

Contention for caches, memory controllers, and interconnects can be eased by contention-aware scheduling algorithms.

**BY ALEXANDRA FEDOROVA, SERGEY BLAGODUROV,
AND SERGEY ZHURAVLEV**

Managing Contention for Shared Resources on Multicore Processors

MODERN MULTICORE SYSTEMS are designed to allow clusters of cores to share various hardware structures, such as LLCs (last-level caches; for example, L2 or L3), memory controllers, and interconnects, as well as prefetching hardware. We refer to these resource-sharing clusters as memory domains because the

shared resources mostly have to do with the memory hierarchy. Figure 1 provides an illustration of a system with two memory domains and two cores per domain.

Threads running on cores in the same memory domain may compete for the shared resources, and this contention can significantly degrade

their performance relative to what they could achieve running in a contention-free environment. Consider an example demonstrating how contention for shared resources can affect application performance. In this example, four applications—Soplex, Sphinx, Gamess, and Namd, from the Standard Performance Evaluation Corporation

(SPEC) CPU 2006 benchmark suite⁶—run simultaneously on an Intel Quad-Core Xeon system similar to the one depicted in Figure 1.

As a test, we ran this group of applications several times, in three different schedules, each time with two different pairings sharing a memory domain. The three pairing permutations afforded each application an opportunity to run with each of the other three applications within the same memory domain:

- ▶ Soplex and Sphinx ran in a memory domain, while Games and Namd shared another memory domain.

- ▶ Sphinx was paired with Games, while Soplex shared a domain with Namd.

- ▶ Sphinx was paired with Namd, while Soplex ran in the same domain with Games.

Figure 2 contrasts the best performance of each application with its worst performance. The performance levels are indicated in terms of the

percentage of degradation from solo execution time (when the application ran alone on the system), meaning that the lower the numbers, the better the performance.

There is a dramatic difference between the best and the worst schedules, as shown in the figure. The workload as a whole performed 20% better with the best schedule, while gains for individual applications Soplex and Sphinx were as great as 50%. This indicates a clear incentive for assigning applications to cores according to the best possible schedule. While a contention-oblivious scheduler might accidentally happen upon the best schedule, it could just as well run the worst schedule. A contention-aware scheduler, on the other hand, would be better positioned to choose a schedule that performs well.

This article describes an investigation of a thread scheduler that would mitigate resource contention on multicore processors. Although we began this investigation using an analytical modeling approach that would be difficult to implement online, we ultimately arrived at a scheduling method that can be easily implemented online with a modern operating system or even prototyped at the user level. To share a complete understanding of the problem, we describe both the offline and online modeling approaches. The article concludes with some actual performance data that shows the impact contention-aware scheduling techniques can have on the performance of applications running on currently available multicore systems.

To make this study tractable we made the assumption that the threads do not share any data (that is, they belong either to different applications or to the same application where each thread works on its own data set). If threads share data, they may actually

Figure 1. A schematic view of a multicore system with two memory domains representing the architecture of Intel Quad-Core Xeon processors.

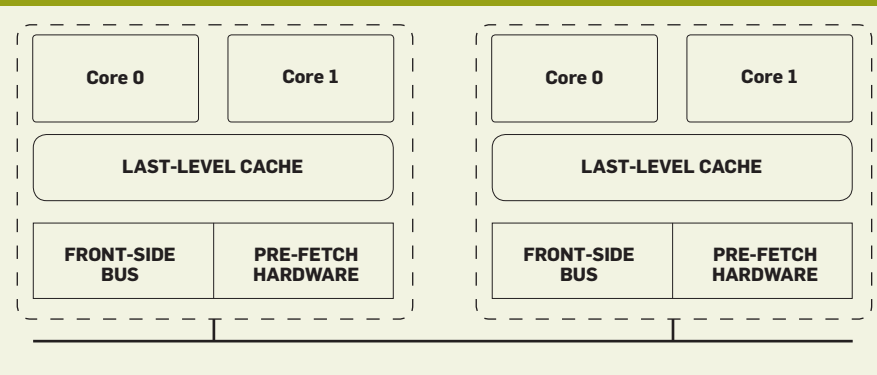


Figure 2. Percentage of performance degradation over a solo run achieved in two different scheduling assignments: the best and the worst. The lower the bar, the better the performance.

