Efficient Techniques for Predicting Cache Sharing and Throughput

Andreas Sandberg
Uppsala University
Sweden
andreas.sandberg@it.uu.se

David Black-Schaffer
Uppsala University
Sweden
david.black-schaffer@it.uu.se

Erik Hagersten
Uppsala University
Sweden
erik.hagersten@it.uu.se

ABSTRACT

This work addresses the modeling of shared cache contention in multicore systems and its impact on throughput and bandwidth. We develop two simple and fast cache sharing models for accurately predicting shared cache allocations for random and LRU caches.

To accomplish this we use low-overhead input data that captures the behavior of applications running on real hardware as a function of their shared cache allocation. This data enables us to determine how much and how aggressively data is reused by an application depending on how much shared cache it receives. From this we can model how applications compete for cache space, their aggregate performance (throughput)¸ and bandwidth.

We evaluate our models for two- and four-application workloads in simulation and on modern hardware. On a four-core machine, we demonstrate an average relative fetch ratio error of 6.7% for groups of four applications. We are able to predict workload bandwidth with an average relative error of less than 5.2% and throughput with an average error of less than 1.8%. The model can predict cache size with an average error of 1.3% compared to simulation.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling Techniques

General Terms

Measurement, Performance

Keywords

Cache Sharing, Modeling, Performance

1. INTRODUCTION

The shared cache in contemporary multicore processors has been repeatedly shown to be a critical resource for application performance [13, 15, 18, 8, 14]. This has motivated a significant amount of research into modeling the impact of cache sharing with the goal of understanding applications' interactions through the shared cache and for providing insight to schedulers and runtime systems [15, 20, 11, 10].

This work presents two models for predicting cache allocations, bandwidth requirements, and performance of application mixes in the presence of a shared last-level cache. The models are developed for random replacement and LRU caches, but are shown to be accurate for the pseudo-LRU caches of modern Intel processors.

These models take into account the complexities of modern hardware (such as out-of-order execution and hardware prefetchers) by leveraging input data that incorporates the applications' behavior on real hardware. This input data consist of the applications' fetch and hit rates, IPCs, and hit ratios as a function of their cache allocation, and can be acquired with low overhead on modern multicore machines [6]. This low-overhead data is in contrast to many existing methods for modeling cache sharing which rely on expensive data such as stack distance traces [2, 4, 3, 17].

To model cache sharing we use an application's fetch and hit rates as a function of cache size to determine how much of its data is reused for a given cache allocation, and how often that data is reused. With this information we can model how multiple applications compete for shared cache space. The model then uses a numerical solver to find a stable solution that determines the final cache allocations. Once we know the cache allocations we can use our input data to predict performance (throughput) and bandwidth requirements for mixes of co-scheduled applications.

This ability to model cache sharing and predict its impact on performance and bandwidth is important for scheduling and performance analysis on complex systems. Such modeling forms the basis for resource-aware placement in modern memory hierarchies, scheduling on heterogeneous architectures, and for making runtime decisions on future chips in the presence of power constraints (e.g., dark silicon). By combining low-overhead data that reflects the complexities of the real hardware and a simple sharing model we are able to quickly and accurately predict sharing and performance, which is essential for such goals.

This paper makes the following contributions:

- We present a statistical cache-sharing model for random caches that uses high-level, low-overhead input data.

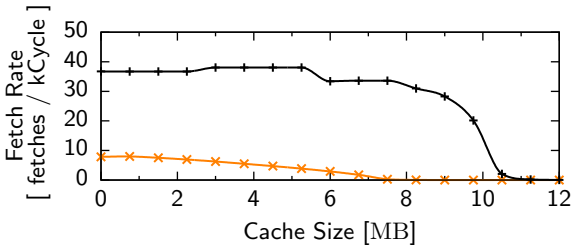


Figure 1: Example Cache Pirate data showing fetch rate as a function of cache size for two applications.

- We extend the model to LRU caches by deriving aggregate data reuse information from the input data, and using this to model competition for cache space based on the data reuse frequencies of each application.
- We demonstrate the accuracy of the model for predicting cache sharing and fetch ratios for mixes of two and four co-scheduled applications through simulation and on real hardware.
- We demonstrate the ability to accurately predict performance (IPC) and bandwidth requirement of application mixes on real hardware.

2. MODELING CACHE SHARING

Consider two applications sharing a cache. Their behavior with respect to the shared cache can intuitively be thought of as two flows of liquid filling an overflowing glass. The two in-flows correspond to fetches into the cache and the liquid pouring out of the glass corresponds to the replacement stream from the cache. If the in-flows are constant, the system will eventually reach a steady state. At steady state, the concentrations of the liquids in the glass are constant and proportional to their relative inflow rates. Furthermore, the outflow rates of the liquids are proportional to their concentrations. This very simple analogy describes the behavior of random caches.

Describing LRU caches requires data reuse to be considered since data reused *frequently enough* will stick in the cache and avoid replacement. In the liquid analogy above, data reused frequently enough can be thought of as ice cubes that never leave the glass. The threshold for how frequently data needs to be reused to exhibit "sticky" behavior varies between sharing situations.

2.1 Low-Overhead Input Data

The goal of our cache sharing models is to find the amount of cache allocated to each application in a mix of co-scheduled applications at steady state. This requires per-application information about fetch[1] rate and data reuse characteristics for all applications as well as how that information is affected by cache contention. In order for this information to accurately describe the target system, it needs to take into consideration effects from complex dynamic hardware, such as super-scalar out-of-order execution and hardware prefetching.

[1]The term *fetch* is used extensively in this paper to describe a movement of data from memory to cache caused by either a cache miss or prefetching activity.

Cache Pirating [6] is a method to capture our required input data on the target hardware. In Cache Pirating, the studied application is co-scheduled to share a cache with a small cache-intensive micro benchmark, the Pirate. The Pirate is designed to steal only cache, leaving other shared resources untouched. In a single run, the amount of cache the Pirate steals is varied, while the effects on the studied application are measured using hardware performance counters. The application's miss rate, fetch rate, hit rate, miss ratio, fetch ratio, hit ratio, memory bandwidth, CPI, etc., *as a function of cache size*, can be measured with an average overhead of 5.5%.

