Inference-based LLC-side Access Pattern Estimation for Shared Cache Modeling on Commercial Processors

Rakhi Hemani[*], Subhasis Banerjee[**] and Apala Guha[*]

[*]Indraprastha Institute of Information and Technology, Delhi, India.
[**]IBM India
{rakhih, apala}@iiitd.ac.in
subaner4@in.ibm.com

Abstract

Cache contention modeling is necessary for good resource utilization on commercial multicore processors. Our goal is to build cache contention models that are sensitive to changes in, 1) the micro-architecture, 2) the co-runner set of each application, and, 3) the inputs to an application. There are two challenges in achieving this goal: 1) it is difficult to determine the LLC behavior for a given memory access pattern, and, 2) it is difficult to obtain the memory access pattern that reaches the LLC.

We propose, 1) a methodology to generate the behavioral model of LLCs on protected-technology multicore processors, and, 2) an inference-based approach to estimate the LLC-side memory access patterns. We build a cache contention model that uses the behavioral LLC models and the LLC-side memory access patterns. We evaluated the cache contention model on two commercial multicores with Sandy Bridge and Ivy Bridge micro-architectures respectively, using more than a thousand combinations of nine SPEC CPU2006 benchmarks. The average prediction error was 5.74% and 7.27% respectively.

1 Introduction

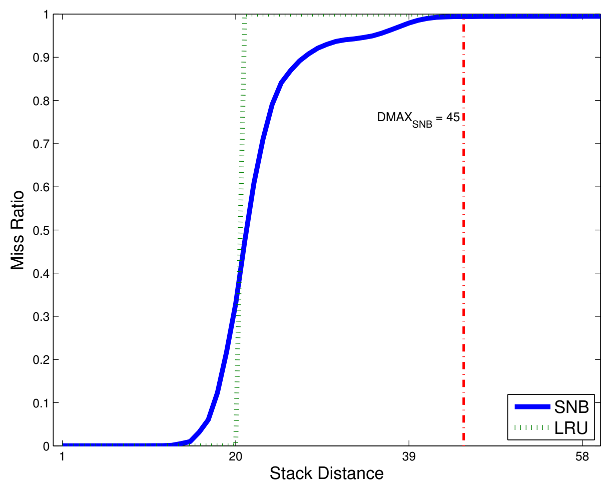
Concurrent execution of multiple applications/threads on multicores is necessary for good resource utilization and cost efficiency, which is critical for all kinds of platforms such as workstations, data centers and clouds. However, concurrent execution also results in contention for shared resources such as the last-level cache (LLC) and may lead to performance degradation. Therefore, we need to model and manage contention. Contention is sensitive to, 1) micro-architectural changes, 2) the co-runner set of each application, and, 3) the inputs to an application. Therefore, contention modeling involves significant effort because it has to deal with variation along all of these axes. Our goal is to build contention models that address all these concerns.

Contention models embody two models within them - a machine model and an application model. We require the machine model to be: 1) protection-proof so that the model works for commercial processors with protected technology, and, 2) micro-architecture-proof so that the model handles changes in the micro-architecture with relative ease. We require the application model to be: 1) co-runner-proof so that we can model applications in isolation and use these models to predict the outcome for any set of co-runners, 2) white-box so that we can use the model as a guide for analyzing the causes of contention and for contention-aware application optimization, 3) input-independent so that new application models do not have to be built for every new input, and 4) portable so that new application models do not have to be built for every new micro-architecture.