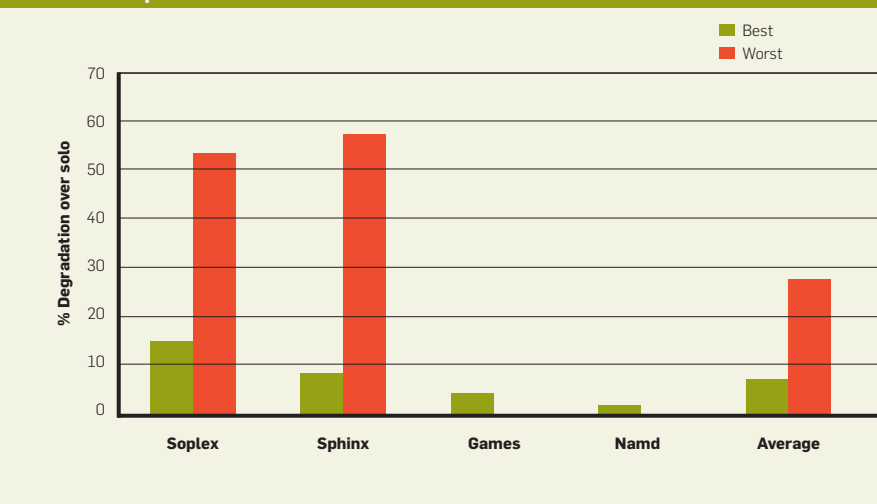
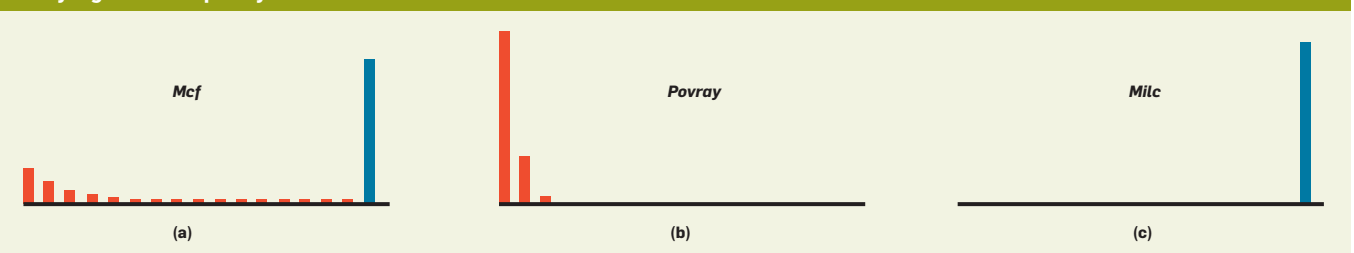


Figure 3. *Mcf* (a) is an application with a rather poor temporal locality, hence the low reuse frequency and high miss frequency. *Povray* (b) has excellent temporal locality. *Milc* (c) rarely reuses its data, therefore showing a very low reuse frequency and a very high miss frequency.



benefit from running in the same domain. In that case, they may access the shared resources cooperatively; for example, prefetch the data for each other into the cache. While the effects of cooperative sharing must be factored into a good thread-placement algorithm for multicores, this subject has been explored elsewhere.⁹ The focus here is managing resource contention.

Understanding Resource Contention

To build a contention-aware scheduler, we must first understand how to model contention for shared resources. Modeling allows us to predict whether a particular group of threads is likely to compete for shared resources and to what extent. Most of the academic work in this area has focused on modeling contention for LLCs, as this was believed to have the greatest effect on performance. This is where we started our investigation as well.

Cache contention occurs when two or more threads are assigned to run on the cores of the same memory domain (for example, Core 0 and Core 1 in Figure 1). In this case, the threads share the LLC. A cache consists of cache lines that are allocated to hold the memory of threads as the threads issue cache requests. When a thread requests a line that is not in the cache—that is, when it issues a cache miss—a new cache line must be allocated. The issue here is that when a cache line must be allocated but the cache is full (which is to say whenever all the other cache lines are being used to hold other data), some data must be evicted to free up a line for the new piece of data. The evicted line might belong to a different thread from the one that issued the cache miss (modern CPUs do not assure any fairness in that regard), so an aggressive thread might end up evicting data for some other thread and thus hurting its performance.

Although several researchers have proposed hardware mechanisms for mitigating LLC contention,^{3,7} to the best of our knowledge these have not been implemented in any currently available systems. We therefore looked for a way to address contention in the systems that people are running now and ended up turning our attention to scheduling as a result. Before building

a scheduler that avoids cache contention, however, we needed to find ways to predict contention.

There are two schools of thought regarding the modeling of cache contention. The first suggests that considering the LLC miss rate of the threads is a good way to predict whether these threads are likely to compete for the cache (the miss rate under contention is an equally good heuristic as the solo miss rate). A miss rate is the number of times per instruction when a thread fails to find an item in the LLC and so must fetch it from memory. The reasoning is that if a thread issues lots of cache misses, it must have a large cache working set, since each miss results in the allocation of a new cache line. This way of thinking also maintains that any thread that has a large cache working set must suffer from contention (since it values the space in the cache) while inflicting contention on others.

This proposal is contradicted by followers of the second school of thought, who reason that if a thread hardly ever reuses its cached data—as would be the case with a video-streaming application that touches the data only once—it will not suffer from contention even if it brings lots of data into the cache. That is because such a thread needs very little space to keep in cache the data that it actively uses. This school of thought advocates that to model cache contention one must consider the memory-reuse pattern of the thread. Followers of this approach therefore created several models for shared cache contention based on memory-reuse patterns, and they have demonstrated that this approach predicts the extent of contention quite accurately.³ On the other hand, only limited experimental data exists to support the plausibility of the other approach to modeling cache contention.⁵ Given the stronger evidence in favor of the memory-reuse approach, we built our prediction model based on that method.

A memory-reuse pattern is captured by a memory-reuse profile, also known as the stack-distance² or reuse-distance profile.¹ Figure 3 shows examples of memory-reuse profiles belonging to applications in the SPEC CPU2006 suite. The red bars on the left show the reuse frequency (how often the data is reused), and the blue bar on the right in-

dicates the miss frequency (how often that application misses in the cache). The reuse-frequency histogram shows the locality of reused data. Each bar represents a range of reuse distances—these can be thought of as the number of time steps that have passed since the reused data was last touched. An application with a very good temporal locality will have many high bars to the left of the histogram (Figure 3b), as it would reuse its data almost immediately. This particular application also has negligible miss frequency. An application with a poor temporal locality will have a flatter histogram and a rather high miss frequency (Figure 3a). Finally, an application that hardly ever reuses its data will result in a histogram indicating negligible reuse frequency and a very high miss frequency (Figure 3c).