Cache Pirating typically measures sensitivity by stealing a whole way at a time, while our models assume continuous data. We therefore interpolate the measured data using monotone cubic splines [7]. We chose this interpolation method over linear interpolation because the resulting function estimates the behavior in applications with sharp steps in their fetch rate curves (e.g., applications with a small fixed data set) more accurately by making the edges sharper. Figure 1 shows an example of data produced using Cache Pirating: Fetch rate as a function of cache size for two applications, measured in 16 discreet steps and interpolated with monotone cubic splines.

The following section describes how the fetch rate and hit rate information measured using Cache Pirating is used to determine the amount of cache allocated to each application in a sharing situation. Knowing the amount of cache allocated to each application allows us to predict additional performance metrics. In Section 4 we show how throughput and bandwidth demand can be predicted using knowledge about cache allocations and the data from Cache Pirating.

2.2 Modeling Random Caches

In random caches, sharing only depends on two events: fetches into the cache and replacements. At steady state, the amount of data an application installs into the cache is equal to the amount of data evicted from that application's cache allocation; that is, the application's fetch rate (fetches per cycle) must equal its replacement rate (replacements per cycle). An application's replacement rate is proportional to the total fetch rate, F, into the cache and its replacement probability. Since replacements are random, the replacement probability is proportional to the amount of cache allocated to the application. For a shared cache of size C and an application, n; the application's fetch rate, f_n, and cache allocation, c_n, are related according to:

$$\begin{cases} \mathrm{f}_n = \mathrm{F}\,\dfrac{\mathrm{c}_n}{\mathrm{C}} \\ \mathrm{C} = \displaystyle\sum_i \mathrm{c}_i \end{cases} \tag{1}$$

We can solve the equation system above, given that we have each application's fetch rate as a function of cache size, using readily available equation system solvers.

2.3 Modeling LRU Caches

LRU caches, unlike random caches, use access history to replace the item that has been unused for the longest time. We refer to the duration of time a cache line has been unused as its *age*. Whenever there is a replacement decision, the oldest cache line is replaced.

In practice, we can not determine the age of individual

cache lines based on our input data. Instead, we look at *groups* of cache lines with the same maximum *age*[2] and let the groups from different applications compete for cache space.

Cache lines that are fetched into the cache but are evicted before they are reused are put in a separate group and handled differently. After the initial fetch into the cache, the age of these cache lines increases whenever *any* application fetches new data into the cache. This allows us to treat them as one group common to all applications, with one common age. We will refer to these cache lines as *volatile* since they are evicted from the cache before they are reused.

Cache lines which are reused before they are evicted from the cache are referred to as *sticky* cache lines since their reuse makes them resilient to eviction. Normally an application reuses more data in the cache when the amount of cache the application has access to grows. This means that some data is *potentially* sticky and only becomes sticky when the application has access to enough cache. The amount of sticky and volatile data in the cache therefore depends on how much cache an application has access to, which is a function of what other applications are co-executing.

To explain the LRU model, we will first describe how to model sharing within the volatile group and how the age of the group is determined. We will then describe how sticky groups are modeled, and finally how our solver uses this information to determine cache sharing.

2.3.1 Modeling Volatile Data

When applications do not reuse their data before it is evicted from the cache, LRU caches degenerate into FIFO queues with data moving from the MRU position to the LRU position before being evicted. Similar to random replacement, the amount of cache allocated to an application will be proportional to its fetch rate. This observation allows us to use the method devised for random replacement to model cache sharing for volatile data. Assuming that we know the amount of cache available to volatile data[3], $\mathrm{C^v}$, we can solve Equation 1 for the volatile part of the cache. This allows us to estimate the amount of volatile data, $\mathrm{c^v}$, each application has.

Since sticky data and volatile data from different applications compete for cache space, we need to be able to compare their maximum age. Because the cache degenerates into a FIFO queue for volatile data, the maximum age of volatile elements can be determined using Little's law [9].[4] Assuming that we know the total size of the volatile part of the cache, $\mathrm{C^v}$, and the *total* fetch rate into the cache, the maximum age of all volatile cache lines, $\mathrm{A^v}$, is:

$$\mathrm{A^v} = \frac{\mathrm{C^v}}{\mathrm{F}} \tag{2}$$

[2]At any given moment, the cache lines in a group will have different ages. It therefore only makes sense to talk about the maximum age of a group.

[3]We can estimate the amount of sticky data each application has, and therefore whatever is left of the cache is used for volatile data.

[4]Little's law sets up a relationship between the number of elements in a queue (size), the time spent in the queue (maximum age) and the arrival rate (fetch rate). The total arrival rate into the queue is the sum of all fetch rates, F, in the system.

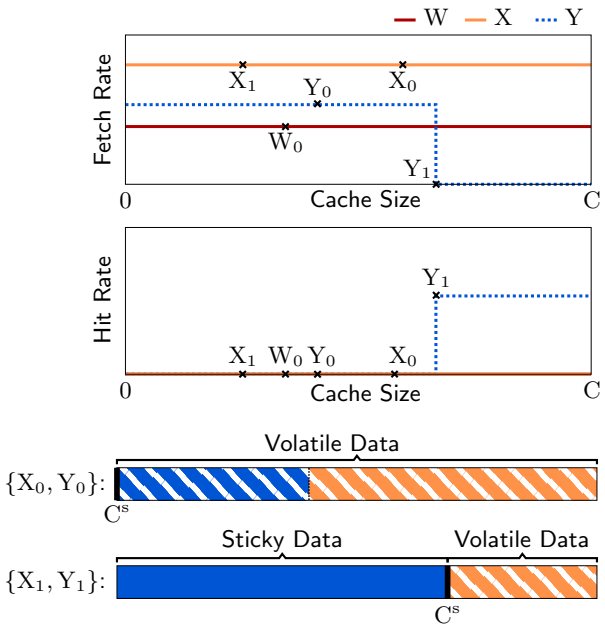


Figure 2: Fetch and hit rate curves for three sample applications. Application W and X always miss in the shared cache, while Y misses only when it has less than $\mathrm{c}_n(\mathrm{Y}_1)$ space in the cache. The cache allocations for the two stable cache sharing configurations of X and Y are shown below.

