A to j and affinity B to i. Linux does the actual swapping. In the future, when DIO will become part of the kernel swapping will be done by manipulating the run-queues.

5. Evaluation on Real Systems

5.1 Evaluation Platform

We performed the experiments on two systems:

Dell-Poweredge-2950 (Intel Xeon X5365) has eight cores placed on four chips. Each chip has a 4MB 16-way L2 cache shared by its two cores. Each core also has private L1 instruction and data caches. In our first series of experiments we used only two chips out of four. This enabled us to verify our analytical results for the 4 thread workloads directly. After that, all eight cores with eight-thread workloads were used.

Dell-Poweredge-R805 (AMD Opteron 2350 Barcelona) has eight cores placed on two chips. Each chip has a 2MB 32-way L3 cache shared by its four cores. Each core also has a private unified L2 cache and private L1 instruction and data caches. All eight cores with eight thread workloads were used.

The experimental workloads were comprised of the 14 benchmarks from SPEC CPU 2006 suite chosen using the clustering technique as described in Section 2. See Table 2. For the eight-core experiments we created eight-thread workloads by doubling the corresponding four-thread workloads. For example, for the four-thread workload (SOPLEX, SPHINX, GAMESS, NAMD) the corresponding eight-thread workload is (SOPLEX, SPHINX, GAMESS, NAMD, SOPLEX, SPHINX, GAMESS, NAMD). The user-level scheduler starts the applications and binds them to cores as directed by the scheduling algorithm.

Since we are focused on CPU-bound workloads, which are *not* likely to run with more threads than cores [1, 25], we only evaluate the scenarios where the number of threads does not exceed the number of cores. If the opposite were the case, the scheduler would simply re-evaluate the mapping of threads to cores every time the set of running threads changes. The decision of which thread is selected to run would be made in the same way as it is done by the default scheduler. While in this case there are also opportunities to separate competing threads *in time* as opposed to in space, we do not investigate these strategies in this work.

Both systems were running Linux Gentoo 2.6.27 release 8. We compare performance under DI and DIO to the default contention-unaware scheduler in Linux, referring to the latter as DEFAULT. The prefetching hardware is fully enabled during these experiments. To account for the varied execution times of benchmark we restart an application as soon as it terminates (to ensure that the same workload is running at all times). An experiment terminates when the longest application executed three times.

	Workloads			
	2 memory-bound		**2 CPU-bound**	
1	SOPLEX	SPHINX	GAMESS	NAMD
2	SOPLEX	MCF	GAMESS	GOBMK
3	MCF	LIBQUANTUM	POVRAY	GAMESS
4	MCF	OMNETPP	H264	NAMD
5	MILC	LIBQUANTUM	POVRAY	PERL
	1 memory-bound		**3 CPU-bound**	
6	SPHINX	GCC	NAMD	GAMESS
	3 memory-bound		**1 CPU-bound**	
7	LBM	MILC	SPHINX	GOBMK
8	LBM	MILC	MCF	NAMD

Table 2. The workloads used for experiments.

5.2 Results

Intel Xeon 4 cores We begin with the results for the four-thread workloads on the four-core configuration of the Intel Xeon ma-

chine. For every workload we first run the three possible unique schedules and measure the aggregate workload completion time of each. We then determine the schedule with the optimal (minimal) completion time, the worst possible schedule (maximum completion time) and the expected completion time of the random scheduling algorithm (it selects all schedules with equal probability). We then compared the aggregate execution times of DI and DIO with the completion times of OPTIMAL, WORST and RANDOM. We do not present results for the default Linux scheduler because when the scheduler is given a processor affinity mask to use only four cores out of eight, the migrations of threads across cores become more frequent than when no mask is used, leading to results with an atypically high variance. Figure 7 shows the performance degradation above the optimal for every workload with DI, DIO, RANDOM and WORST. The results show that DI and DIO perform better than RANDOM and are within 2% of OPTIMAL.

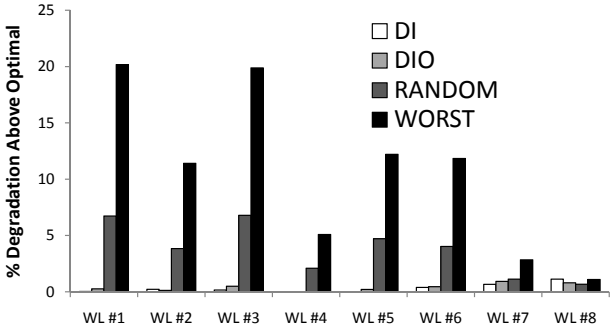


Figure 7. Aggregate performance degradation of each workload with DI, DIO, RANDOM and WORST relative to OPTIMAL (low bars are good) for the Intel machine and 4 threads.