Memory-reuse profiles have been used in the past to effectively model the contention between threads that share a cache.³ These models, the details of which we omit from this article, are based on the shape of memory-reuse profiles. One such model, the SDC (stack distance competition) examines the reuse frequency of threads sharing the cache to determine which of the threads is likely to “win” more cache space; the winning thread is usually the one with the highest overall reuse frequency. Still, the SDC model, along with all the other models based on memory-reuse profiles, was deemed too complex for our purposes. After all, our goal was to use a model in an operating-system scheduler, meaning it needed to be both efficient and lightweight. Furthermore, we were interested in finding methods for approximating the sort of information memory-reuse profiles typically afford using just the data that’s available at runtime, since memory-reuse profiles themselves are very difficult to obtain at runtime. Methods for obtaining these profiles online require unconventional hardware⁷ or rely on hardware performance counters available only on select systems.⁸

Our goal, therefore, was to capture the essence of memory-reuse profiles in a simple metric and then find a way to approximate this metric using data that a thread scheduler can easily obtain online. To this end, we discovered that memory-reuse profiles are highly

successful at modeling contention largely because they manage to capture two important qualities related to contention: sensitivity and intensity. Sensitivity measures how much a thread suffers whenever it shares the cache with other threads. Intensity, on the other hand, measures how much a thread hurts other threads whenever it shares a cache with them. Measuring sensitivity and intensity appealed to us because together they capture the key information contained within memory-reuse profiles; we also had some ideas about how they could be approximated using online performance data. Before learning how to approximate sensitivity and intensity, however, we needed to confirm that these were indeed good bases for modeling cache contention among threads. To accomplish that, we formally derived the sensitivity and intensity metrics based on data in the memory-reuse profiles. After confirming that the metrics derived in this way did indeed accurately model contention, we could then attempt to approximate them using just online data.

Accordingly, we derived sensitivity S and intensity Z for an application using data from its memory-reuse profile. To compute S , we applied an aggregation function to the reuse-frequency histogram. Intuitively, the higher the reuse frequency, the greater an application is likely to suffer from the loss of cache space due to contention with another application—signifying a higher sensitivity. To compute Z , we simply used the cache-access rate, which can be inferred from the memory-reuse profile. Intuitively, the higher the access rate, the higher the degree of competition

from the thread in question since the high access rate shows that it allocates new cache lines while retaining old ones. Details for the derivation of S and Z are described in another article.¹⁰

Using the metrics S and Z , we then created another metric called *Pain*, where the *Pain* of thread A due to sharing a cache with thread B is the product of the sensitivity of A and the intensity of B , and vice versa. A combined *Pain* for the thread pair is the sum of the *Pain* of A due to B and the *Pain* of B due to A , as shown here:

$$\begin{aligned} \text{Pain}(A|B) &= S_A * Z_B \\ \text{Pain}(B|A) &= S_B * Z_A \\ \text{Pain}(A,B) &= \text{Pain}(A|B) + \text{Pain}(B|A) \end{aligned}$$

Intuitively, $\text{Pain}(A|B)$ approximates the performance degradation of A when A runs with B relative to running solo. It will not capture the absolute degradation entirely accurately, but it is good for approximating relative degradations. For example, given two potential neighbors for A , the *Pain* metric can predict, which will cause a higher performance degradation for A . This is precisely the information a contention-aware scheduler would require.

The *Pain* metric shown here assumes that only two threads share a cache. There is evidence, however, showing this metric applies equally well when more than two threads share the cache. In that case, in order to compute the *Pain* for a particular thread as a consequence of running with all of its neighbors concurrently, the *Pain* owing to each neighbor must be averaged.

After developing the *Pain* metric

based on memory-reuse profiles, we looked for a way to approximate it using just the data available online via standard hardware performance counters. This led us to explore two performance metrics to approximate sensitivity and intensity: the cache-miss rate and the cache-access rate. Intuitively these metrics correlate with the reuse frequency and the intensity of the application. Our findings regarding which metric offers the best approximation are surprising, so to maintain some suspense, we postpone their revelation until the section entitled “Evaluation of Modeling Techniques.”

Using Contention Models in a Scheduler

In evaluating the new models for cache contention, our goal was to determine how effective the models would be for constructing contention-free thread schedules. We wanted the model to help us find the best schedule and avoid the worst one (recall Figure 2). Therefore, we evaluated the models on the merit of the schedules they managed to construct. With that in mind, we describe here how the scheduler uses the *Pain* metric to find the best schedule.

