Modeling Performance Variation Due to Cache Sharing

Andreas Sandberg, Andreas Sembrant, Erik Hagersten and David Black-Schaffer
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{andreas.sandberg, andreas.sembrant, eh, david.black-schaffer}@it.uu.se

Abstract

Shared cache contention can cause significant variability in the performance of co-running applications from run to run. This variability arises from different overlappings of the applications' phases, which can be the result of offsets in application start times or other delays in the system. Understanding this variability is important for generating an accurate view of the expected impact of cache contention. However, variability effects are typically ignored due to the high overhead of modeling or simulating the many executions needed to expose them.

This paper introduces a method for efficiently investigating the performance variability due to cache contention. Our method relies on input data captured from native execution of applications running in isolation and a fast, phase-aware, cache sharing performance model. This allows us to assess the performance interactions and bandwidth demands of co-running applications by quickly evaluating hundreds of overlappings.

We evaluate our method on a contemporary multicore machine and show that performance and bandwidth demands can vary significantly across runs of the same set of co-running applications. We show that our method can predict application slowdown with an average relative error of 0.41% *(maximum* 1.8%*) as well as bandwidth consumption. Using our method, we can estimate an application pair's performance variation* $213\times$ *faster, on average, than native execution.*

1. Introduction

Shared caches in contemporary multicores have repeatedly been shown to be critical resources for performance [15, 23, 28, 8, 17]. A significant amount of research has investigated the impact of cache sharing on application performance [23, 30, 12, 11]. However, most previous research provides a single value for the slowdown of an application pair due to cache sharing and ignores the variability that occurs across multiple runs. This variability occurs due to different overlappings of application phases that occur when they are offset in time. As the different phases have varying sensitivities to contention for the shared cache, the result is a wide range of slowdowns for the same application pair.

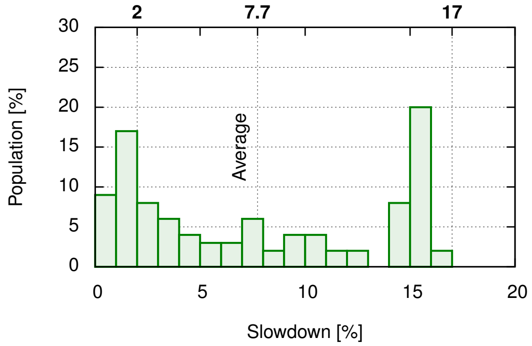


Figure 1. Performance distribution for astar co-running together with bwaves on an Intel Xeon E5620 based system. ***Ignoring performance variability can be misleading, since the average (7.7%) hides the fact that the performance can vary between 1% and 17% depending on how the two applications' phases overlap.***

In multicore systems, there can be large performance variations due to cache contention, since an application's performance depends on how its memory accesses are interleaved with other applications' memory accesses. For example, when running astar/lakes and bwaves from SPEC CPU2006, we observe an average slowdown of 8% for astar compared to running it in isolation. However, the slowdown can vary between 1% and 17% depending on how the two applications' phases overlap. Figure 1 shows astar's slowdown distribution based on 100 runs with different offsets in starting times. A developer assessing the performance of these applications could draw the wrong conclusions from a single run, or even a few runs, since the probability of measuring a slowdown smaller than 2%

is more than 25%, while the average slowdown is almost 8% and the maximum slowdown is 17%.

In order to accurately estimate the performance of a mixed workload, we need to run it multiple times and estimate its performance distribution. This is a both time- and resource-consuming process. The distribution in Figure 1 took almost seven hours to generate; our method reproduces the same performance distribution in less than $40\,\mathrm{s}$.

To do this, we combine the cache sharing model proposed by Sandberg et al. [16], the phase detection framework developed by Sembrant et al. [19], and the co-execution phase optimizations proposed by Van Biesbrouck et al. [25]. This allows us to efficiently predict the performance and bandwidth requirements of mixed workloads. In addition, the input data to the cache model is captured using low-overhead profiling [7] of each application running in isolation. This means that only a small number of profiling runs need to be done on the target machine. The modeling can then be performed quickly for a large number of mixed workloads and runs.

The main contributions of this paper are:

- An extension to a statistical cache-sharing model [16] to handle time-dependent execution phases.
- A fast and efficient method to predict the performance variations due to shared cache contention on modern hardware by combining a cache sharing model [16] with phase optimizations [19, 25].
- A comparison with previous cache-sharing methods [16] demonstrating a $2.78\times$ improvement in accuracy (the relative error is reduced from 1.14% to 0.41%) and a $3.5\times$ reduction in maximum error (from 6.3% to 1.8%).
- An analysis of how different types of phase behavior impact the performance variations in mixed workloads.

2. Putting it Together

Our method combines and extends three existing pieces of infrastructure: a cache sharing model [16], a low-overhead cache analysis tool [7], and a phase detection framework [19]. In this section, we describe the different pieces and how we extend them.

2.1. Cache Sharing

We use the cache sharing model proposed by Sandberg et al. [16] for cache modeling. It accurately predicts the amount of cache used, CPI, and bandwidth demand for an application in a mixed workload of co-executing single-threaded applications. The input to the model is a set of independent *application profiles*. These profiles contain information about how the *miss rate* (misses per cycle) and *hit rate* (hits per cycle) vary for an application as a function of cache size. We use the Cache Pirating [7] technique (discussed below) to capture the model's input data.

The model conceptually partitions the cache into two parts with different reuse behavior. The model keeps frequently reused data safe from replacements, while less frequently reused data shares the remaining cache space proportionally to its application's miss rate. The partitioning between frequently reused data and infrequently reused data is an application property that is cache size dependent (i.e., the partitioning depends on how much cache an application receives). The model uses an iterative solver that first solves cache sharing for the infrequently reused data and then updates partitioning between frequently reused data and infrequently reused data.