Example 1: Consider application X and application W in Figure 2. Both of the applications have fetch rates that are independent of cache size and X has twice the fetch rate of W. Since the hit rate is zero for both of them, neither reuses any data in the cache (i.e., all data is volatile). Using the random replacement model for the volatile data, we conclude that X gets twice the cache allocation of W (i.e., X uses two thirds of the cache) causing the applications to stabilize at the solution $\{\mathrm{X}_0, \mathrm{W}_0\}$. Since the entire cache is filled with volatile data, the maximum age of volatile elements is described by:

$$\mathrm{A^v} = \frac{\mathrm{C^v}}{\mathrm{F}} = \frac{\mathrm{C}}{\mathrm{f(X_0) + f(W_0)}}$$

□

2.3.2 Modeling Sticky Data

In most cases, there is both sticky and volatile data in the cache at the same time. Unlike volatile data, sticky data stays in the cache because it is reused while it is in the cache. When sticky and volatile cache lines compete for cache space, the decision to let a sticky cache line remain sticky depends on its age. A sticky cache line becomes volatile if it is older than the oldest volatile cache line. In our model, we make this decision for entire groups of cache lines with the same maximum age. A group of sticky cache lines with the same maximum age, $\mathrm{a^s}$, is allowed to stay in the cache as sticky cache lines if it is younger than the oldest volatile cache line:

$$\mathrm{a^s} < \mathrm{A^v} \tag{3}$$

Similar to volatile data, we can estimate the maximum

age for a group of sticky data using Little's law if we know the size of the group and its reuse rate. This can best be illustrated with an example:

Example 2: Application Y in Figure 2 does not reuse any data (its hit rate is zero) when it has access to less cache than $c(Y_1)$. However, it reuses all its data when it has access to more cache (its fetch rate is zero). This means that it has one group of potentially sticky data. The size of the group is $c(Y_1)$ and the aggregate reuse rate of all elements in the group is its hit rate, $h(Y_1)$.

When Y starts it will bring its entire data set into the cache and start reusing it, causing the data to become sticky. If X is then started, it will first install data into the unused part of the cache. The size of Y's sticky data set will at this point be $c(Y_1)$ and the rest of the cache will be filled with data belonging to X. Since X does not reuse its data, all its data will be volatile. We obtain the ages of sticky, a^s, and volatile, A^v, elements when they start to compete for cache as follows:

$$\begin{aligned} a^s &= \frac{c(Y_1)}{h(Y_1)} \\ A^v &= \frac{C^v}{F} = \frac{C - c(Y_1)}{f(X_1)} \end{aligned}$$

$\{X_1, Y_1\}$ is a stable cache sharing configuration if Y's sticky cache lines are younger than X's volatile cache lines. This means that Y is reusing its data set frequently enough to prevent X from pushing it out of the cache.

An interesting feature of this benchmark combination is that it can have two stable cache sharing configurations. If X starts first and is allowed to fill the cache with its volatile data, when Y starts, it will have to compete with X to bring its data into the cache. At this point, the entire cache consists of volatile data since Y has not installed enough of its data to be able to reuse it before it is evicted from the cache. Since X has a higher fetch rate than Y, it fetches data faster and will therefore get more cache than Y. In this case, Y will never fit its entire group of potentially sticky data, and its data it will instead remain volatile. Both $\{X_0, Y_0\}$ and $\{X_1, Y_1\}$ are therefore valid sharing configurations, depending on the starting order. □

In the examples so far, both of the benchmarks in a pair have either had sticky or volatile data, but not both. Real applications typically have both sticky and volatile data in the cache at the same time:

Example 3: Application Z in Figure 3 has two drops in its fetch rate curve, which means that it has two groups of potentially sticky data. This can happen in applications reusing two arrays of different size. For a cache size of $c(Z_0)$, the application is able to fit its first group of data in the cache (the fetch rate drops just before $c(Z_0)$) and that group becomes sticky. If the application has access to more cache than $c(Z_1)$, its fetch rate drops to zero and all of its data becomes sticky. In order to calculate the age, a^s, of a group of sticky data, we need to know how much that group contributes to the total hit rate and how big the group is. Assuming that we know the amount of sticky data in an application, c^s, as a function of cache size (we will show how

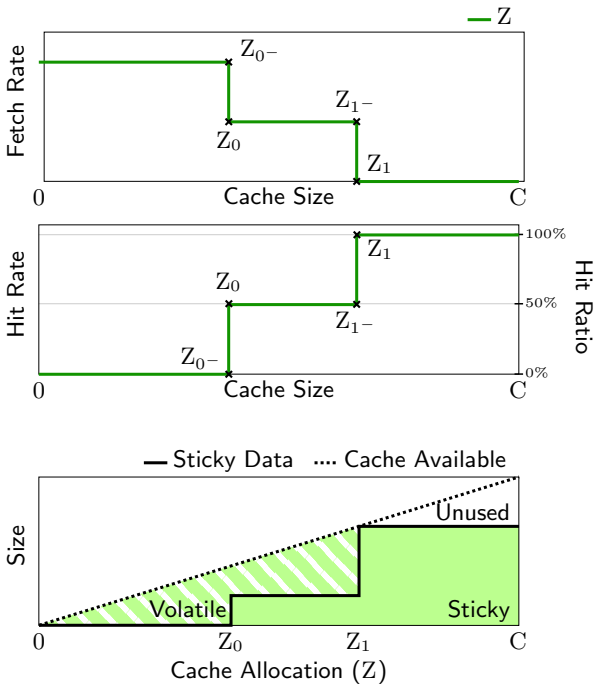


Figure 3: Z has two groups of sticky data of different sizes. When a group of sticky data starts to fit in the cache, the fetch rate starts to drop and the hit ratio increases. Whenever the hit ratio increases, the amount of volatile data at that point decreases (it becomes sticky) by the same relative amount.