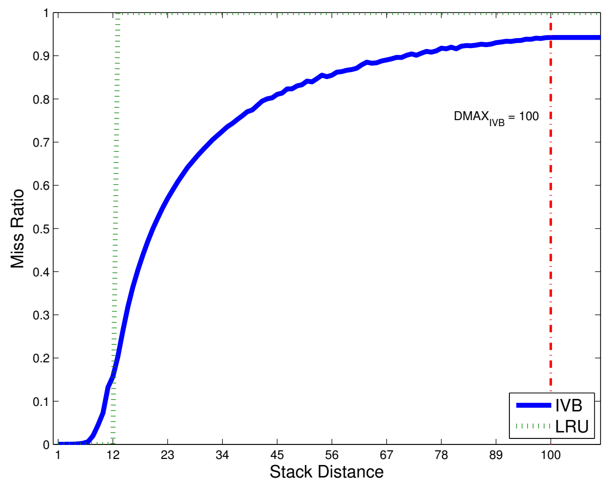
Our proposed solution satisfies all the requirements for the machine model. Our solution also satisfies the first three requirements for the application model. Our solution does not fulfill the portability requirement. But we believe that this research is a precursor to a fully portable contention model, as discussed below. There has been other research in this area, however they either do not fulfill the requirements of the machine model [1, 2], or they are not white-box [3, 4], or they are neither input-independent nor concerned with portability [5, 6, 7, 8]. We discuss the challenges we faced and our proposed solutions below.

The first challenge was to determine the cache behavior for a given access pattern. The behavior would be easy to determine if we knew the implementation of the LLC. However, the challenge is that the implementation of LLCs in commercial multicores is usually protected. While there are many cache implementation policies described in literature, and most caches are said to have a pseudo-LRU behavior, all of these only describe the class of the cache implementation policy. For a particular cache, a significant amount of reverse engineering effort is needed to determine its implementation because the exact parameters with which the cache policy has been configured varies (assuming we are even able to identify the correct class). Also, reverse engineering requires significant domain expertise and may not be practicable for everyone. For example, we found that the cache behavior of the Sandy Bridge micro-architecture varied significantly as compared to the Ivy Bridge micro-architecture. Note that these micro-architectures are only one generation apart. Figure 1 plots the relationship between the *stack distance* and miss ratio of memory accesses, where stack distance is defined as the reuse distance of a memory access when we only consider the accesses going to the same cache set as this access. While the Sandy Bridge machine has a behavior that is closer to the LRU model than Ivy Bridge, it is not obvious how each of these caches are configured.

The alternative to implementation-based models are LLC behavioral models, which are depicted in Figure 1, and are indeed the methodology that we use in this report. However, it is challenging to build the behavioral LLC models of protected, commercial multicores because it is difficult to generate a memory



(a) The behavioral LLC model for an Intel Sandy Bridge machine with associativity 20.



(b) The behavioral LLC model for an Intel Ivy Bridge machine with associativity 12.

Figure 1: An illustration of the behavioral LLC models. Note that the Sandy Bridge and Ivy Bridge behavioral models are dramatically different.

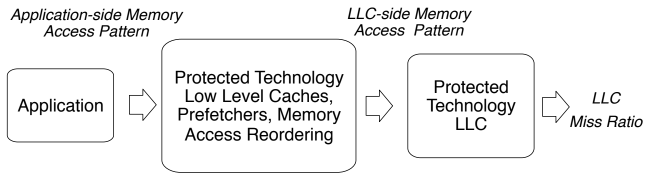

Application-side Memory Access Pattern
Application
Protected Technology Low Level Caches, Prefetchers, Memory Access Reordering
LLC-side Memory Access Pattern
Protected Technology LLC
LLC Miss Ratio

Figure 2: A concept diagram illustrating the relation between application-side and LLC-side memory access patterns and the LLC miss ratio.

access pattern with a uniform stack distance on such platforms. As illustrated in the Figure 2, there are several hardware mechanisms that modify the memory access pattern emitted by an application. While lower-level caches, hardware prefetchers and memory access reordering logic are well-known examples of such mechanisms, their exact configuration is unknown. In protected processors, there may also be other mechanisms whose existence we are not aware of. These mechanisms will especially modify simple, uniform stack distance access patterns which are used to build the behavioral models. We give our solution to this problem in Section 3. For open processors, it is still worth having models that do not require knowledge of the implementation because it reduces the effort.