Intel Xeon 8 cores Since this setup does not require a processor affinity mask, we evaluated the results of DI and DIO against DEFAULT as in this case DEFAULT does not experience excessive migrations. Figure 8 shows the percent aggregate workload speedup over DEFAULT.

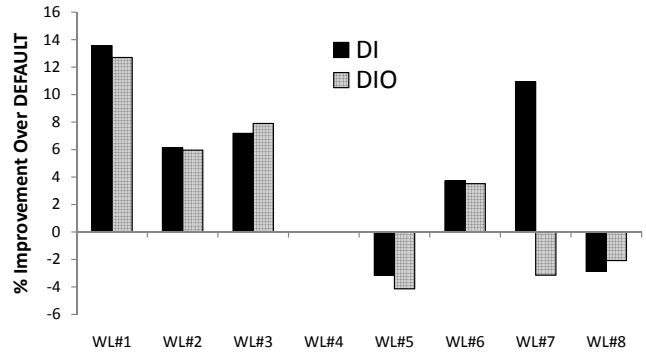


Figure 8. Aggregate performance improvement of each workload with DI and DIO relative to DEFAULT (high bars are good) for Intel 8 threads.

We note that although generally DI and DIO improve aggregate performance over DEFAULT, in a few cases they performed slightly worse. However, the biggest advantage of DI and DIO is that they offer much more stable results from run to run and avoid the worst-case thread assignment. This effect is especially significant if we look at performance of individual applications. Figure 10(a) shows relative performance improvement for individual applications of the worst-case assignments of DI and DIO over the worst case assignments under DEFAULT. The results show that DEFAULT consistently stumbles on much worse solutions than DIO or DIO and as such there are cases when the performance of individual applications is unpredictably bad under DEFAULT. What this means is that if an application were repeatedly executed on a multicore system, running it under DEFAULT vs., for instance, DIO may occasionally cause its performance to degrade by as much as 100% in some cases! Figure 10(b) shows the deviation of the execution time of consecutive runs of the same application in the same workload with DI, DIO and DEFAULT. We note that DEFAULT has a much higher deviation from run to run than DI and DIO. DIO has a slightly higher deviation than DI as it is sensitive to phase changes of applications and as a result tends to migrate applications more frequently.

AMD Opteron 8 cores Finally, we report the results for the same eight-thread workloads on the AMD system. The results for the percent aggregate workload speedup over DEFAULT (Figure 9), relative performance improvement of the worst case assignments over DEFAULT (Figure 10(c)) and the deviation of the execution times (Figure 10(d)) generally repeat the patterns observed on the Intel Xeon machine with eight threads.

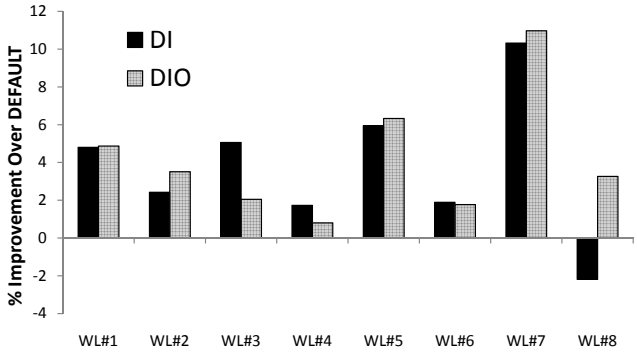
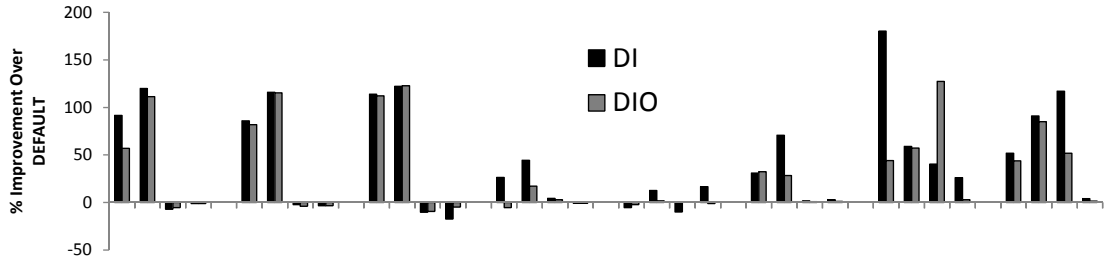