To simplify the explanation for this evaluation, we have a system with two pairs of cores sharing the two caches (as illustrated in Figure 1), but as mentioned earlier, the model also works well with more cores per cache. In this case, however, we want to find the best schedule for four threads. The scheduler would construct all the possible permutations of threads on this system, with each of the permutations being unique in terms of how the threads are paired on each memory domain. If we have four threads— A , B , C , and D —there will be three unique schedules: (1) $\{(A,B), (C,D)\}$; (2) $\{(A,C), (B,D)\}$; and (3) $\{(A,D), (B,C)\}$. Notation (A,B) means that threads A and B are co-scheduled in the same memory domain. For each schedule, the scheduler estimates the *Pain* for each pair: in schedule $\{(A,B), (C,D)\}$ the scheduler would estimate $\text{Pain}(A,B)$ and $\text{Pain}(C,D)$ using the equations presented previously. Then it averages the *Pain* values of the pairs to estimate the *Pain* for the schedule as a whole. The schedule with the lowest *Pain* is deemed to be the estimated

Figure 4. Computing the pain for all possible schedules. The schedule with the lowest *Pain* is chosen as the estimated best schedule.

Schedule $\{(A,B), (C,D)\}$:	$\text{Pain} = \text{Average}(\text{Pain}(A,B), \text{Pain}(C,D))$
Schedule $\{(A,C), (B,D)\}$:	$\text{Pain} = \text{Average}(\text{Pain}(A,C), \text{Pain}(B,D))$
Schedule $\{(A,D), (B,C)\}$:	$\text{Pain} = \text{Average}(\text{Pain}(A,D), \text{Pain}(B,C))$

Figure 5. The metric for comparing the actual best and the estimated best schedule.

Estimated Best Schedule $\{(A,B), (C,D)\}$:	$\text{DegradationEst} = \text{Average}(\text{Degrad}(A B), \text{Degrad}(B A), \text{Degrad}(C D), \text{Degrad}(D C))$
Actual Best Schedule $\{(A,C), (B,D)\}$:	$\text{DegradationAct} = \text{Average}(\text{Degrad}(A C), \text{Degrad}(C A), \text{Degrad}(B D), \text{Degrad}(D B))$
Degradation over actual best =	$\left(\frac{\text{DegradationEst}}{\text{DegradationAct}} - 1 \right) \times 100\%$

best schedule. Figure 4 is a summary of this procedure.

The estimated best schedule can be obtained either by using the *Pain* metric constructed via actual memory-reuse profiles or by approximating the *Pain* metric using online data.

Once the best schedule has been estimated, we must compare the performance of the workload in the estimated best schedule with the performance achieved in the *actual* best schedule. The most direct way of doing this is to run the estimated best schedule on real hardware and compare its performance with that of the actual best schedule, which can be obtained by running all schedules on real hardware and then choosing the best one. Although this is the most direct approach (which we used for some experiments in the study), it limited the number of workloads we could test because running all possible schedules for a large number of workloads is time consuming.

To evaluate a large number of workloads in a short amount of time, we invented a semi-analytical evaluation methodology that relies partially on data obtained from tests on a real system and otherwise applies analytical techniques. Using this approach, we selected 10 benchmark applications from the SPEC CPU2006 suite to use in the evaluation. They were chosen using the minimum spanning-tree-clustering method to ensure the applications represented a variety of memory-access patterns.

We then ran all possible pairings of these applications on the experimental platform, a Quad-Core Intel Xeon system, where each Quad-Core processor looked like the system depicted in Figure 1. In addition to running each possible pair of benchmark applications on the same memory domain, we ran each benchmark alone on the system. This gave us the measure for the *actual degradation in performance* for each benchmark as a consequence of sharing a cache with another benchmark, as opposed to running solo. Recall that degradation relative to performance in the solo mode is precisely the quantity approximated by the *Pain* metric. So by comparing the scheduling assignment constructed based on the actual degradation to that constructed based on the *Pain* metric, we can evaluate how



In evaluating the new models for cache contention, our goal was to determine how effective the models would be for constructing contention-free thread schedules.



good the *Pain* metric is in finding good scheduling assignments.

Having the actual degradations enabled us to construct the *actual* best schedule using the method shown in Figure 5. The only difference was that, instead of using the model to compute $Pain(A,B)$, we used the actual performance degradation that we had measured on a real system (with *Pain* being equal to the sum of degradation of A running with B relative to running solo and the degradation of B running with A relative to running solo).

Once we knew the actual best schedule, we needed a way to compare it with the estimated best schedule. The performance metric was the average degradation relative to solo execution for all benchmarks. For example, suppose the estimated best schedule was $\{(A,B), (C,D)\}$, while the actual best schedule was $\{(A,C), (B,D)\}$. We computed the average degradation for each schedule to find the difference between the degradation in the estimated best versus the degradation in the actual one, as indicated in Figure 5. The notation $Degrad(A|B)$ refers to the measured performance degradation of A when running alongside B, relative to A running solo.

This illustrates how to construct estimated best and actual best schedules for any four-application workload on a system with two memory domains so long as actual pair-wise degradations for any pair of applications have been obtained on the experimental system. Using the same methodology, we can evaluate this same model on systems with a larger number of memory domains. In that case, the number of possible schedules grows, but everything else in the methodology remains the same. In using this methodology, we assumed that the degradation for an application pair (A,B) would be the same whether it were obtained on a system where only A and B were running or on a system with other threads, such as (C,D) running alongside (A,B) on another domain. This is not an entirely accurate assumption since, with additional applications running, there will be a higher contention for the front-side bus. Although there will be some error in estimating schedule-average degradations under this method, the error is not great enough to affect

Figure 6. The percentage by which performance of schedules estimated to be best according to various modeling techniques varies from the actual best schedules. Low bars are good.