to estimate this in Section 2.3.3), we can calculate $a^s(Z_0)$ as:

$$a^s(Z_0) = \frac{c^s(Z_0) - c^s(Z_{0-})}{h(Z_0) - h(Z_{0-})}$$

□

The age derived in the example above is simply a finite difference approximation of a differential equation. In general, the access rate for sticky elements is defined as:

$$a^s(c) = \frac{dc^s}{dh} \tag{4}$$

Equation 4 is actually a simplification that assumes that an application's execution rate does not change with cache size. However, execution rate generally increases as an application gets access to more cache. This variation is accurately captured in our Cache Pirate input data. Not compensating for the change in execution rate leads an erroneous estimate of a block's contribution to the total hit rate. We address this by using the difference in *hit ratio* (hits per memory access) which is execution rate independent. We then scale the difference in *hit ratio* with the application's *accesses rate* (which is execution rate dependent) to get the block's contribution to the total hit rate.

2.3.3 Estimating Sticky Group Sizes

The amount of sticky data can be estimated from how an application's hit ratio changes with its cache allocation. The

relative change in hit ratio is proportional to the relative change in the sticky data.

Example 4: As seen in Figure 3, Z's hit ratio increases in two steps. This means that it has two groups of potentially sticky data. When it has access to less cache than $c(Z_0)$, the hit ratio is zero and it has no sticky data. The amount of sticky data can be broken down into three different cases based on the cache allocation c:

$0 \leq c < c(Z_0)$
: Since the hit ratio is zero, there is no sticky data.

$c(Z_0) \leq c < c(Z_1)$
: When the amount of cache is increased to $c(Z_0)$, the hit ratio increases from 0% to 50% (i.e., 50% of the fetches become hits). We would therefore expect 50% of the volatile data to become sticky. Since the amount of volatile data just before Z_0 is $c(Z_0)$, the amount of sticky data in this range is $\frac{1}{2}c(Z_0)$.

$c(Z_1) \leq c$
: At $c(Z_1)$, all of the fetches become hits. The sticky data set size is therefore $c(Z_1)$.

Using Z's hit ratio function, $\hat{h}$, the reasoning above can be generalized into:

$$\frac{c^s(Z_x) - c^s(Z_{x-})}{c(Z_x) - c^s(Z_{x-})} = \frac{\hat{h}(Z_x) - \hat{h}(Z_{x-})}{1 - \hat{h}(Z_{x-})}$$

□

The difference approximation above can be generalized into the following differential equation:

$$\frac{dc^s}{dc}\frac{1}{c - c^s} = \frac{d\hat{h}}{dc}\frac{1}{1 - \hat{h}} \tag{5}$$

2.3.4 *Putting It All Together*

We have described how an LRU cache can be modeled by splitting it into groups of cache lines with the same maximum age. Volatile data is treated as a separate group of data where the maximum age is determined by the *total* fetch rate into the cache. Each application's sticky data is allowed to stay in the cache as sticky data as long as its maximum age is lower than the maximum age of any application's volatile data, otherwise it becomes volatile. This leads to the following requirement, which must hold for every application, n, at steady state:

$$a_n^s < A^v \tag{6}$$

If the requirement does not hold for an application, its sticky data that is too old to remain sticky becomes volatile.

Volatile data in the cache can be modeled using the model derived for random replacement (Equation 1), that is:

$$f_n = F\,\frac{c_n^v}{C^v} \tag{7}$$

The total amount of cache an application has access to is the sum of its sticky and volatile cache allocations:

$$c_n = c_n^s + c_n^v \tag{8}$$

Using the requirements defined above, we can find a sharing solution using a numerical solver. The solver starts with an initial guess, wherein the application that starts first has access to the entire cache[5] and the other applications do not have access to any cache. The initial guess corresponds to the state of the cache just before a new application is started, which enables us to find the correct solution if the application mix has multiple solutions.

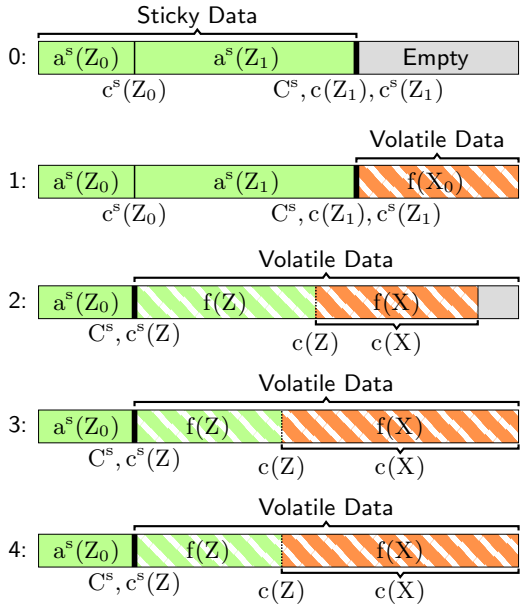


Figure 4: Solver steps for determining cache sharing when running example applications X and Z from Figure 2 and Figure 3 together. (See Example 5.)

The solver then lets all applications compete for cache space by enforcing the age requirement between sticky and volatile cache lines. If the age requirement can not be satisfied for an application, the solver shrinks that application's cache allocation until the remaining sticky data satisfies the age requirement. If multiple applications fail to satisfy the age requirement, we shrink the application with the oldest cache lines. The cache freed by the shrinking operation is then distributed among *all* applications by solving the sharing equations for the volatile part of the cache.