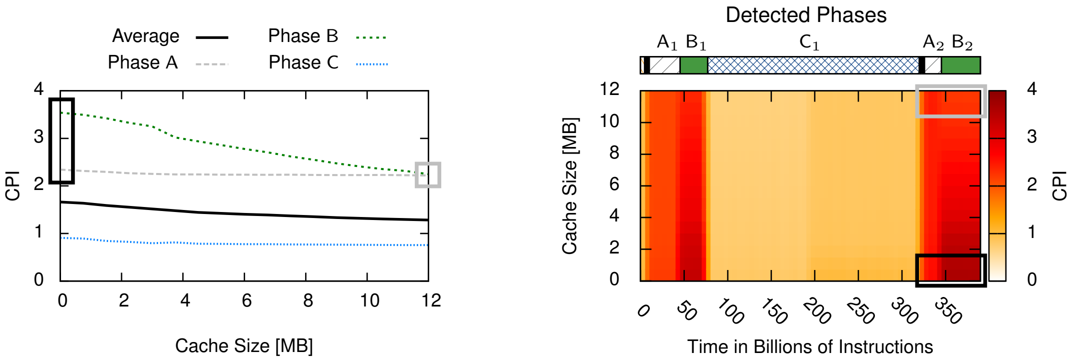
The model however only works on phase-less applications where the average behavior is representative of the entire application. In practice, most applications have phases. To handle this, we extend the model by slicing applications into multiple small time windows. As long as the windows are short enough, the model's assumption of constant behavior holds within the window. We then apply the model to a set of co-executing windows instead of data averaged across the entire execution.

2.2. Cache Pirating

The input to the cache sharing model is an application profile with information about cache miss rates and hit rates *as a function of cache size*. Traditionally, such profiles have been generated through simulation, but such an approach is slow and it is difficult to build accurate simulators for modern processor pipelines and memory systems. Instead, we use Cache Pirating [7] to collect the data. Cache Pirating solves both problems by measuring how an application behaves as a function of cache size on the target machine with very low overhead.

Cache Pirating uses hardware performance monitoring facilities to measure target application properties at runtime, such as cache misses, hits, and execution cycles. To measure this information for varying cache sizes, Cache Pirating co-runs a small cache intensive stress application with the target application. The amount of cache available to the target application is then varied by changing the cache footprint of the stress application. This allows Cache Pirating to measure any performance metric exposed by the target machine as a function of available cache size.

The cache pirate method produces average measurements for an entire application run. This is illustrated in Figure 2a. It shows CPI as a function of cache size for astar. The solid black line (Average) is the output produced



(a) Time oblivious

(b) Time aware

Figure 2. Performance (CPI) as a function of cache size as produced by Cache Pirating. Figure (a) shows the time-oblivious application average as a solid line. Figure (b) shows the time-dependent variability of the cache sensitivity and the phases identified by ScarPhase above. The behavior of the three largest phases vary significantly from the average as can be seen by the dashed lines in Figure (a).

with Cache Pirating.

Just examining the average behavior can however be misleading since most applications have time-dependent behavior. Figure 2b instead shows astar's CPI as a function of both time and cache size. As seen in the figure, the application displays three different *phases* of behavior: some parts of the application execute with a very high CPI (phase A & phase B), while other parts execute with a very low CPI (phase C). This information is lost unless time is taken into account.

In this paper, we extend the cache pirate method to produce time-dependent data by dividing the execution into *sample windows* by sampling the performance counters at regular intervals.

2.3. Phase Detection

A naive approach to phase-aware cache modeling would be to model the effect of every pair of measured input sample windows. However, to make the analysis more efficient, we incorporate application phase information. This enables us to analyze multiple sample windows with similar behavior at the same time, which reduces the number of times we need to invoke the cache sharing model.

We use the ScarPhase [19] library to detect and classify phases. ScarPhase is an execution-history based, low-overhead (2%), online phase-detection library. It examines the application's execution path to detect hardware independent phases [21, 14]. Such phases can be readily missed by performance counter based phase detection, while changes in executed code reflect changes in many different metrics [20, 21, 5, 22, 9, 18]. To leverage this, ScarPhase monitors what code is executed by dividing the application into windows and using hardware performance counters to sample which branches execute in a window. The address of each branch is hashed into a vector of counters called a basic block vector (BBV) [20]. Each entry in the vector shows how many times its corresponding branches were sampled during the window. The vectors are then used to determine phases by clustering them together using an online clustering algorithm [6]. Windows with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

The phases detected by ScarPhase can be seen in the top bar in Figure 2b for astar, with the longest phases labeled. This benchmark has three major phases; A, B and C, all with different cache behaviors. To highlight the differences in CPI, we have plotted the average CPI of each phase in Figure 2a. For example, phase A runs slower than C, since it has a higher CPI. Phase B is more sensitive to cache-size changes than phase A since phase B's CPI decreases with more cache.

The same phase can occur several times during execution. For example, phase A recurs two times, once in the beginning and once at the end of the execution. We refer to multiple repetitions of the same phase as *instances* of the same phase, e.g., A_1 and A_2 in Figure 2b.

In addition, Figure 2b also demonstrates the limitation of defining phases based on changes in hardware-specific metrics. For example, the CPI is very similar from 325 to 390 billion instructions when using 12 MB of cache (the gray rectangle), but clearly different when using less than 4 MB (the black rectangle). This difference is even more noticeable in Figure 2a when comparing phase A and B. A phase detection method looking at only the CPI would draw the conclusion that phase A and B are the same phase when the application receives 12 MB of cache, while in reality they are two very different phases. It is therefore important to find phases that are independent of the execution environment (e.g., co-scheduling).

3. Time Dependent Cache Sharing

The key difficulty in modeling time-dependent cache sharing is to determine which parts of the application (i.e., sample windows or phases) will co-execute. Since applications typically execute at different speeds depending on phase, we can not simply use the ith sample windows for each application since they may not overlap. For example, consider two applications with different executions rates (e.g., CPIs of 2 and 4), executing sample windows of 100 million instructions. The slower application with a CPI of 4 will take twice as long to finish executing its sample windows as the one with a CPI of 2. Furthermore, when they share a cache they impact each others execution rates. Instead, we advance time as follows:

1. Determine the cache sharing using the model for the current windows and the resulting CPI for each application due to its shared cache allocation.

2. Advance the fastest application (i.e., the one with lowest CPI) to its next sample window. The slower applications will not have had time to completely execute their windows. To handle this, their windows are first split into two smaller windows so that the first window ends at the same time the fastest applications sample window. Finally, time is advanced to the beginning of the latter windows.

This means that the cache model is applied several times per sample window, since each window is usually split at least twice. For example, when modeling the slowdown of astar co-executing together with bwaves, we invoke the cache sharing model roughly 13 000 times while astar only has 4 000 sample windows by itself.