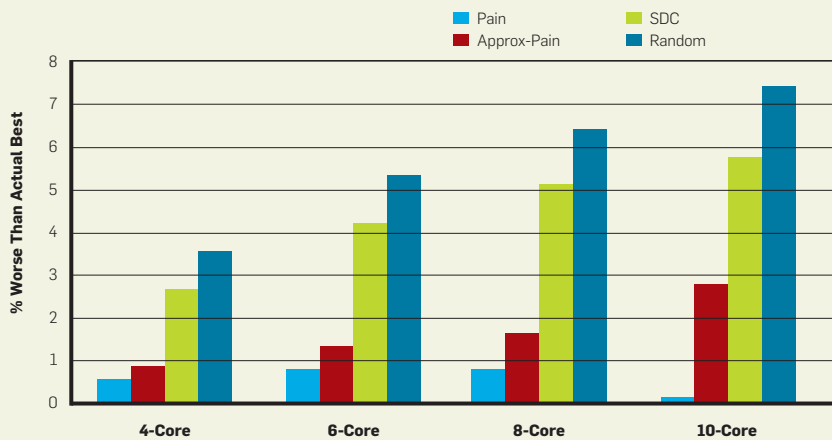


Figure 7. A breakdown of factors causing performance degradation due to contention for shared hardware on multicore systems based on tests using select applications in the SPEC CPU2006 suite.

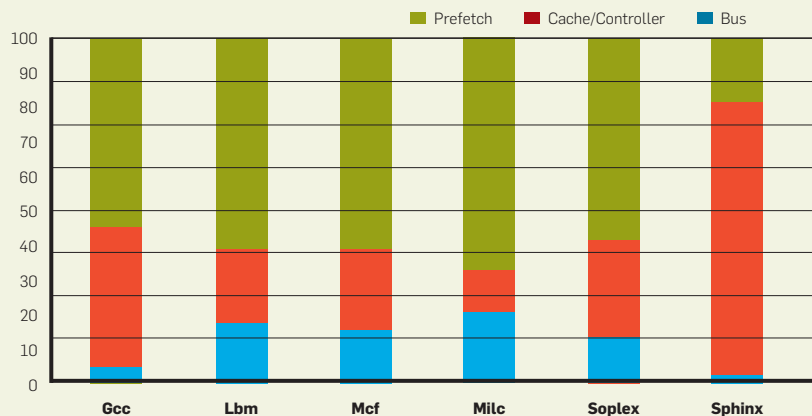
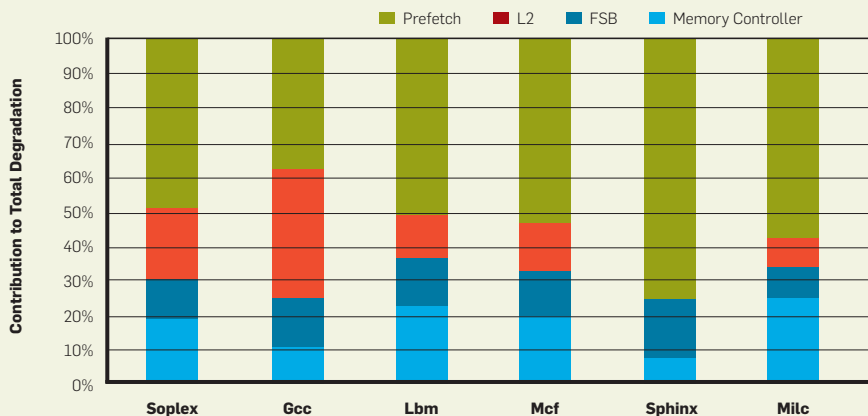


Figure 8. A breakdown of factors causing performance degradation due to contention for shared hardware on multicore systems based on tests using select applications in the SPEC CPU2006 suite. These experiments were performed on an Intel Xeon (Cloverton) processor. We also obtained data showing that cache contention is not dominant on AMD Opteron systems.



the evaluation results significantly. Certainly, the error is not large enough to lead to a choice of the “wrong” best schedule.

Evaluation of Modeling Techniques

Here, we present the results obtained using our semi-analytical methodology, followed by the performance results obtained via experiments only. Figure 6 compares the degradation over the actual best schedules (the method for which was indicated earlier) with estimated best schedules constructed using various methods. The blue bar indicating *Pain* is the model that uses memory-reuse profiles to estimate *Pain* and find the best schedule (the method for which was set forth in Figure 4). In the red bar indicating *Approx-Pain*, the *Pain* for a given application running with another is estimated with the aid of data obtained online (we explain which data this is at the end of this section); once *Pain* has been estimated, we can once again use the method shown in Figure 4. In *SDC*, a previously proposed model based on memory-reuse profiles³ can be used to estimate the performance degradation of an application when it shares a cache with a co-runner. This estimated degradation can then be used in place of *Pain(A|B)*; apart from that, the method shown in Figure 4 applies. Although *SDC* is rather complex for use in a scheduler, we compared it with our new models to evaluate how much performance was being sacrificed by using a simpler model. Finally, in Figure 6 the bar labeled *Random* shows the results for selecting a random-thread placement.