The process of shrinking and growing the amount of cache allocated to the applications is repeated until the solution stabilizes (i.e., no application changes its cache allocation significantly).

Example 5: Assume that we run application Z from Figure 3 and X from Figure 2. Z starts first. The solver will then make the following decisions (illustrated in Figure 4):

0. Since Z starts first, it is given access to all cache and all its sticky data will fit in the cache. This is the initial guess.

1. X starts and the random sharing equations are solved for the volatile part of the cache. X fills the entire volatile part of the cache with its data since Z has a fetch rate of 0.

[5]The start order can be generalized to more than two applications by first determining the sharing for two applications and using that as the initial guess when start the next application and so on.

Cache line size	64 B
L1 latency	3 cycles
L1 associativity	16
L1 size	64 kB
L2 latency	30 cycles
L2 associativity	16
L2 size	8 MB
Memory latency	200 cycles

Table 1: M5 Simulator parameters

2. Based on the access rates and group sizes, we compute that Z's oldest sticky cache lines are now older than the oldest volatile cache line (i.e., the age requirement does not hold). Z can therefore not keep all its sticky data in the cache. The solver decides to shrink Z until the age condition can be satisfied. The condition is satisfied when the second sticky group becomes volatile.

3. Sharing in the volatile part of the cache is updated using the random model.

4. When the age conditions are satisfied, applying the random model to the volatile part of the cache does not change cache allocations and a stable solution is found.

□

3. EVALUATION (SIMULATOR)

3.1 Experimental Setup

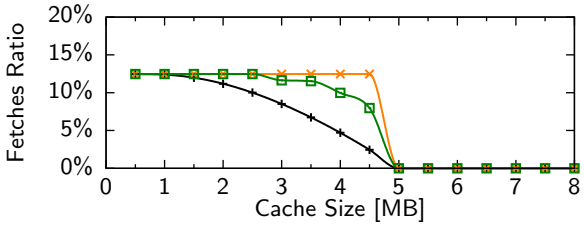
To evaluate the quality of the model, we simulated a simple in-order quad-core processor without prefetching using M5 [1]. The simulated processor implemented a snooping MOESI protocol with all L1 caches connected to a shared L2 cache through a common bus. The simulator does not enforce inclusion between cache levels. The detailed parameters are listed in Table 1.

To obtain the input data for the model, we simulated each application running in isolation and changed the L2 size from 512 kB to 8 MB in steps of 512 kB and measured its cache behavior.

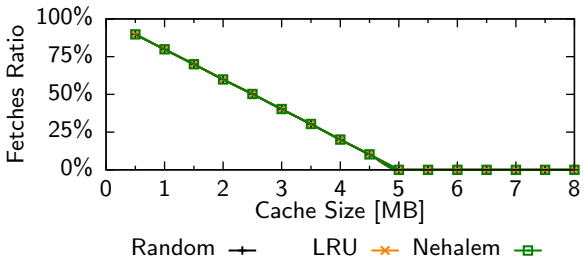
We evaluated our models against three different L2 replacement policies: random replacement, LRU, and the pseudo-LRU algorithm used in the Intel Nehalem microarchitecture [6]. We included the pseudo-LRU policy to determine if our LRU model generalizes to the hardware we use for evaluating the model in Section 4.1. In our experiments, this pseudo-LRU algorithm behaved very similar to normal LRU, so we will limit our discussion to random and LRU.

We selected benchmarks with time-stable behavior from SPEC CPU2006 and PARSEC, and tried to select applications with a wide variety of fetch rate behaviors. Applications with high fetch rates or fetch rates that change significantly when their cache allocation changes are particularly interesting because they are affected by cache contention the most. To avoid unstable start-up behavior, all benchmarks were fast forwarded 85 billion instruction before starting the simulation. The subsequent 2 billion memory accesses were used to drive the simulation.

To further stress our models, we included the two classes of microbenchmarks shown in Figure 5. The first class,



(a) Block (5 MB)



(b) Random (5 MB)

Figure 5: Shared cache fetch ratios for the block and random microbenchmarks.

App 0	App 1	App 2	App 3
block 5 MB	random 3 MB	streamcluster	bodytrack
bodytrack	soplex	astar	lbm
bodytrack	streamcluster	blackscholes	lbm
lbm	leslie3d	astar	bodytrack
libquantum	block 5 MB	random 5 MB	random 7 MB
libquantum	lbm	astar	bodytrack
random 5 MB	streamcluster	astar	leslie3d
random 7 MB	lbm	leslie3d	bodytrack
random 3 MB	block 5 MB	lbm	astar
streamcluster	leslie3d	soplex	bodytrack

Table 2: Mixes of four applications

block, repeatedly accesses its data in a sequential order. This behavior causes the fetch ratio for LRU caches to drop sharply when the cache size is larger than the data set size. The block benchmark is particularly challenging for the LRU model for two reasons: First, since it has a sharp edge on the fetch ratio curve, estimating group ages is hard as it involves taking a derivative of the curve at the point of the sharpest drop off. Second, as seen in Example 2, such benchmarks have a tendency to induce multiple stable sharing configurations in a given pair of benchmarks. The correct configuration generally depends on which application started first.

The *random* microbenchmark class accesses its entire data set randomly. This causes the fetch ratio to decrease linearly with cache size. An interesting observation is that all three replacement policies behave the same in this case. One of the main reasons to include this benchmark is that its steep fetch ratio curve means that a small error in estimating cache allocation will translate into a large error in fetch ratio. For example, a 1 MB error in predicted cache allocation would lead to a fetch ratio error of 20 percentage points, which is larger than the fetch ratio of most normal applications.

We ran all pairs of the following benchmarks from PARSEC: *blackscholes, bodytrack, streamcluster*; SPEC CPU2006: *astar, LBM, leslie3d, libquantum, soplex*; and the following microbenchmarks: *block (3 MB, 5 MB, 7 MB), random (3 MB, 5 MB, 7 MB)*. Since the simulation time needed to simulate all possible combinations of four applications would

be prohibitive, we limited our study to the groups shown in Table 2.