We refer to the method described so far as the *window-based method* (Window) in the rest of paper. In the rest of this section, we will introduce two more methods, the *dynamic-window-based method* (Dynamic Window) and the *phase-based method* (Phase), which both use phase information to improve the performance by reducing number of times the cache sharing model needs to be applied[1].

3.1. Dynamic-Windows: Merging Sample-Windows

To improve performance we need to reduce the number of times the cache sharing model is invoked. To do this, we merge multiple adjacent sample windows belonging to the same phase into larger windows, a dynamic window. For example, in astar (Figure 2), we consider all sample windows in A_1 as one unit (i.e., the average of the sample windows) instead of looking at every individual sample window within the phase. Merging consecutive windows within a phase assumes that the behavior is stable within a that instance (i.e., all windows have similar behavior). This is usually true and does not significantly affect the accuracy of the method. However, compared to the window-based method, it is dramatically faster. For example, modeling astar running together with bwaves we reduce the number of times the cache sharing model is used from 13 000 to 520, which leads to 25x speedup over the window-based method.

3.2. Phase: Reusing Cache-Sharing Results

The performance can be further improved by merging the data for all instances of a phase. For example, when considering astar (Figure 2), we consider all phase instances of A (i.e., A_1 + A_2) as one unit. This makes the assumption that all instances of the same phase have similar behavior in an execution. This is not necessarily true for all applications (e.g., same function but different input data), but works well in practice.

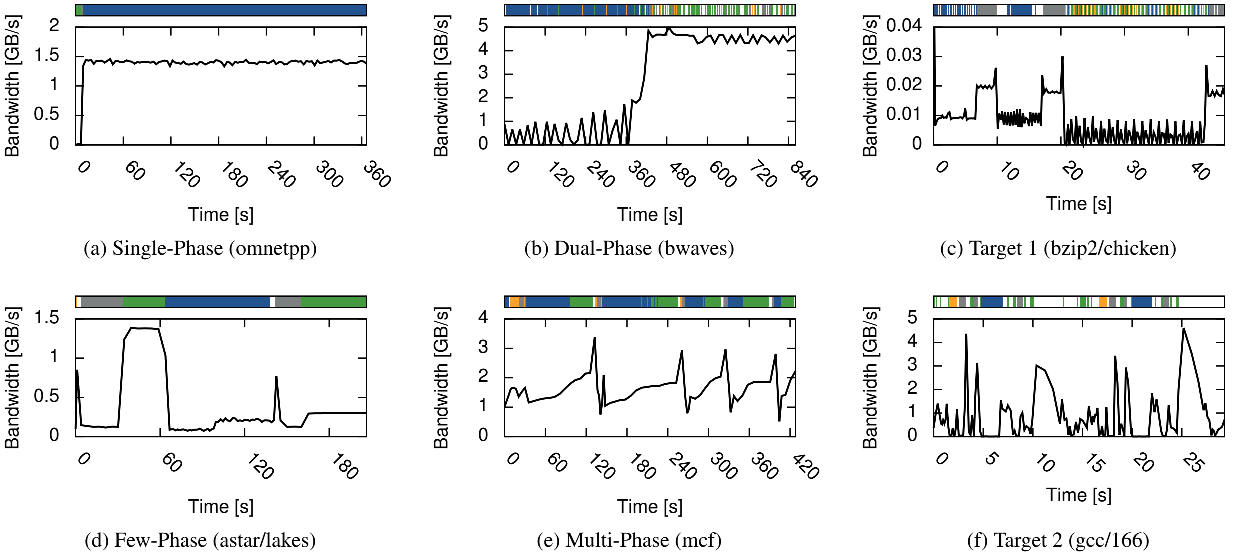
Looking at whole phases does not change the number of times we need to determine an applications cache sharing. It does however enables us to reuse cache sharing results for co-executing phases that reappear later [25]. For example, when astar's phase A_1 co-executes with bwave's phase B, we can save the cache sharing result, and later reuse the result if the second instance (A_2) co-executes with bwaves B.

In the example with astar and bwaves, we can reuse the results from previous cache sharing solutions 380 times. We therefore only need to run the cache sharing model 140 times. The performance of the phase-based method is highly dependent on an application's phase behavior, but it normally leads to a speed-up of 2–10x over the dynamic-window method.

The main benefit of the phase-based method is when determining performance variability of a mix. In this case, the same mix is run several times with slightly different offsets in starting times. The same co-executing phases will usually reappear in different runs. For example, when modeling 100 different runs of astar and bwaves, we need to evaluate 1 400 000 co-executing windows, but with the phase-based method we only need to run the model 939 times.

In addition to reducing the number of model invocations, using phases reduces the amount of data needed to run the model. Instead of storing a profile per sample window, all sample windows in one phase can be merged. This typically

[1] The cache sharing model is implemented in Python and takes approximately 88 ms per invocation on our reference system (see Section 4.1).



(a) Single-Phase (omnetpp)

(b) Dual-Phase (bwaves)

(c) Target 1 (bzip2/chicken)

(d) Few-Phase (astar/lakes)

(e) Multi-Phase (mcf)

(f) Target 2 (gcc/166)

Figure 3. Bandwidth usage across the whole execution of our six benchmark applications, including the four interference applications. Detected phases are shown above. The Single-Phase, Dual-Phase, Few-Phase, and Multi-Phase behavior is clearly visible for the interference applications.

leads to a 100–1000x size reduction in input data. For example, bwaves, which is a long running benchmark with a large profile, reduces its profile size from 57 MB to 82 kB.

4. Evaluation

To evaluate our method we compare the overhead and the accuracy against results measured on real hardware. We ran each *target* application together with an *interference* application and measured the behavior of the target application. In order to measure the performance variability, we started the applications with an offset by first starting the interference application and then waiting for it to execute a predefined number of instructions before starting the target. We then restarted the interference application if it terminated before the target.

In order to get an accurate representation of the performance, we ran each experiment (target-interference pair) 100 times with random start offsets for the target. We used the same starting time offsets for both the hardware reference runs and for the modeled runs.

4.1. Experimental Setup

We ran the experiments on a 2.4 GHz Intel Xeon E5620 system (Westmere) with 4 cores and 3×2 GB memory distributed across 3 DDR3 channels. Each core has a private 32 kB L1 data cache and a private 256 kB L2 cache. All four cores share a 12 MB 16-way L3 cache with a pseudo-LRU replacement policy.