Figure 6 shows how much worse the schedule chosen with each method ended up performing relative to the actual best schedule. This value was computed using the method shown in Figure 4. Ideally, this difference from the actual best ought to be small, so in considering Figure 6, remember that low bars are good. The results for four different systems—with four, six, eight, and 10 cores—are indicated. In all cases there were two cores per memory domain. (Actual results from a system with a larger number of cores per memory domain are shown later.) Each bar represents the average for all the benchmark pairings that could be constructed out of our 10 representa-

tive benchmarks on a system with a given number of cores, such that there is exactly one benchmark per core. On four- and six-core systems, there were 210 such combinations, whereas an eight-core system had 45 combinations, and a 10-core system had only one combination. For each combination, we predicted the best schedule. The average performance degradation from the actual best for each of these estimated best schedules is reported in each bar in the figure.

The first thing we learned from the metric in Figure 5 was that the *Pain* model is effective for helping the scheduler find the best thread assignment. It produces results that are within 1% of the actual best schedule. (The effect on the actual execution time is explored later). We also found that choosing a random schedule produces significantly worse performance, especially as the number of cores grows. This is significant in that a growing number of cores is the expected trend for future multicore systems.

Figure 6 also indicates that the *Pain* approximated by way of an online metric works very well, coming within just 3% of the actual best schedule. At the same time, the SDC, a well-proven model from an earlier study, turns out to be less accurate. These results—both the effectiveness of the approximated *Pain* metric and the disappointing performance of the older SDC model—were quite unexpected. Who could have imagined that the best way to approximate the *Pain* metric would be to use the LLC miss rate? In other words, the LLC miss rate of a thread is the best predictor of both how much the thread will suffer from contention (its sensitivity) and how much it will hurt others (its intensity). As explained at the beginning of this article, while there was limited evidence indicating that the miss rate predicts contention, it ran counter to the memory-reuse-based approach, which was supported by a much larger body of evidence.

Our investigation of this paradox led us to examine the causes of contention on multicore systems. We performed several experiments that aimed to isolate and quantify the degree of contention for various types of shared resources: cache, memory controller, bus, prefetching hardware. The precise

setup of these experiments is described in another study.¹⁰ We arrived at the following findings.

First, it turns out that contention for the shared cache—the phenomenon by which competing threads end up evicting each others' cache lines from the cache—is not the main cause of performance degradation experienced by competing applications on multicore systems. Contention for other shared resources, such as the front-side bus, prefetching resources, and the memory controller are the dominant causes for performance degradation (see Figure 7). That is why the older memory-reuse model, designed to model cache contention only, was not effective in our experimental environment. The authors of that model evaluated it on a simulator that did not model contention for resources other than shared cache, and it turns out that, when applied to a real system where other types of contention were present, the model did not prove effective.

On the other hand, cache-miss rate turned out to be an excellent predictor for contention for the memory controller, prefetching hardware, and front-side bus. Each application in our model was co-scheduled with the *Milc* application to generate contention. Given limitations of existing hardware counters, it was difficult to separate the effects of contention for prefetching hardware itself and the effects of additional contention for memory controller and front-side bus caused by prefetching. Therefore, the impact of prefetching shows the combined effect of these two factors.

An application issuing many cache misses will occupy the memory controller and the front-side bus, so it will not only hurt other applications that use that hardware, but also end up suffering itself if this hardware is usurped by others. An application aggressively using prefetching hardware will also typically have a high LLC miss rate, because prefetch requests for data that is not in the cache are counted as cache misses. Therefore, a high miss rate is also an indicator of the heavy use of prefetching hardware.

In summary, our investigation of contention-aware scheduling algorithms has taught us that high-miss-rate applications must be kept apart.

That is, they should not be co-scheduled in the same memory domain. Although some researchers have already suggested this approach, it is not well understood why using the miss rate as a proxy for contention ought to be effective, particularly in that it contradicts the theory behind the popular memory-reuse model. Our findings should help put an end to this controversy.

Based on this new knowledge, we have built a prototype of a contention-aware scheduler that measures the miss rates of online threads and decides how to place threads on cores based on that information. Here, we present some experimental data showing the potential impact of this contention-aware scheduler.

Implications

Based on our understanding of contention on multicore processors, we have built a prototype of a contention-aware scheduler for multicore systems called Distributed Intensity Online (DIO). The DIO scheduler distributes intensive applications across memory domains (and by intensive we mean those with high LLC miss rates) after measuring online the applications' miss rates. Another prototype scheduler, called Power Distributed Intensity (Power DI), is intended for scheduling applications in the workload across multiple machines in a data center. One of its goals is to save power by determining how to employ as few systems as possible without hurting performance. The following are performance results of these two schedulers.

Distributed Intensity Online. Different workloads offer different opportunities to achieve performance improvements through the use of a contention-aware scheduling policy. For example, a workload consisting of non-memory-intensive applications (those with low cache miss rates) will not experience any performance improvement since there is no contention to alleviate in the first place. Therefore, for our experiments we constructed eight-application workloads containing from two to six memory-intensive applications. We picked eight workloads in total, all consisting of SPEC CPU2006 applications, and then executed them under the DIO and the default Linux sched-

uler on an AMD Opteron system featuring eight cores—four per memory domain. The results are shown in Figure 8. The performance improvement relative to default has been computed as the average improvement for all applications in the workload (since not all applications are memory intensive, some do not improve). We can see that DIO renders workload-average performance improvements of up to 11%.