3.2 Simulation Results

3.2.1 Random Replacement

We simulated a random replacement cache and measured the cache allocation and fetch ratio per co-scheduled application. Figure 6a shows the predicted cache size versus simulated cache size and predicted fetch ratio versus simulated fetch ratio for all pairs of applications. The better a prediction, the closer it is to the diagonal. As seen in the figure, there is an excellent agreement between the amount of cache used by the applications and that predicted by the random model. The average error in cache size prediction as a fraction of the total cache size was 0.8%.

Some applications, such as the microbenchmarks in Figure 5, change their fetch ratio significantly when there is only a slight change in cache size. The effect an error in cache size has on the memory system will therefore depend on the shape of an application's miss ratio curve. In order to more accurately assess how the model predicts cache performance, we also evaluated how well the model predicts fetch ratio. We define the relative fetch ratio error as the absolute difference in predicted and simulated fetch ratio over the simulated fetch ratio. It makes little sense to look at relative errors for benchmarks with small fetch ratios, since with a fetch ratio close to zero, even an insignificant error will cause the relative error to explode. Excluding benchmarks with a simulated fetch ratio less than 0.5%, we measure a relative fetch ratio error of 6.1%. The average absolute error for the excluded benchmarks was 0.04%, which corresponds to an insignificant difference in performance.

As seen in Figure 6c, groups of four applications can be predicted with similar accuracy. In this case, the average error in cache size was 0.9% and the average relative error in fetch ratio was 3.3%.

3.2.2 LRU Replacement

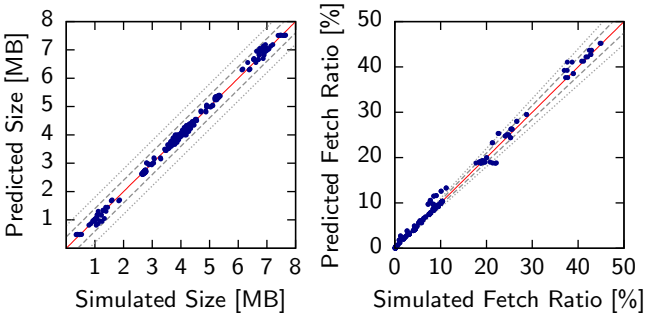
Figure 6b shows the predicted and simulated behavior for pairs of applications with LRU replacement. The scatter plot compares the simulator solutions with their predicted counterparts. The average error in predicted cache size was 0.9% and the average relative error for the fetch ratio prediction was 5.4%. Similar to the random replacement case, we excluded applications with a fetch ratio lower than 0.5% from the fetch ratio average. The average absolute error for the excluded benchmarks was 0.05%. The model accurately solves the more complex task of modeling cache sharing for LRU caches, even the microbenchmarks with sharp edges in their fetch ratio curves can be handled accurately.

As seen in Figure 6d, the average error for groups of four applications is similar to when modeling random replacement. The average error in cache size in this case is 1.3% and the average relative error in fetch ratio is 4.2%.

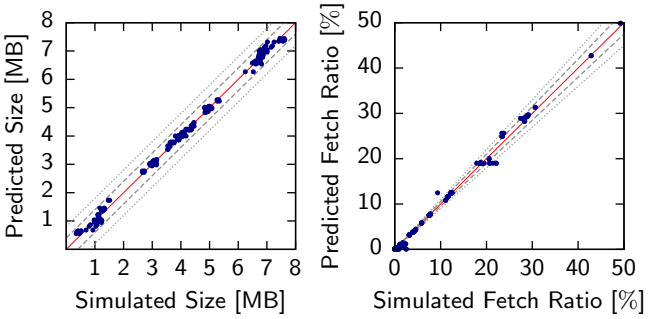
The solver typically finds a solution within 5 to 10 iterations. Our prototype Python-based solver normally finds a solution in less than 100 ms.

3.2.3 Multiple LRU Sharing Solutions

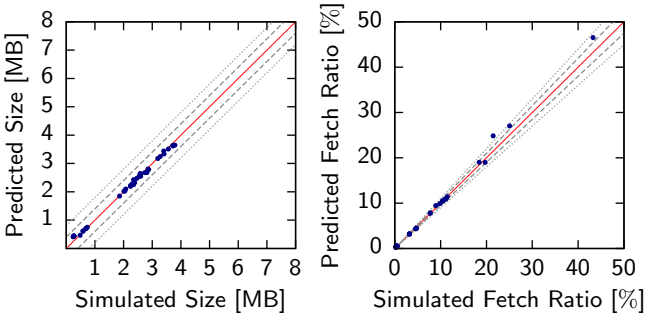
As seen in Section 2.3, some combinations of applications can result in multiple stable sharing configurations. It turns out that such benchmark combinations are uncommon, and



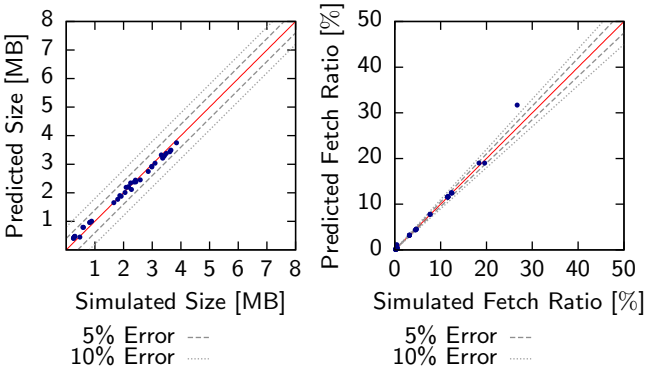
(a) Random replacement, two co-scheduled benchmarks



(b) LRU replacement, two co-scheduled benchmarks



(c) Random replacement, four co-scheduled benchmarks



(d) LRU replacement, four co-scheduled benchmarks

Figure 6: Predicted vs. simulated sharing and fetch ratio

we only observed such behavior for microbenchmarks running in the simulator. In order for a pair of applications to have multiple stable sharing configurations, at least one of the applications must have a sharp knee in its fetch rate curve. In that case, the configuration the simulator finds will depend on