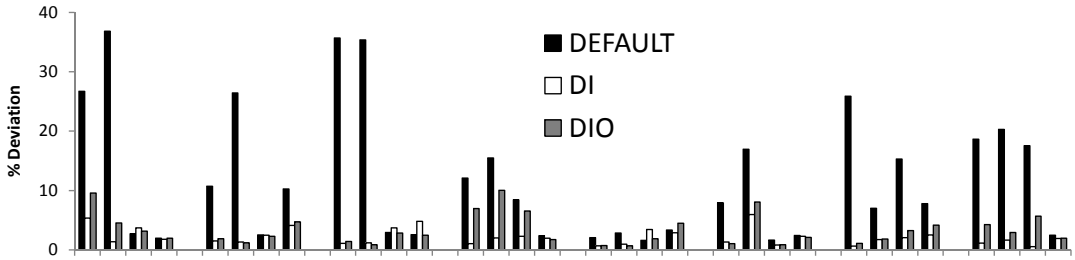
Figure 9. Aggregate performance improvement of each workload with DI and DIO relative to DEFAULT (high bars are good) for AMD 8 threads.

5.3 Discussion

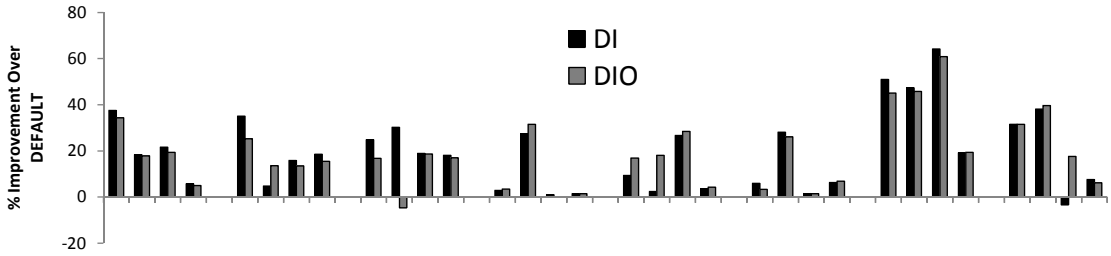
We draw several conclusions from our results. First of all, the classification scheme based on miss rates effectively enables to reduce contention for shared resources using a purely scheduling approach. Furthermore, an algorithm based on this classification scheme can be effectively implemented online as demonstrated by our DIO prototype. Using contention-aware scheduling can help improve overall system efficiency by reducing completion time for the entire workload as well as reduce worst-case performance for individual applications. In the former case, DIO improves performance by up to 13% relative to DEFAULT and in the isolated cases where it does worse than DEFAULT, the impact on performance is at most 4%, far smaller than the corresponding benefit. On average, if we examine performance across all the workloads we have tried DEFAULT does rather well in terms of workload-wide performance – in the worst case it does only 13% worse than DIO. But if we consider the variance of completion times and the effect on individual applications, the picture changes significantly. DEFAULT achieves a much higher variance and it is likely to stumble into much worse worst-case performance for individual applications. This means that when the goal is to deliver QoS, achieve performance isolation or simply prioritize individual applications,



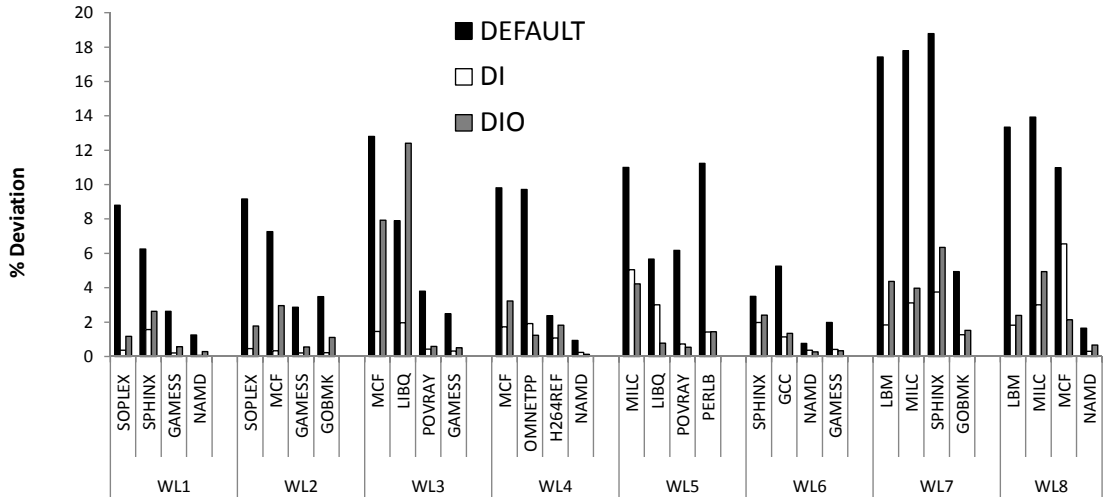
(a) The relative performance improvement of the worst case DI and DIO over the worst case DEFAULT for Intel 8 threads.