The second challenge relates to the application model. The contention model basically measures the response of the LLC to the memory access patterns that reach it. As illustrated in Figure 2 the LLC-side memory access pattern is both a function of the application-emitted memory access pattern and interfering hardware mechanisms. While it is very easy to obtain the application-side memory access pattern (through instrumentation, for example), it is difficult to obtain the LLC-side memory access pattern for a protected-technology processor, as the exact configuration of interfering mechanisms is not available to us. Also, there is no direct method such as instrumentation or hardware counters to capture the LLC-side memory access pattern. Figure 3 shows the results of a study to compare the cache miss ratio prediction of a contention model using application-side memory access patterns with the observed cache miss ratio. The prediction error of the application-based model is 68.4% on average, which is very high for solo application runs. In summary, application-side memory access patterns are portable (because they are independent of the micro-architecture) and easy to measure but not accurate, while LLC-side memory access patterns are accurate, but not portable or easy to measure. Therefore, neither is a good choice when we want both accuracy and portability. Therefore the challenge is to model application properties that are not only solo, composable and input-independent, but are also accurate and at the least pave the way for portability.

To solve the first challenge of generating memory address traces with a given

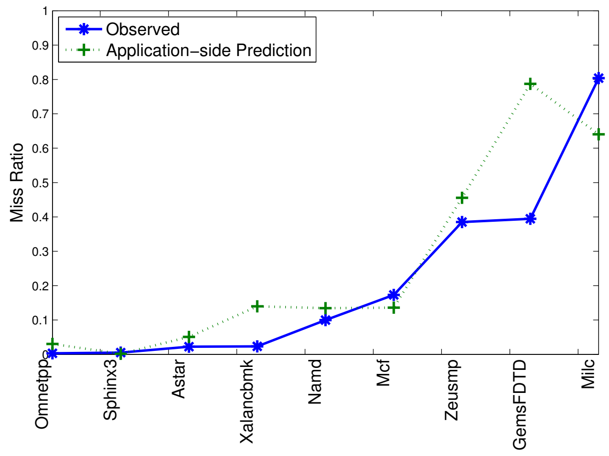


Figure 3: A comparison of the observed LLC miss ratio and the trace-predicted miss ratio for 9 Spec CPU2006 Benchmarks on a Sandy Bridge machine. The accuracy of predictions is low. The correlation coefficient is 0.47 and the error is 68.4%.

stack distance, we created a synthetic benchmark *PSD*, which posts memory accesses with a fixed stack distance in such a way that the pattern is not discovered by the interfering hardware. Since the interfering hardware does not discover any pattern, it leaves the memory access stream unmodified. We describe the solution in detail in Section 3. Figure 1a and Figure 1b depict the LLC behavioral models obtained using the PSD benchmark, for a Sandy Bridge machine and an Ivy Bridge machine respectively.

To solve the second challenge, we need to use a solo, composable, input-independent and white-box property of memory access patterns. Also, we need a property that can be measured both application-side and LLC-side. The idea is that we will use the difference of the LLC-side and application-side properties to create an abstract model of the hardware mechanisms that interfere with the memory access stream before the LLC. We will then use this abstract model to convert application-side models to LLC-side models. This means that we will model applications only once and then produce LLC-models for all micro-architectures of interest without requiring any extra experimentation, thereby achieving true portability. The particular property that we compute is a probability distribution of the stack distance and the *time distance* (time elapsed between two successive accesses to the same location) of the application memory accesses reaching the LLC. This distribution is called the *circular sequence profile* of an application [1]. The stack distance is able to predict the cache miss ratio of an application when it is running solo, because in that situation the cache is only concerned with how many distinct memory accesses to the same cache set have occurred between each reuse. However, when there is more than one application sharing the cache, the time elapsed between each reuse also becomes important in predicting the cache miss ratio, because there are other applications inserting accesses before the reuse, thereby increasing the effective stack distance. For a given stack distance, a higher time distance leads to a higher effective stack distance. Such a property is solo, composable, white-box, input-independent [1, 2, 9] and also has meaning both application-side and LLC-side. While this particular property has been proposed before, the real challenge is estimating it for real processors. We solved this challenge by inferring these properties from the observed cache miss ratio. We co-ran each application with different versions of a synthetic application (with a known circular sequence profile) and observed the change in cache miss ratio from the solo application run. These changes gave us clues about the circular sequence profile for that particular application. Essentially, this method samples the probability distribution since there are a large number of stack distance and time distance combinations. We describe our inference method in Section 4.