The cache sharing model requires information about application fetch rate, access rate and hit rate as a function of cache size and time. We measured cache-size dependent data using cache pirating in 16 steps of 768 kB (the equivalent of one way) up to 12 MB, and used a sample window size of 100 million instructions.

4.2. Benchmark Selection

In order to see how time-dependent phase behavior affects cache sharing and performance, we selected benchmarks from SPEC CPU2006 with interesting phase behavior. In addition to interesting phase behavior, we also wanted to select applications that make significant use of the shared L3 cache. For our evaluation, we selected four interference benchmarks that represent four different phase behaviors: Single-Phase (omnetpp), Dual-Phase (bwaves), Few-Phase (astar/lakes) and Multi-Phase (mcf).

Figure 3 shows the interference applications' bandwidth usage (high bandwidth indicates significant use of the shared L3 cache), and the detected phases. In addition to the interference benchmarks, we selected two more benchmarks, gcc/166 and bzip2/chicken, that we only use as targets. These benchmarks have a lower average bandwidth usage than the interference benchmarks, but they are still sensitive to cache contention. For the evaluation, we ran all combinations of the six applications as targets vs. each of the four interference applications.

	REF	Single-Phase (omnetpp)				Dual-Phase (bwaves)				Few-Phase (astar)				Multi-Phase (mcf)			
	Time	# Model Invocations			Speedup	# Model Invocations			Speedup	# Model Invocations			Speedup	# Model Invocations			Speedup
	ISO	W	D	P	P	W	D	P	P	W	D	P	P	W	D	P	P
astar	5.9h	723k	302	84.0	2.1k×	1.4M	123k	938	88.7×	797k	1.8k	460	506×	575k	6.9k	465	435×
bwaves	24.3h	4.9M	62.0k	174	113×	7.3M	2.4M	2.0k	54.9×	5.2M	249k	1.0k	105×	4.3M	973k	1.1k	98.5×
bzip2	1.3h	242k	3.7k	63.0	103×	383k	475k	711	31.6×	272k	17.0k	373	59.7×	213k	64.0k	414	58.3×
gcc	0.8h	119k	870	140	133×	209k	203k	1.5k	18.9×	139k	4.5k	759	38.4×	101k	16.0k	786	36.6×
mcf	12.3h	1.0M	1.2k	97.0	1.9k×	2.3M	578k	1.1k	86.0×	1.2M	8.6k	518	798×	702k	30.9k	589	594×
omnetpp	10.3h	1.1M	33.0	14.0	18.6k×	2.2M	52.8k	172	110×	1.2M	358	85.0	2.5k×	856k	1.4k	99.0	1.8k×
average	9.2h	1.3M	11.4k	95.3	**695**×	2.3M	634k	1.1k	**54.9**×	1.5M	46.9k	540	**250**×	1.1M	182k	582	**215**×
													global average	1.6M	219K	572	**213**×

Table 1. Performance statistics for 100 runs with different starting time offsets. The number of model invocations for the three methods (W:Window, D:Dynamic-Window, and P:Phase) is shown along with the speedup for running the phase-based model vs. reference executions on the hardware. We discuss the highlighted results in the text. The model-based approach is on average 213× faster than hardware execution.

4.3. Performance: Speedup

Table 1 presents the performance of the three methods, Windows (W), Dynamic-Windows (D) and Phase (P) per interference application. The Model Invocations columns shows the number of times the cache sharing model was invoked. For example, when astar co-executes with bwaves (see the highlighted area), the cache model is invoked $1\,400\,000$, $123\,000$, and 938 times for the window, the dynamic-window and the phase-based method respectively.

The reference column (REF) shows the execution time to run each target in isolation 100 times. For example, on our system, it takes 5.9 hours to run astar 100 times. The speedup column shows the speedup to model 100 co-executed runs with the phase-based method compared to running the target 100 times[2]. For example, it is $88.7\times$ faster to model 100 co-executions of astar with bwaves than to run astar 100 times in isolation.

Single-Phase. As expected, the speedup is greatest for omnetpp since it consists of just one phase. The dynamic-window method can therefore use a single large window for the whole execution. The phase-based method can then easily reuse cache sharing results whenever the target executes more instances of a phase. The geometric mean of the speedup is $695\times$, the highest of the four interference benchmarks.

Dual-Phase. In a similar sense, we should expect a high speedup for bwaves as well. However, bwaves executes much longer than the other interference benchmarks. So, even though the phase-based method reduces the number of times the cache sharing model is used, it has a high overhead from reading through all application profile data. On average the speedup is only $54.9\times$.

Few-Phase and Multi-Phase. The three methods have roughly the same performance for astar and mcf, and fall in between Single-Phase and Dual-Phase in performance. On average the speedup is $250\times$ and $215\times$ for astar and mcf respectively.

It is clear from the table that the phase-based method provides the best performance for all benchmarks, with an average speedup of $213\times$ for all interference benchmarks[3]. Next, we will evaluate the accuracy of three methods to determine if there are any trade-offs associated with the phase-based method.

4.4. Accuracy: Average Slowdown Error

Figure 4 presents the relative error when predicting the *average* slowdown for the three methods. On average, the windows-based method has an error of 0.39% and a maximum error of 2.2% (bzip2 + omnetpp), while the phase-based method has an average error of 0.41% and a maximum of 1.8% (omnetpp + bwaves). We can therefore safely use the much faster phase-based method without sacrificing accuracy. In the rest of this paper, we will therefore only look at the phase-based method.

In addition to the three methods, the figure also includes the error of using the previous phase-oblivious cache sharing model [16] that does not take time-varying phase behavior into consideration. The phase oblivious method has a reasonably good accuracy for omnetpp since it only has one phase. However, the error is noticeably larger for applications with more phase behavior. For example, 6.3% when astar co-executes with bwaves. This indicates that even when considering average slowdowns (i.e., ignoring variability), it is still important to consider the time-varying behavior and how the two applications' phases overlap.

On average, our phase-based method is $2.78\times$ more accurate than previous work, with an average error of 0.41%

[2] The speedup excludes the time to collect the applications profiles with Cache Pirating. That data is collected only once, and is then used in all the application mixes, and hence not included.

[3] Note that the speedup numbers are based on our Python implementation. A C/C++ implementation would most likely result in greater speedups.