(b) Deviation of the same application in the same workload with DI, DIO and Default (low bars are good) for Intel 8 threads.



(c) The relative performance improvement of the worst case DI and DIO over the worst case DEFAULT for AMD 8 threads.



(d) Deviation of the same application in the same workload with DI, DIO and Default (low bars are good) for AMD 8 threads.

Figure 10. The relative performance improvement and deviation

contention-aware scheduling can achieve much larger performance impacts, speeding up individual applications by as much as a factor of two.

To understand why DEFAULT performs relatively well on average let us discuss several examples. Consider a four core machine where each pair of cores shares a cache. If the workload to be executed on this machine involves two intensive applications and two

non-intensive applications and if the threads are mapped randomly to cores (which is a good approximation for DEFAULT) then there is *only* a $1/3$ probability of stumbling onto the *worst* solution where the intensive applications share the same cache. If there are three intensive applications in the workload and only one non-intensive application then all mappings are relatively equivalent *on average* since two of the intensive applications will experience performance degradation and one will not (the one paired with a non-intensive one). Similarly, workloads with no intensive applications, one intensive applications, and all intensive applications show no real difference between solutions. As such DEFAULT is able to perform well on average. Therefore, we believe that future research must focus on the performance of individual threads which can vary greatly under the DEFAULT scheduler as opposed to trying to improve average performance.

6. Related Work

In the work on Utility Cache Partitioning [17], a custom hardware solution estimates each application's number of hits and misses for all possible number of ways allocated to the application in the cache (the technique is based on stack-distance profiles). The cache is then partitioned so as to minimize the number of cache misses for the co-running applications. UCP minimizes cache contention given a particular set of co-runners. Our solution, on the other hand, decides which co-runners to co-schedule so as to minimize contention. As such, our solution can be complementary to UCP and other solutions relying on cache partitioning [22].

Tam, Azimi, Soares and Stumm [24] similarly to several other researchers [6, 14, 15, 27] address cache contention via software-based cache partitioning. The cache is partitioned among applications using page coloring. Each application is reserved a portion of the cache, and the physical memory is allocated such that the application's cache lines map only into that reserved portion. The size of the allocated cache portion is determined based on the marginal utility of allocating additional cache lines for that application. Marginal utility is estimated via an application's reuse distance profile, which is very similar to a stack-distance profile and is approximated online using hardware counters [24]. Software cache partitioning, like hardware cache partitioning, is used to isolate workloads that hurt each other. While this solution delivers promising results, it has two important limitations: first of all, it requires non-trivial changes to the virtual memory system, a very complicated component of the OS. Second, it may require copying of physical memory if the application's cache portion must be reduced or reallocated. Finally, it does not address contention for resources other than shared cache, which, as we have shown, are the dominant cause of performance degradation. Given these limitations, it is desirable to explore options like scheduling, which are not subject to these drawbacks.

Herdrich et al. suggested rate-based QoS techniques, which involved throttling the speed of the core in order to limit the effects of memory-intensive applications on its co-runners [9]. Rate-based QoS can be used in conjunction with scheduling to provide isolation from contention for high-priority applications.

Several prior studies investigated the design of cache-aware scheduling algorithms. Symbiotic Jobscheduling [21] is a method for co-scheduling threads on simultaneous multithreading processors (SMT) in a way that minimizes resource contention. This method could be adapted to co-schedule threads on single-threaded cores sharing caches. This method works by trying (or sampling) a large number of thread assignments and picking the ones with the best observed rate of instructions per cycle.

The drawback of this solution is that it requires a sampling phase during which the workload is not scheduled optimally. When the number of threads and cores is large, the number of samples that must be taken will be large as well.

Fedorova et. al designed a cache-aware scheduler that compensates threads that were hurt by cache contention by giving them extra CPU time [8]. This algorithm accomplishes the effect of fair cache sharing, but it was not designed to improve overall performance.