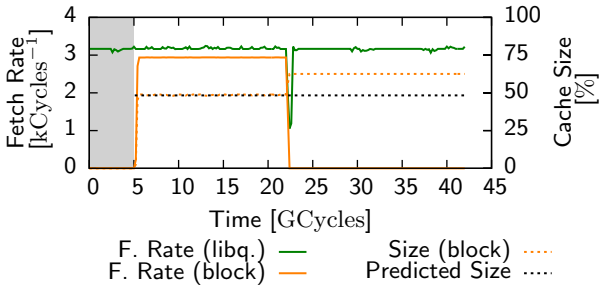


Figure 7: Fetch rate and cache sharing as a function of time for libquantum and the 5 MB block microbenchmark. The shaded part of the graph is the warm-up period where libquantum is executes in isolation.

the start order of the applications. To find such application pairs, we ran every application pair twice, starting one of the applications 5 billion cycles after the other.

Out of the 98 benchmark pairs, the simulator found multiple stable solutions in three cases, all involving the 5 MB block microbenchmark. The model accurately found both solutions in all of these cases by modeling the start order of the applications.

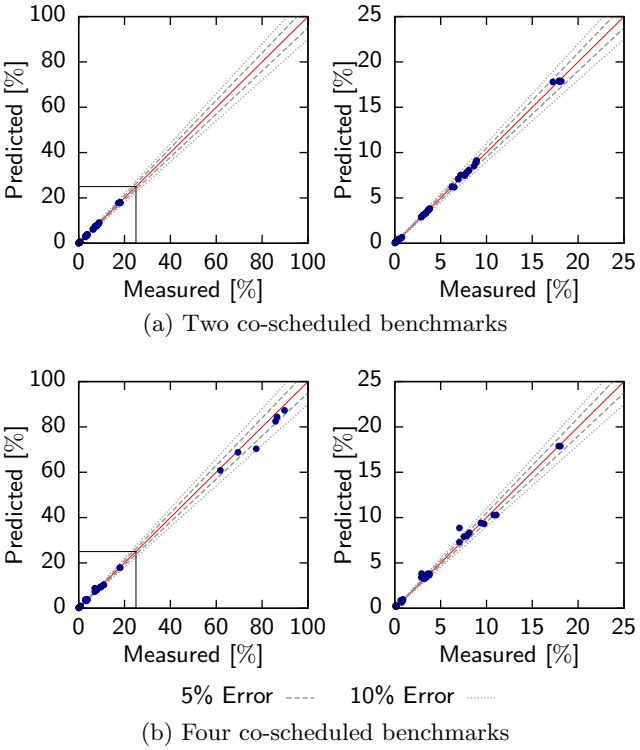
In two benchmark pairs, the simulator found a stable solution, but later switched to a different solution. This can occur for applications, which despite having fairly time stable behavior, have short hiccups where their fetch rate temporarily drops. Figure 7 shows the 5 MB block microbenchmark running together with libquantum. In this case, libquantum started first and was allowed to execute in isolation for 5 billion cycles before the block microbenchmark was started. When the block application starts, its fetch rate immediately rises and stays high since it is unable to make its data sticky. Later libquantum has a short drop in its fetch rate which allows the block microbenchmark to install its entire data set into the cache and stabilize at the second solution. The model, being unaware of libquantum's time-varying behavior, predicts that the simulator will stay in the first solution.

In addition to the five benchmark pairs where the simulator found multiple solutions, our model found an additional solution in four other benchmark pairs. As in the cases where the simulator found multiple solutions, all of these cases involved the 5 MB block microbenchmark. The main reason for the additional solutions found by the model is input data limitations. The model uses sparse data collected for a limited set of cache sizes for each benchmark. This makes it feasible to apply the model to real-world systems, but causes numerical problems for benchmarks with sharp edges in their fetch rate curves. For example, inaccuracies in the input data caused the model to predict two sharing configurations when running the 5 MB block microbenchmark together with the 7 MB random microbenchmark.

4. EVALUATION (HARDWARE)

4.1 Experimental Setup

Our evaluation system consisted of a 2.4 GHz Intel Xeon E5620 system (Westmere) with 4 cores and 6 GB DDR3 memory. Each core has a private 32 kB L1 data cache and a



(a) Two co-scheduled benchmarks

(b) Four co-scheduled benchmarks

Figure 8: Predicted vs. measured fetch ratio for applications running on an Intel Xeon E5620 based system.

private 256 kB L2 cache. All four cores share a 12 MB 16-way L3 cache with a pseudo-LRU replacement policy.

Our cache sharing model requires information about application fetch rate, access rate and hit ratio as a function of cache size. We used Cache Pirating [6] to measure this data for different cache sizes in steps 16 steps of 768 kB (the equivalent of one way) up to 12 MB.

We used the same benchmarks in the hardware study and the simulation study. However, we increased the size of the microbenchmarks' data set by 50% to better stress the 50% larger shared last-level cache.

While measuring the microbenchmarks, we discovered that the Cache Pirate slightly overestimates working set sizes. The average error in working set for the random microbenchmark was 256 kB due to the Pirate application and the monitoring framework using some of the shared cache. We compensated for this error by shrinking the total cache size in the model by this amount and offsetting the input data.