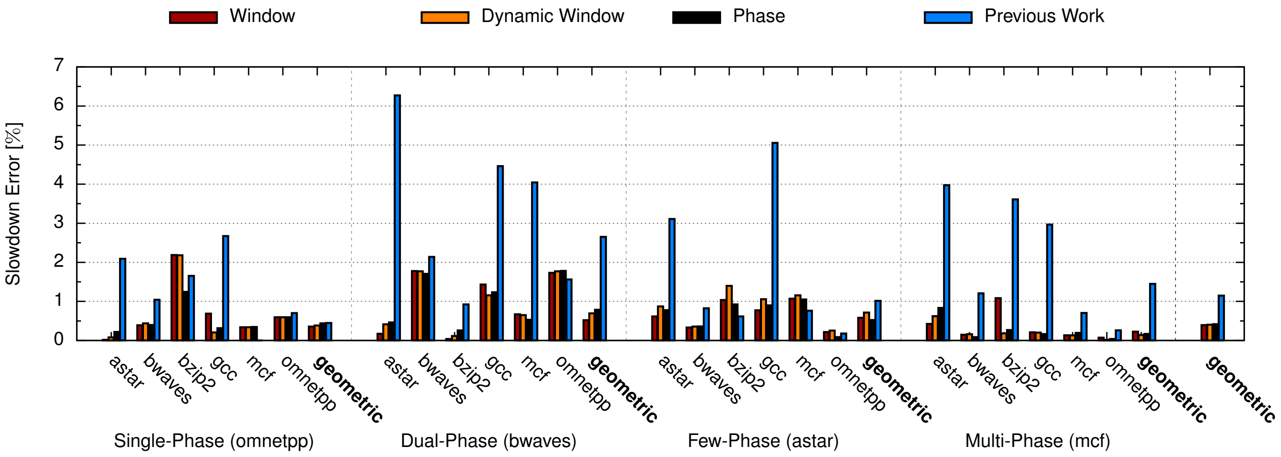


Figure 4. Relative error in predicted slowdown for the three methods and the previous phase-oblivious cache sharing model [16]. ***This shows that the phase-based method (the fastest) can be used without lowering the accuracy. In addition, ignoring applications phase behavior will result in noticeable larger prediction errors (e.g., a 6.3% error for the phase-oblivous method when astar co-executes with bwaves).***

instead of 1.14% and a $3.5\times$ lower maximum error of 1.8% instead of 6.3% (astar + bwaves).

4.5. Performance Variability

The average slowdown is a good metric for evaluating the overall accuracy of the different methods. However, it does not take performance variation into consideration. We therefore use another more descriptive metric, the *cumulative slowdown distributions* (CDF), to display the performance variations. Figure 5 presents the CDF for the phase-based method along with results from the reference hardware runs. The graphs with white backgrounds highlight the benchmark pairs with interesting performance variations.

The cumulative slowdown distributions can be interpreted as showing the probability for a certain maximum slowdown. For example, in Figure 5b, when astar is co-running together with bwaves, it has a 50% probability of having a slowdown less than 6.5%. At the same time, there is a 25% probability that the slowdown is larger than 15%.

Single-Phase. The CDF curves are mostly flat when omnetpp is used as a interference application. For example, Figure 5e, where bwaves is co-running with omnetpp, the curve is basically flat at 1% slowdown. This means that there are no performance variations for bwaves co-running with omnetpp, which is to be expected since omnetpp does not have any time-varying behavior.

Dual-Phase. In contrast to omnetpp, bwaves has two phases with very different behavior. The higher bandwidth usage in the second phase indicate that it uses a larger part of the L3 cache, and will thus impose a larger slowdown on the target application. The effect on the target applications will therefore depend on the starting offset. Since the two phases have roughly the same length, we expect the target's behavior to depend on how it is aligned with the phase change. For example, short targets (e.g., bzip2 in Figure 5j and gcc in Figure 5n), have a sharp turn in the CDF because their execution is not likely to overlap with the phase change. Longer targets (e.g., mcf in Figure 5r), have smoother distribution since they are more likely to overlap with the phase change, causing the part of the application running before the phase change to have a small slowdown, while the parts after the phase-change have a larger slowdown. Since the position of the phase change relative to the target application will change, the CDFs will tend to become smooth.

Few-Phase. There are both flat and curved CDFs for astar as interference application. This is due to differences in the execution lengths (see Figure 3). The CDF in Figure 5g (bwaves) is flat because astar is much shorter than bwaves. Whenever the interference application terminates, it is restarted. This means that astar will be restarted over and over until bwaves terminates. The phase behavior will therefore appear homogeneous from a distance, and it results in a flat CDF. However, shorter targets (e.g., gcc in Figure 5o) will overlap with different phases in astar. We therefore see different target performance between runs and we find a curved CDF.

Multi-Phase. The CDFs for mcf are similar in shape to astar's for mostly the same reasons. However, mcf has a slightly different phase behavior. The same set of phases reappear several times in mcf (see Figure 3e). Since astar