The specific contributions of this report are the following:

- Behavioral models of the LLC for two machines with Sandy Bridge and Ivy Bridge micro-architectures respectively.
- A micro-architecture-proof methodology to calculate the behavioral model of LLCs in commercial multicores.
- An inference-based methodology to compute the LLC-side circular sequence profile of an application.
- A demonstration with more than a thousand experiments that our proposed cache contention model produces accurate results for both the Sandy Bridge and Ivy Bridge machines. We experimented with nine SPEC CPU2006 benchmarks, with configurations consisting of a maximum of eight co-running applications. The average prediction error was 5.74% on Sandy Bridge and 7.27% on Ivy Bridge.

The rest of this report is organized as follows: In Section 2 we provide background on set-associative caches, introduce the equation to compute the cache miss ratio for solo applications and discuss the concepts underlying the cache contention model. In Section 3 the method of computing the behavioral cache model is presented, and the obtained behavioral models are discussed. The methodology for estimating the LLC-side application characteristics is discussed in Section 4. Section 5 describes the experimental setup. We present and discuss our results in Section 6. Related work is given in Section 7 and finally we give our conclusions and future work in Section 8.

2 Background

This section provides the background necessary to understand the rest of this report. Section 2.1 gives the general background on set-associative caches. Section 2.2 presents the formula for computing the solo cache miss ratio of an application using the cache behavioral model and the application stack distance profile. Section 2.3 discusses the general concepts underlying the contention model.

2.1 Set-Associative Caches

Caches are storage structures composed of blocks of storage known as *cache lines*. The two extremes of cache implementation are *direct-mapped* caches and *fully-associative* caches. In direct-mapped caches, each memory address maps to a single cache line. In fully-associative caches, any memory address can map to any cache line. Set-associative caches occupy the spectrum between direct-mapped and fully-associative caches. In set-associative caches, each memory address can map to a small group of cache lines, which together constitute a *cache set*. The number of cache lines in each set is known as the *associativity* of the cache.

Caches have widely varying replacement policies, the most well-known one being *LRU* (Least Recently Used). As per this policy, if there is no free cache line in the destination set, then the least recently used cache line is over-written by the incoming data. For LRU caches, memory accesses with a stack distance higher than the associativity are guaranteed to be cache misses, while other memory accesses are guaranteed to be cache hits. However, LRU is difficult to implement in hardware, and pseudo-LRU is often used. Pseudo-LRU has pre-defined levels of recency. Any line in the least recent level may be evicted.

2.2 Computing the Solo Cache Miss Ratio

The underlying principle of computing the solo application cache miss ratio is that the probability of a memory access missing in the LLC depends on how closely it reuses a prior memory access. Specifically, the probability of a cache miss is a function of the *stack distance* of a memory access, which is defined as the number of accesses to distinct locations in the same cache set that appear between this access and the previous access to the same memory location. The cache miss ratio increases with an increase in the stack distance and vice-versa.

For example, assume that A, B and C denote three distinct memory locations going to the same cache set, in the following sequence of accesses. The stack distance of the first three accesses is infinity, that of the second and third access to B is 2 and 1 respectively, that of the second access to C is 2 and that of the last access to A is 3.

$$A\ B\ C\ B\ B\ C\ A$$

Our behavioral cache model gives the relationship between the stack distance and cache miss probability. Once the behavioral model is in place, we need to determine the stack distance profile of the application memory accesses reaching the cache, to be able to predict the solo cache miss ratio. The mathematical formulation of the model is:

$$\hat{MR} = \sum_{d=1}^{\infty} P(d) * M(d) \qquad (1)$$

where, $\hat{MR}$ = the expected solo application cache miss ratio,
$P(d)$ = the probability of memory accesses with stack distance d,
$M(d)$ = the miss probability for accesses with stack distance d.