The idea of distributing benchmarks with high miss rates across the caches was also suggested in [7, 12]. The authors propose to reduce cache interference by spreading the intensive applications apart and co-scheduling them with non-intensive applications. Cache misses per cycle were used as the metric for intensity. Our idea of DI is similar, however we provide a more rigorous analysis of this idea and the reasons for its effectiveness and also demonstrate that DI operates within a narrow margin of the optimal. The other paper did not provide the same analysis, so it was difficult to judge the quality of their algorithm comprehensively. Furthermore, our work is broader in a sense that we propose and analyze several new algorithms and classification schemes.

7. Conclusions and Future Work

In this work we identified factors other than cache space contention which cause performance degradation in multicore systems when threads share the same LLC. We estimated that other factors like memory controller contention, memory bus contention and prefetching hardware contention contribute more to overall performance degradation than cache space contention. We predicted that in order to alleviate these factors it was necessary to minimize the total number of misses issued from each cache. To that end we developed scheduling algorithms DI and DIO that distribute threads such that the miss rate is evenly distributed among the caches.

The Miss Rate heuristic, which underlies the DI and DIO algorithms was evaluated against the best known strategies for alleviating performance degradation due to cache sharing, such as SDC, and it was found to perform near the theoretical optimum. DIO is a user level implementation of the algorithm relying on the Miss Rate heuristic that gathers all the needed information online from performance counters. DIO is simple, can work both at the user and kernel level, and it requires no modifications to hardware or the non-scheduler parts of the operating system. DIO has been shown to perform within 2% of the oracle optimal solution on two different machines. It performs better than the default Linux scheduler both in terms of average performance (for the vast majority of workloads) as well as in terms of execution time stability from run to run for individual applications.

In the future we hope to make DIO more scalable by implementing a purely distributed version of this algorithm, which does not rely on centralized sort. We also hope to embed DIO directly into the kernel. Another major focus of our research is exploring other factors that affect performance degradation both negatively and positively (e.g. inter-thread data sharing) and ways to exploit these factors in scheduling solutions.

As mentioned in the discussion section, one of the clear advantages of a contention aware scheduler over default is that it offers stable execution times from run to run, which can ensure better QoS for applications. We are working on ways to expand the DI contention aware scheduler to allow the user to set priorities for certain applications which would then be given a better contention isolation than other non-prioritized applications.

8. Acknowledgments

The authors thank Sun Microsystems, National Science and Engineering Research Council of Canada, and in part the Strategic Project Grants program, and Google for supporting this research.

References

[1] D. an Mey, S. Sarholz, and C. Terboven et al. The RWTH Aachen SMP-Cluster User's Guide, Version 6.2. 2007.

[2] E. Berg and E. Hagersten. Statcache: a Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the IEEE International Symmposium on Performance Analysis of Systems and Software*, pages 20–27, 2004.

[3] S. Blagodurov, S. Zhuravlev, S. Lansiquot, and A. Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *Simon Fraser University, Technical Report 2009-16*, 2009.

[4] C. Cascaval, L. D. Rose, D. A. Padua, and D. A. Reed. Compile-Time Based Performance Prediction. In *LCPC '99: Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, pages 365–379, 2000.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.

[6] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[7] G. Dhiman, G. Marchetti, and T. Rosing. vGreen: a System for Energy Efficient Computing in Virtualized Environments. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2009.

[8] A. Fedorova, M. I. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT'07)*, pages 25–38, 2007.

[9] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS Techniques for Cache/Memory in CMP Platforms. In *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, pages 479–488, 2009.

[10] K. Hoste and L. Eeckhout. Microarchitecture-Independent Workload Characterization. *IEEE Micro*, 27(3):63–72, 2007.

[11] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pages 220–229, 2008.

[12] 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.

[13] E. Koukis and N. Koziris. Memory Bandwidth Aware Scheduling for SMP Cluster Nodes. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 187–196, 2005.

[14] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled Cache Predictability for Real-Time Systems. In *RTAS '97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, page 213, 1997.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of International Symposium on High Performance Computer Architecture (HPCA 2008)*, pages 367–378, 2008.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[17] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[18] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 245–258, 2007.

[19] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. *WIOSCA*, 2008.

[20] D. Shelepov, J. C. Saez, and S. Jeffery et al. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM Operating System Review*, 43(2), 2009.

[21] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28(5):234–244, 2000.

[22] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, 2002.

[23] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Acheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys '07)*, 2007.

[24] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 121–132, 2009.

[25] R. van der Pas. The OMPlab on Sun Systems. In *Proceedings of the First International Workshop on OpenMP*, 2005.

[26] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proc. of CMP-MSI, held in conjunction with ISCA-35*, 2008.

[27] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys'09)*, pages 89–102, 2009.