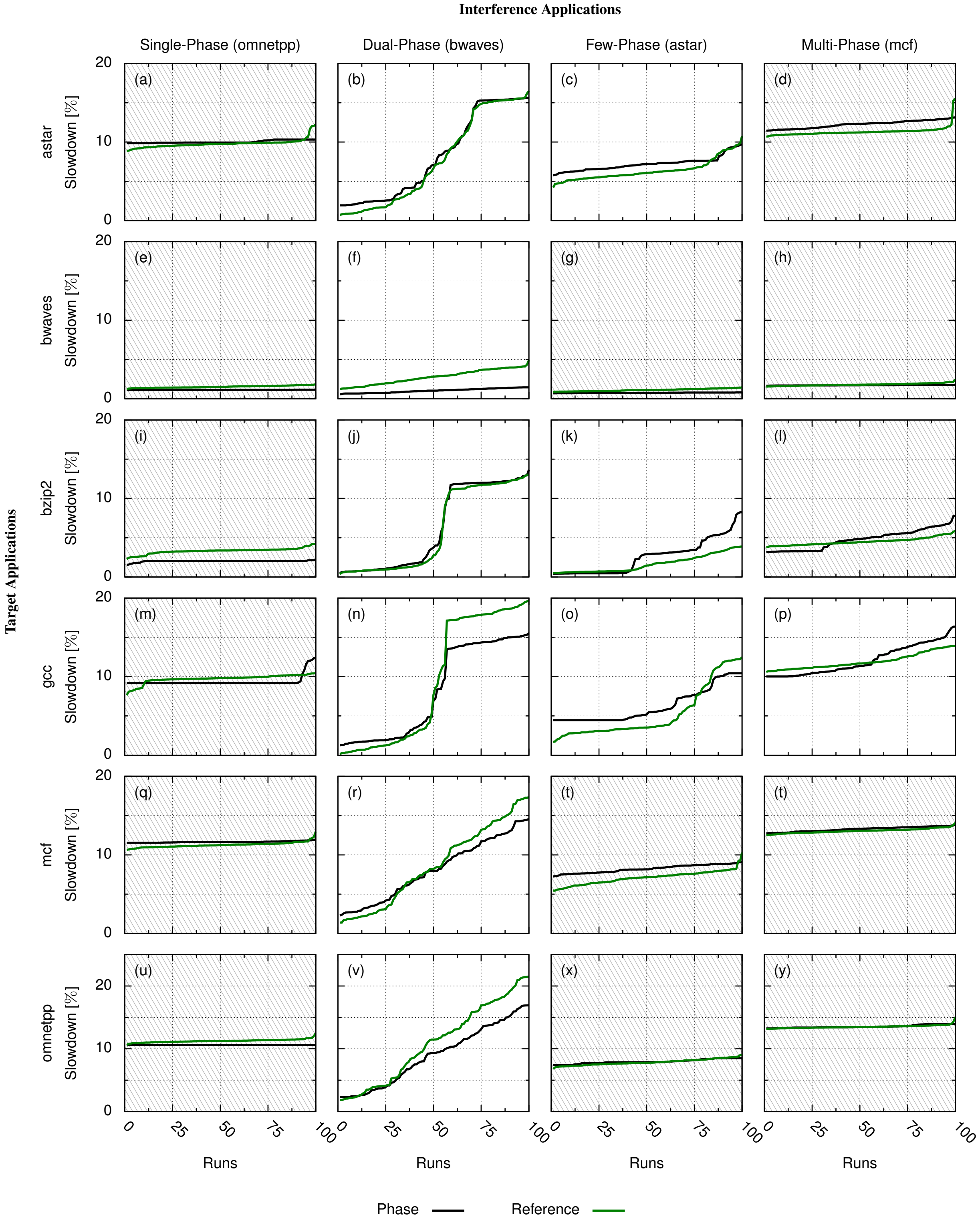


Figure 5. Cumulative distributions of target-slowdowns for 100 runs of each pair of applications with random start time offsets. The 100 application runs were sorted by slowdown, with the largest slowdown on the right. A flat line indicates no performance variation between the slowest and fastest run, and hence no variation. A steep curve indicates significant performance variation across the 100 runs. ***In general, the performance will vary depending on how the phases overlap.***

takes about half the time to execute, its execution will overlap with several of mcf's phases. Changing offsets in starting time will therfore not change astars performance, since astar will just co-execute with the same set of phases but with different instances of the same phases. We therefore see a flatter curve for astar co-running with mcf (Figure 5d) than with astar (Figure 5c).

4.6. Error: Performance Variability

The CDFs produced with the phase-based method have an overall good accuracy, but do not always overlap completely with the reference curves. There are two main sources of error: cache pirating data and bandwidth limitations. We will discuss these two problems in the following sections.

4.6.1. Pirate Data

To measure cache-size dependent data, cache pirating co-executes a cache intensive stress application that tries to steal parts of the cache. This approach has two limitations: First, if the target is also cache intensive, the pirate will have trouble keeping its working set in the cache. Second, when stealing a large portion of the cache, the pirate will have trouble reusing all of its the data before it is evicted.

Figure 5k shows the CDF for bzip2 when co-executing with astar. The problem here is that bzip2 is cache intensive and only uses a small part of the L3 cache compared to the others (see Figure 3c). This makes it hard for Cache Pirating to steal the required cache space. As a consequence, we incorrectly estimated some cache-size dependent effects, which leads to our overestimating the slowdown in the CDF.

One solution would be to instead use more cumbersome and expensive methods to acquire the data. For example, page coloring [10] could be used to limit the amount of cache the target application is allocated.

4.6.2. Bandwidth

The cache sharing model assumes that the system has infinite bandwidth. This is obviously not the case, and as a result the model will underestimate the slowdown whenever the targets need more bandwidth than the system can provide. Figure 5 shows that we tend to underestimate the slowdown of bwaves. The second phase in bwaves (see Figure 3b) consumes more bandwidth than the other applications. If this is a problem, we should expect that we will find the largest errors when modeling bwaves, which is indeed the case.

One feature of the cache sharing model is that it can predict the bandwidth an application mix requires to avoid being bandwidth limited. Figure 6 shows the estimated cumulative bandwidth demand[4] and the measured cumulative bandwidth the application mix received during the fastest (i.e., run 1 in Figure 5) and the slowest (i.e., run 100 in Figure 5) runs. We interpret the figures as follows: x percent of the execution has a bandwidth demand of more than $y\,\mathrm{GB/s}$. For example, during the slowest run with mcf (Figure 6r), 50% of mcf's execution needs more than $7.5\,\mathrm{GB/s}$ to avoid slowing down due to bandwidth limitations.

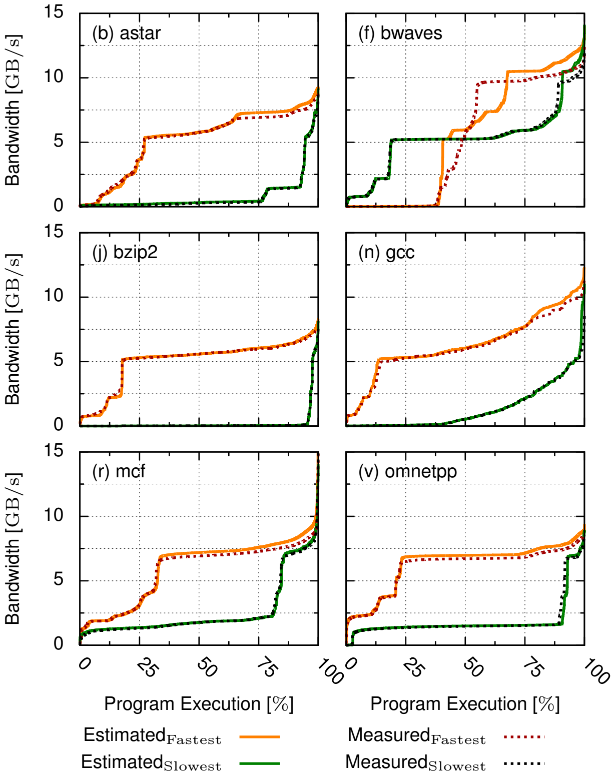


Figure 6. Predicted cumulative bandwidth demand (Estimated) and measured cumulative bandwidth usage (Measured) for the fastest and the slowest run when co-executing with bwaves.