Similar cache models have been proposed by Guo and Sen et al. [10] and [11]. However, their derivation for M(d) is based on cache implementation details whereas we derive M(d) on the basis of the observed behavior of an LLC. Complete details of our approach are described in Section 3.1.

2.3 Abstract Formulation of the Cache Contention Model

The application whose miss ratio is being predicted is termed as the *victim* and the other co-running applications are termed as *aggressors*. As discussed before, in the context of multiple co-running applications, the notion of *effective* stack distance is used. The effective stack distance of a cache access by the victim is defined as the number of distinct accesses posted to the same cache set by *all* co-running applications between this cache access and the immediately preceding access to the same memory location.

In the example in Section 2.2, the stack distance of the last access is 3. If a co-running aggressor inserts a single access (to memory location Z) to the same cache set in the sequence, and the sequence of accesses changes to the following sequence, then the effective stack distance of the last access increases to four.

$$A\ B\ C\ Z\ B\ B\ C\ A$$

The key to the cache contention model is finding a mapping from the original stack distance of each victim memory access to its effective stack distance. Note that two memory accesses with the same stack distance may not map to the same effective stack distance for a given set of aggressors. The reason is that although these two memory accesses exhibit the same stack distance, they may appear at varying time intervals after the previous access to the same memory location. Varying time intervals implies that the aggressors post different numbers of intermediary memory accesses which leads to differing effective stack distances. In summary, the effective stack distance is a function of the stack distance and the *time distance* of the access.

Therefore, we divide the entire set of victim memory accesses by stack distance as well as time distance and rewrite Equation 1 as Equation 2.

$$\hat{MR}_v = \sum_{d=1}^{\infty} \sum_{r} \wp_v(d,r) * \xi(d,r) \qquad (2)$$

where, r = time distance of a victim memory access,
$\wp_v(d,r)$ = probability of a victim memory access having stack distance d and time distance r i.e. circular sequence probability,

$\xi(d, r)$ = cache miss probability of a victim memory access with stack distance d and time distance r for a given set of aggressors.

The probability distribution of victim memory accesses w.r.t the stack distance and the time distance is defined as the *circular sequence profile* [1]. Here we have expressed the time distance as the total number of intermediary memory accesses to any cache set (a discrete value) rather than the elapsed time (a continuous value), for ease of expression and computation. We assume that the number of memory accesses is directly proportional to the time elapsed for a given application i.e. applications have uniform memory access rates.

For a victim access with a stack distance of d and a time distance of r, let the number of accesses inserted by an aggressor be r_{agg}. Then, the following holds.

$$r_{agg} = (r/AR_v) * AR_{agg} \tag{3}$$

where, AR_v = victim access rate,
and, AR_{agg} = aggressor access rate

For the moment, let us assume that the aggressor impacts all cache sets uniformly. Given r_{agg} we are able to compute $dist(r_{agg})$ i.e. the average number of distinct accesses posted by the aggressor to each cache set using the methodology in [1]. Therefore, the effective stack distance of a victim access with stack distance d and absolute distance r becomes $(d + dist(r_{agg}))$ because the aggressor has inserted $dist(r_{agg})$ lines into each cache set. As a result, the miss probability of the victim access is,

$$\xi(d, r) = M(d + dist(r_{agg})) \tag{4}$$

Now let us relax the assumption that an aggressor impacts all cache sets uniformly. In the cache sets that are unaffected by an aggressor, the effective stack distance remains d. In the other cache sets, the effective stack distance is $d + dist(r_{agg})$. Note that the value of $dist(r_{agg})$ changes upon relaxing the uniform access assumption. Let the fraction of cache sets accessed by the aggressor in the time that it makes r_{agg} accesses be $S(r_{agg})$. $S(r_{agg})$ is defined as the *cache footprint* of the aggressor [12, 2]. Therefore, the miss probability is given by Equation 5, which supersedes Equation 4.