Another potential use of DIO is as a

way to ensure QoS (quality of service) for critical applications since DIO essentially provides a means to make sure the worst scheduling assignment is never selected, while the default scheduler may occasionally suffer as a consequence of a bad thread placement. Figure 9 shows for each of the applications as part of the eight test workloads its worst-case performance under DIO relative to its worst-case performance under the default Linux scheduler. The

numbers are shown in terms of the percentage of improvement or the worst-case behavior achieved under DIO relative to that encountered with the default Linux scheduler, so higher bars in this case are better. We can see that some applications are as much as 60% to 80% better off with their worst-case DIO execution times, and in no case did DIO do significantly worse than the default scheduler.

Power Distributed Intensity. One of the most effective ways to conserve CPU power consumption is to turn off unused cores or entire memory domains in an active system. Similarly, if the workload is running on multiple machines—for example, in a data center—power savings can be accomplished by clustering the workload on as few servers as possible while powering down the rest. This seemingly simple solution is a double-edged sword, however, because clustering the applications on just a few systems may cause them to compete for shared system resources and thus suffer performance loss. As a result, more time will be needed to complete the workload, meaning that more energy will be consumed. In an attempt to save power it is also necessary to consider the impact that clustering can have on performance. A metric that takes into account both the energy consumption and the performance of the workload is the energy-delay product (EDP).⁴ Based on our findings about contention-aware scheduling, we designed Power DI, a scheduling algorithm meant to save power without hurting performance.

Power DI works as follows: Assuming a centralized scheduler has knowledge of the entire computing infrastructure and distributes incoming applications across all systems, Power DI clusters all incoming applications on as few machines as possible, except for those applications deemed to be memory intensive. Similarly, within a single machine, Power DI clusters applications on as few memory domains as possible, with the exception of memory-intensive applications. These applications are not co-scheduled on the same memory domain with another application unless the other application has a very low cache miss rate (and thus a low memory intensity). To determine if an application is memory-intensive,

Figure 9. Performance of eight workloads under DIO relative to the default Linux scheduler.

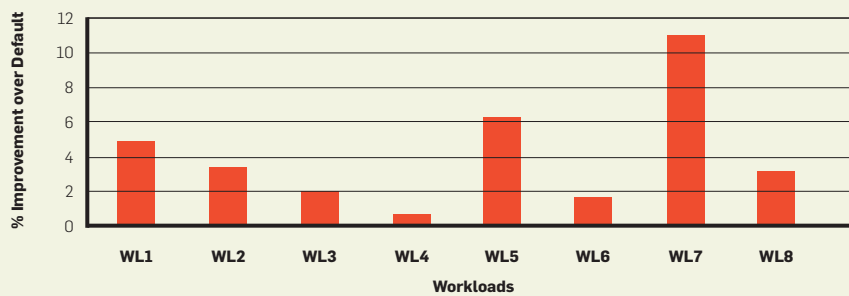


Figure 10. Worst-case performance for each of the applications included as part of the eight test workloads.

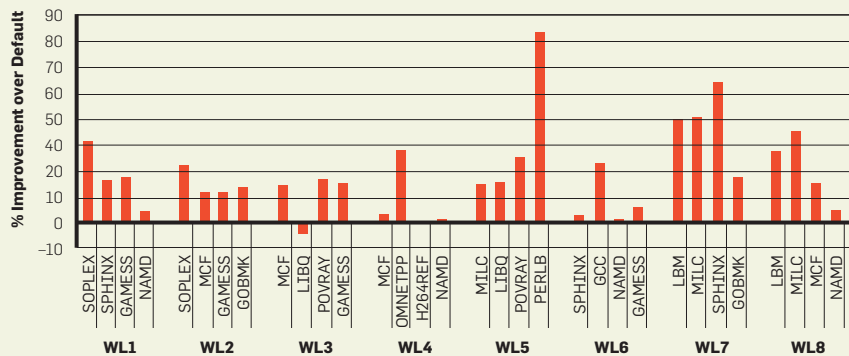
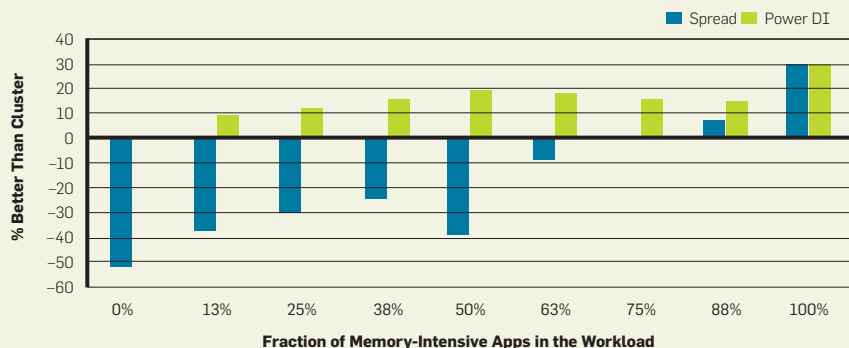


Figure 11. Percentage reduction in EDP.



Power DI uses an experimentally derived threshold of 1,000 misses per million instructions; an application whose LLC miss rate exceeds that amount is considered memory intensive.