The bandwidth demand is lowest for the fastest run since the target applications is co-running with the first phase in bwaves. Here, the estimated bandwidth demand and the measured bandwidth usage closely match each other. This means that the system can provide the required bandwidth. But also, since we accurately estimate the slowdown, this also implies that the method can accurately estimate the bandwidth demand.

The slowest runs occur when the targets are co-running

[4] The model produces bandwidth estimates using the input profile to estimate the application's bandwidth consumption for a given cache allocation.

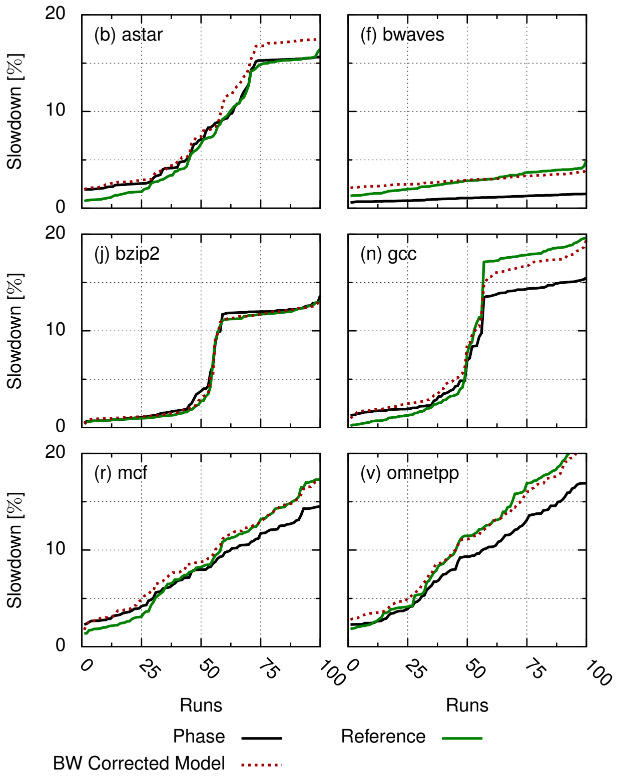


Figure 7. Cumulative slowdown distributions for 100 runs (as in Figure 5) with the bandwidth corrected model. ***This shows that the accuracy can be improved by combining a bandwidth model with our cache sharing model to handle both cache sharing and bandwidth.***

with the second phase in bwaves. Here the bandwidth demand is much higher, and sometimes the estimated bandwidth demand is higher than the measured bandwidth received. This means that the target is slowing down due to bandwidth limitations. To see if we can correct the slowdown estimations by taking this into consideration, we use the measured bandwidth the application mix receives from the reference hardware runs. To do this, we update the estimated number of executed cycles (c_{est}) with the following formula:

$$c_{new\_est} = c_{est} + \frac{(BW_{est} - BW_m) * c_{est}}{BW_{MAX}}$$

where c_{new_est} is the new estimate, c_{est} is the old estimate, BW_{est} is the estimated bandwidth demand, BW_m is the measured bandwidth received and finally, BW_{MAX} is the maximum bandwidth our system can provide[5]. In other words, we extend the modeled execution time by the number of additional cycles incurred by bandwidth limitations.

Figure 7 shows the result of estimating the slowdown with the bandwidth corrected model. This correction reduces the slowdown error for bwaves, mcf, and omnetpp. However, we still underestimate the slowdown slightly for gcc, and now overestimate the slowdown for astar.

Unfortunately, such a bandwidth correction will not work in practice since it uses oracle information (i.e., BW_m), but it illustrates that a better slowdown estimate can be obtained by combining the cache sharing model with a bandwidth model to model both cache sharing and bandwidth limitations. This is a promising direction for future work.

5. Case Study – Modeling Multi-Cores

In the previous section we investigated performance variations of application pairs. However, modern processors have more than two cores. In this section, we perform a small case study to demonstrate that our method can be used to model larger application mixes, and to model system throughput.

Since all of the techniques we integrate in this method scale beyond two cores, we demonstrate that our method can scale as well by estimating the system throughput when co-running a mix of four applications on our four core reference system. To do this we compare the estimated behavior (IPC and bandwidth) to that of the actual behavior for a max of four applications. Figure 8 shows the IPC and system throughput over time for the first ten seconds when co-running gcc, bzip2, astar and bwaves. The figure shows that the estimated IPCs matches the reference well.

The two main sources of error, pirate data and bandwidth, will become more problematic when modeling larger application mixes. The amount of cache available to each application is reduced when adding more programs to the mix, which puts more pressure on the cache pirate to collect data for smaller cache allocations.

The bandwidth limitation will also become more noticeable for two reasons: First, more applications will contend for bandwidth, and thus lower the amount available to each application. Second, when an application receives less cache space, its bandwidth usage increases since it misses more in L3 and that data needs to be fetch from memory again.

6. Related Work

Techniques to explore and understand multicore performance can generally be divided into three different categories; full system simulation, partial simulation/modeling,

[5] We estimated the real-world bandwidth limit of our reference system to approximately 12 GB/s using the STREAM benchmark [13].