$$\xi(d, r) = S(r_{agg}) * M(d + dist(r_{agg})) + (1 - S(r_{agg})) * M(d) \tag{5}$$

3 A Behavioral Cache Model

As discussed in the introduction, we build a behavioral model of the LLC. We present the methodology to construct the models in Section 3.1, and the models themselves in Section 3.2.

```
 1: procedure PSD( $d$, $buffer$, $NS$, $LS$ )
 2:     loop
 3:         for $i \leftarrow 1, d$ do
 4:             for $start\_index \leftarrow 0, stride - 1$ do
 5:                 $j\_index \leftarrow start\_index$
 6:                 for $j \leftarrow 0, NS/stride - 1$ do
 7:                     $read\ buffer[(i * NS + j\_index) * LS]$
 8:                     $j\_index \leftarrow j\_index + stride$
 9:                 end for
10:             end for
11:         end for
12:     end loop
13: end procedure
```

Figure 4: PSD: A benchmark for generating cache accesses with a given stack distance.

3.1 Constructing the Cache Model

We design a benchmark - Parameterized Stack Distance (PSD), that can be configured to generate LLC accesses with a given stack distance, and observe the cache miss ratio corresponding to each PSD configuration.

The key idea for the PSD benchmark is to insert a parameterized number of cache lines into every set of the cache and keep accessing these cache lines in a circular fashion. However, this is a very simple pattern which will be quickly detected by prefetching hardware mechanisms and get modified. To hide the pattern, we do not access successive cache sets in successive steps. Instead, after accessing each cache set, we skip *stride* sets. For example, if there were 10 cache sets, instead of accessing sets 0 to 9 in order, we may set stride to 5 and access sets in the order 0,5,1,6,2,7,3,8,4,9. We still get the same stack distance as before for each memory access, but the prefetching mechanisms fail to modify the sequence. In reality, the stride values we use are 256 for SNB and 64 for IVB. We found that using these stride values successfully hides the memory access pattern from the prefetching mechanisms as the number of prefetches (measured using hardware counters) became insignificant as compared to the number of cache accesses. If fixed, large stride values do not work, it is possible to go one level further by making accesses in the order 0,9,1,8,2,7,3,6,4,5 for example. In this strategy, the stride value changes with each access.

Also, all accesses made by PSD may not reach the LLC i.e. some of them may be serviced by L1 or L2 caches. To prevent any of the accesses being serviced by L1 or L2, we need to ensure that the cache levels below the LLC should miss for every memory access in line 5. This effect is achieved if the number of distinct accesses between consecutive accesses to the same memory location is much greater than the total number of cache lines in the lower level caches. For our experimental setup (as detailed in Section 5), this property always holds true.

The implementation of this benchmark is given in Figure 4. There are four inputs to this benchmark: the requisite stack distance - d, a large chunk of memory - $buffer$, the number of cache sets - NS and the cache line size - LS. The inner two for loops (lines 4-10) access NS elements once in a strided pattern. Given NS, first the access sequence is divided into *stride* subsequences with consecutive starting indices and size $NS/stride$. Next, given the starting address of a subsequence, $NS/stride$ accesses are made, each access being *stride* elements away from the previous one. The outer for loop (lines 3-11) repeats the inner for loops d times, i.e. it inserts d lines into each cache set. The outermost loop (lines 2-12) repeats this process infinitely. On each iteration of the outermost loop, the same d cache lines are inserted into each cache set, i.e., each cache line is reused after d distinct accesses to that cache set. Therefore the benchmark posts cache accesses with a fixed stack distance d.