4.2 Results

Figure 8a compares the predicted and measured fetch ratio of pairs of co-scheduled applications. We do not show the amount of cache allocated to each application since there is no accurate way to measure this on the hardware. Unlike the simulator, we only found one solution for each benchmark pair. We believe that the reason for this is that the fetch rate curves of the applications running on our reference hardware do not have as sharp edges as in the simulator. The average relative error in predicted fetch ratio was 7.1%. Similar to the simulator evaluation, we excluded applications with a fetch ratio lower than 0.5% from the average. Figure 8b

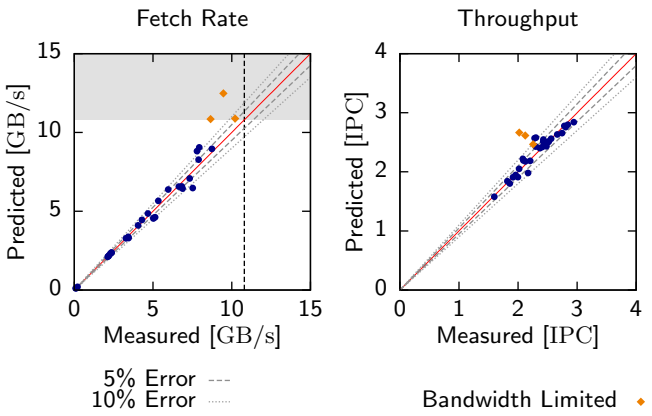


Figure 9: Estimated bandwidth vs. measured bandwidth and estimated IPC vs. measured IPC for pairs of benchmarks running on an Intel Xeon E5620 based system. The gray area indicates the bandwidth limit.

shows the results for groups of four co-scheduled applications. The average fetch ratio error was 6.7%.

4.3 Estimating Bandwidth and Throughput

Knowing how applications share cache allows us to predict other performance metrics, such as bandwidth requirements and throughput. The data measured using Cache Pirating contains information about each application's individual CPI and bandwidth requirement as a function of cache size. Since we can predict each application's cache allocation, we can trivially find its *bandwidth demand*. Assuming that a mix is not bandwidth limited, we can calculate the combined IPC of the mix (throughput) and its expected bandwidth usage.

Knowing the combined bandwidth demand of an application mix can guide a scheduler to avoid mixes with bandwidth demands too close to the system's bandwidth limitation. For mixes well below that limitation, our throughput estimates should be accurate enough to find the best mixes.

We estimated the real-world bandwidth limit of our reference system to approximately 12 GB/s using the STREAM benchmark [12]. We consider an application mix to be bandwidth limited if it uses more than 90% of the maximum bandwidth.

Figure 9 compares the combined bandwidth and throughput of our estimation with the corresponding numbers measured on real hardware. As seen in the figure, as long as the estimated bandwidth is low enough (below 11 GB/s), our bandwidth estimate is quite accurate. Excluding the mixes with a too high bandwidth demand, we can predict the bandwidth of a mix with an average relative error of less than 5.9% and throughput with an average error less than 2.5%.

5. RELATED WORK

Cache sharing models can be divided into two categories: trace driven and high-level data driven. The trace driven models generally use memory access traces or stack distance[6] traces. The benefit of using traces is that they contain detailed information about the execution. Unfortunately, acquiring a memory access trace is slow and storage intensive. Using high-level data, such as sampled memory accesses or statistics provided by performance counters, has become a common approach to reduce data collection overhead.

There are several models [2, 4, 3, 17] using stack distance traces. Chandra et al. [2] pioneered the field with a statistical model that estimates the probability that an access becomes a miss by prolonging its stack distance with the expected number of accesses performed by other applications. One drawback with their model is that it assumes that an application's execution rate is independent of the amount of cache it has access to. Chen and Aamodt [4, 3] extended Chandra's model by including variable execution rate. They also improved the accuracy of the model for low cache associativity by taking the access distribution across sets into account.

The method most similar to ours is CAMP [19] by Xu et al. They use high-level input data, similar to the input data used by our model to model sharing among pairs of applications. However, their model depends on a linear approximation of CPI as a function of fetch ratio. Such an approximation is often inaccurate for processors with out-of-order execution and prefetching. We do not need to model execution rate as this is implicit in our input data. Xu et al. also evaluate two simpler models, which assume that an application's cache share is either proportional to its access rate or its fetch rate. The latter is equivalent to the model we use for random caches, but is applied to LRU caches and approximates fetch rates using their linear execution rate model.

Eklöv et al. proposed a statistical cache sharing model [5] using memory access samples, which can be measured with low overhead. They use a performance model similar to the one used by Xu et al. to estimate the relative execution rate of co-scheduled applications and merge the sampled access streams from each of them. Unfortunately, since they use a linear approximation of execution rate, they suffer from the same drawbacks as the model by Xu et al.

Two recent works focus on estimating how resource contention affects performance. Mars et al. [10] use a stress benchmark to induce contention in an application and then measure its slowdown. The slowdown is used as a contention-sensitivity metric which can be used to guide schedulers. Unlike our method, they do not try to estimate the performance of specific combinations of applications. Instead they focus on a general classification of applications as either being sensitive or insensitive to contention. The approach by Van Craeynest and Eeckhout [16] is more similar to our method in that they estimate the throughput of mixes of applications. 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 low overhead on the target system.

6. FUTURE WORK

We are currently working on extending our models to handle time-varying application behavior. A simple approach would be to slice applications into time windows and estimate sharing between windows. Such an approach would work, however, the amount of data needed would most likely be prohibitively large. Instead, we envision using phase infor-

[6] A stack distance is the number of *unique* between two accesses to the same cache line.

mation, which would enable us to analyze larger regions of stable behavior.

Another exciting direction is to extend the model to more accurately predict throughput for bandwidth limited mixes. This, however, most likely requires a detailed analytical performance model of the processor or performance data as a function of bandwidth.

7. ACKNOWLEDGMENTS

The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). This work was financed by the CoDeR-MP project and the UPMARC research center.

8. 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*, PP(99), 2011.

[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] D. Eklöv, D. Black-Schaffer, and E. Hagersten. Fast Modeling of Cache Contention in Multicore Systems. In *Proc. International Conference on High Performance and Embedded Architecture and Compilation (HiPEAC)*, 2011.

[6] 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.

[7] F. N. Fritsch and R. E. Carlson. Monotone Piecewise Cubic Interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238, 1980.

[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. D. C. Little. A Proof for the Queuing Formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.