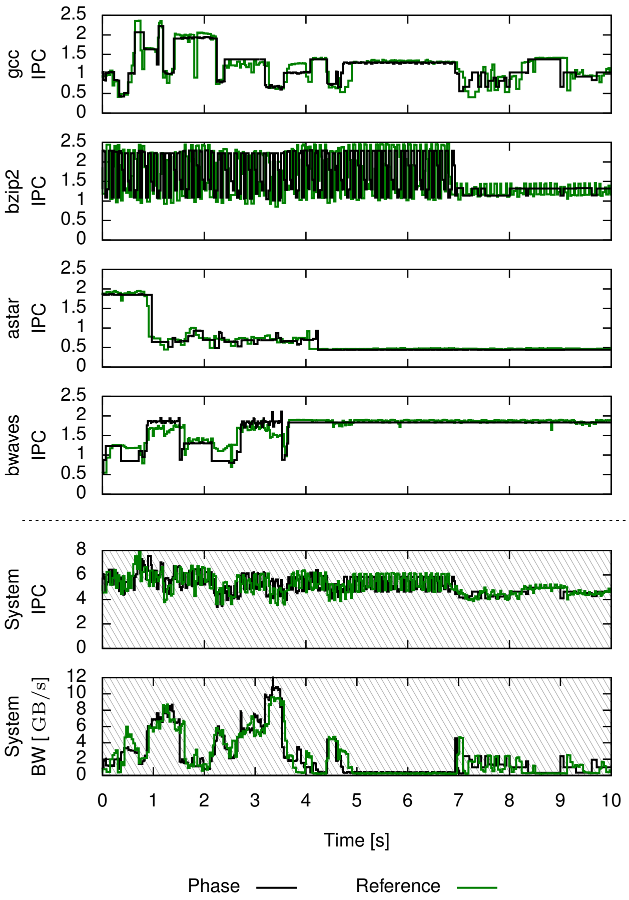


Figure 8. Predicted IPC (Phase) and measured IPC (Reference) for four co-running applications over time on a four-core system, as well as predicted and reference aggregate throughput (IPC) and bandwidth.

and higher level modeling. The most expensive but also the most detail approach is full system simulation [25, 1, 24] where all cores and the entire memory system are simulated. A faster, but less detailed, approach is to only simulate/model parts of the system, and in particular the memory system. Such methods are either trace driven [2, 4, 3, 27] or use high-level data [29, 16] similar to the data we use. Finally, the least detailed approach simply aims to identify which applications are sensitive to resource contention [28, 17, 11].

Simulation normally requires combinations of applications to be simulated together, which leads to poor scaling. Van Craeynest and Eeckhout [26] combine simulation and memory system modeling to reduce the cost of simulating co-scheduled applications. Instead of simulating how applications contend for shared resources, they simulate applications running in isolation and use the output from the simulator to drive a cache sharing model. A major difference between our methods is that they depend on a single high-fidelity simulation to generate the application profiles used by their model, whereas we measure our input data with a relatively low overhead on the target system. Also, accurately simulating commodity hardware is often hard, or even impossible, since manufacturers seldom release enough information to implement a cycle-accurate simulator. Additionally, their evaluation focuses on the performance variations of the underlying hardware due to different application mixes, whereas we focus on the performance variations of the individual applications.

The method most similar to ours is the phase guided simulation methods by Van Biesbrouck et al. [25, 24]. Similar to our phase-based method, they use phase information to reuse simulation results. However, since their method relies on simulation they need to find and simulate representative regions (i.e., sample windows) of co-running phases. We do not have this problem since we can use the average behavior for the entire phase in our profiles.

7. Conclusions

In this paper, we have presented an analytical method that predicts performance variability due to the cache sharing effects imposed by other co-running applications. The per-application profile data the method requires can be captured cheaply and accurately during native execution on real hardware for each application in isolation. Three alternative cache-sharing methods with different performance properties were compared. We showed that the fastest method provides excellent accuracy. We have analyzed the performance variations caused by bandwidth sharing and showed that even a simple bandwidth sharing model could explain most of the deviations observed when the bandwidth contention is high. In future work, we plan on extending our analytical method to include such bandwidth-sharing effects.

Due to its speed, simple input data, and accuracy, this method can be used to build efficient tools for software developers or system designers, and is fast enough to be leveraged in scheduling and operating system designs.

References

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems,. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, 2006.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-thread Cache Contention on a Chip Multi-Processor Architecture. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[3] X. Chen and T. Aamodt. Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors. *IEEE Transactions on Computers*, 61(7):913–927, 2012.

[4] X. E. Chen and T. M. Aamodt. A First-Order Fine-Grained Multithreaded Throughput Model. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[5] A. S. Dhodapkar and J. E. Smith. Comparing Program Phase Detection Techniques. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[6] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*, chapter 10.11. On-line Clustering, pages 559–565. Wiley-Interscience, 2 edition, 2001.

[7] D. Eklöv, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache Pirating: Measuring the Curse of the Shared Cache. In *Proc. International Conference on Parallel Processing (ICPP)*, 2011.

[8] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, J. Simon Steely, and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[9] J. Lau, S. Schoemackers, and B. Calder. Structures for Phase Classification. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2004.

[10] 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 *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[11] J. Mars, L. Tang, and M. L. Soffa. Directly Characterizing Cross Core Interference Through Contention Synthesis. In *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*, 2011.

[12] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention Aware Execution. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2010.

[13] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Tech. rep., University of Virginia, 1991–2007. http://www.cs.virginia.edu/stream/.

[14] N. Peleg and B. Mendelson. Detecting Change in Program Behavior for Adaptive Optimization. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[15] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. Annual International Symposium on Microarchitecture (MICRO)*, 2006.

[16] A. Sandberg, D. Black-Schaffer, and E. Hagersten. Efficient Techniques for Predicting Cache Sharing and Throughput. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[17] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing Cache Pollution Through Detection and Elimination of Non-Temporal Memory Accesses. In *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[18] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. In *Proc. International Symposium on Workload Characterization (IISWC)*, 2012.

[19] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *Proc. International Symposium on Workload Characterization (IISWC)*, 2011.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[22] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *Proc. International Symposium on Computer Architecture (ISCA)*, 2003.

[23] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing Shared L2 Caches on Multicore Systems in Software. In *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[24] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Considering All Starting Points for Simultaneous Multithreading Simulation. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2006.

[25] M. Van Biesbrouck, T. Sherwood, and B. Calder. A Co-Phase Matrix to Guide Simultaneous Multithreading Simulation. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2004.

[26] K. Van Craeynest and L. Eeckhout. The Multi-Program Performance Model: Debunking Current Practice in Multi-Core Simulation. In *Proc. International Symposium on Workload Characterization (IISWC)*, 2011.

[27] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi. All-Window Profiling and Composable Models of Cache Sharing. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

[28] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proc. Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, 2008.

[29] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache Contention and Application Performance Prediction for Multi-Core Systems. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.