Although we did not have a data-center setup available to us to evaluate this algorithm, we simulated a multi-server environment in the following way. The in-house AKULA scheduling simulator created a schedule for a given workload on a specified data-center setup, which in this case consisted of 16 eight-core systems, assumed by the simulator to be Intel Xeon dual quad-core servers. Once the simulated scheduler decided how to assign applications across machines and memory domains within each machine, we computed the performance of the entire workload from the performance of the applications assigned by the scheduler to each eight-core machine. The performance on a single system could be easily measured via experimentation. This simulation method was appropriate for our environment, since there was no network communication among the running applications, meaning that inferring the overall performance from the performance of individual system was reasonable.

To estimate the power consumption, we used a rather simplistic model (measurements with the actual power meter are still under way) but captured the right relationships between power consumed in various load conditions. We assumed that a memory domain where all the cores are running applications consumes one unit of power. A memory domain where one out of two cores are busy consumes 0.75 units of power. A memory domain where all cores are idle is assumed to be in a very low power state and thus consumes 0 units of power. We did not model the latency of power-state transitions.


We constructed a workload of 64 SPEC CPU2006 applications randomly drawn from the benchmark suite. We varied the fraction of memory-intensive applications in the workload from zero to 100%. The effectiveness of scheduling strategies differed according to the number of memory-intensive applications. For example, if there were no memory-intensive applications, it was perfectly fine to cluster all the applications to the greatest extent

possible. Conversely, if all the applications were memory intensive, then the best policy was to spread them across memory domains so that no two applications would end up running on the same memory domain. An intelligent scheduling policy must be able to decide to what extent clustering must be performed given the workload at hand.

Figure 10 shows the EDP for three different scheduling methods: Power DI, a naïve Spread method (which always spreads applications across machines to the largest extent possible), and the Cluster method (which in an attempt to save power always clusters applications on as few machines and as few memory domains as possible). The numbers are shown as a percentage reduction in the EDP (higher is better) of Power DI and Spread over Cluster.

We can see that when the fraction of memory-intensive applications in the workload is low, the naïve Spread method does much worse than the Cluster method, but it beats Cluster as that fraction increases. Power DI, on the other hand, is able to adjust to the properties of the workload and minimize EDP in all cases, beating both Spread and Cluster—or at least matching them—for every single workload.

Conclusion

Contention for shared resources significantly impedes the efficient operation of multicore systems. Our research has provided new methods for mitigating contention via scheduling algorithms. Although it was previously thought that the most significant reason for contention-induced performance degradation had to do with shared cache contention, we found that other sources of contention—such as shared prefetching hardware and memory interconnects—are just as important. Our heuristic—the LLC miss rate—proves to be an excellent predictor for all types of contention. Scheduling algorithms that use this heuristic to avoid contention have the potential to reduce the overall completion time for workloads, avoid poor performance for high-priority applications, and save power without sacrificing performance. 

Related articles on queue.acm.org

Maximizing Power Efficiency with Asymmetric Multicore Systems

Alexandra Fedorova, Juan Carlos Saez, Daniel Shelepov, and Manuel Prieto
<http://queue.acm.org/detail.cfm?id=1658422>

The Future of Microprocessors

Kunle Olukotun
<http://queue.acm.org/detail.cfm?id=1095418>

References

- Berg, E. and Hagersten, E. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2004), 20–27.
- Cascaval, C., DeRose, L., Padua, D.A. and Reed, D. 1999. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing* (1999), 365–379.
- Chandra, D., Guo, F., Kim, S. and Solihni, Y. Predicting inter-thread cache contention on a multiprocessor architecture. In *Proceedings of the 11th International Symposium on High-performance Computer Architecture* (2005), 340–351.
- Gonzalez, R. and Horowitz, M. Energy dissipation in general-purpose microprocessors. *IEEE Journal of Solid State Circuits* 31, 9 (1999), 1277–1284.
- Knauerhase, R., Brett, P., Hohlt, B., Li, T. and Hahn, S. Using OS observations to improve performance in multicore systems. *IEEE Micro* (2008), 54–66.
- SPEC: Standard Performance Evaluation Corporation; <http://www.spec.org>.
- Suh, G., Devadas, S. and Rudolph, L. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-performance Computer Architecture* (2002), 117.
- Tam, D., Azimi, R., Soares, L. and Stumm, M. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), 121–132.
- Tam, D., Azimi, R. and Stumm, M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (2007), 47–58.
- Zhuravlev, S., Blagodurov, S. and Fedorova, A. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (2010).

Alexandra Fedorova is an assistant professor of computer science at Simon Fraser University in Vancouver, Canada, where she co-founded the SYNAR (Systems, Networking and Architecture) research lab. Her research interests span operating systems and virtualization platforms for multicore processors, with a specific focus on scheduling. Recently she started a project on tools and techniques for parallelization of video games, which has led to the design of a new language for this domain.

Sergey Blagodurov is a Ph.D. student in computer science at Simon Fraser University, Vancouver, Canada. His research focuses on operating-system scheduling on multicore processors and exploring new techniques to deliver better performance on non-uniform memory access (NUMA) multicore systems.

Sergey Zhuravlev is a Ph.D. student in computer science at Simon Fraser University, Vancouver, Canada. His recent research focuses on scheduling on multiprocessor systems to avoid shared resource contention as well as simulating computing systems.