3.2 The Cache Models

We built the models for the LLC on a Sandy Bridge and an Ivy Bridge machine. The LLC on the Sandy Bridge is a L3 cache with a size of 20 MB, and an associativity of 20. Figure 1a depicts the cache model, with the blue line corresponding to the actual LLC model and the green line corresponding to the LRU cache model. Note that, the miss ratio saturates to one for stack distance values greater than 45. Thus, the model does not differentiate between stack distances greater than 45. We name this value as $DMAX_{SNB}$.

Figure 1b depicts the LLC behavioral model for Ivy Bridge. Its size is 6 MB and associativity is 12. Notice that the Ivy Bridge machine is sensitive to a larger range of stack distances as compared to the Sandy Bridge machine. We set the $DMAX_{IVB}$ value as 100.

Note that the implementation policy used for both the models is not obvious. Also, the models for Sandy Bridge and Ivy Bridge are dramatically different although they are only one generation apart.

4 Estimating the Circular Sequence Profile and Footprint

The methodology for the circular sequence profile estimation is given in Section 4.1 and that for the cache footprint estimation is given in section 4.2.

4.1 Estimating the Circular Sequence Profile

The key idea is that we co-run the unknown application with the PSD benchmark as the aggressor, for whom the circular sequence profile is known. For a victim with a stack distance of d, and a time distance of r, the solo miss probability is $M(d)$. In the presence of an aggressor, the miss probability is $\xi(d, r)$. As the aggressor has a known circular sequence profile, $\xi(d, r)$ can be computed easily. Therefore, when an application is co-run with an instance of the PSD

```
procedure PSDAR( $d$, $t$, $buffer$, $NS$, $LS$ )
    loop
        for $i \leftarrow 1, d$ do
            for $start\_index \leftarrow 0, stride - 1$ do
                $j\_index \leftarrow start\_index$
                for $j \leftarrow 0, NS/stride - 1$ do
                    $read\ buffer[(i * NS + j\_index) * LS]$
                    $j\_index \leftarrow j\_index + stride$
                    for $k \leftarrow 1, t$ do
                        nop
                    end for
                end for
            end for
        end for
    end loop
end procedure
```

Figure 5: PSDAR: A Benchmark with Parameterized Stack Distance and Access Rate

benchmark, the lhs of Equation 2 is known through observation, and the $\xi(d, r)$ values on the rhs are also known. The only unknowns are the $\wp_v(d, r)$ values. In other words, we have an equation involving the $\wp_v(d, r)$ values as variables. To solve for these unknowns, we must produce as many equations as there are unknowns. We produce these equations by forming the required number of variants of the PSD benchmark, and co-running the application with each variant.

In Equation 2, for every value of r, we compute r_{agg} using Equation 3. The access rate of any application is the ratio of the total LLC accesses made by the application to the application execution time. The total LLC accesses are given by hardware counters. For a given value of r_{agg}, the fraction of cache sets touched by PSD is $min(1, r_{agg}/NS)$. In other words, the cache footprint of PSD is r_{agg}/NS if r_{agg} is less than NS, and 1 otherwise, because PSD keeps accessing NS cache sets in a circular fashion. Similarly, $dist(r_{agg})$ can have a maximum value of δ (the stack distance configuration of PSD) because PSD inserts at most δ distinct elements into each cache set. Otherwise, the average number of cache lines inserted by PSD into each cache set is r_{agg}/NS. In summary, for the PSD aggressor,

$$S(r_{agg}) = min(1, r_{agg}/NS) \tag{6}$$

and,

$$dist(r_{agg}) = min(\delta, r_{agg}/NS) \tag{7}$$

where δ = the stack distance configuration of PSD.

Since, $S(r_{agg})$ and $dist(r_{agg})$ are available, $\xi(d, r)$ can be computed using Equation 5 for every value of d and r. Recall that M values are derived from the behavioral cache model. As a result, in Equation 2, the known values are MR (by observation) and $\xi(d, r)$. The only unknowns are the circular sequence probabilities ($\wp_v(d, r)$) which are functions of d and r. In accordance with Figures 1a and 1b, we vary d from 1 to $DMAX_{arch}$. Note, that for IVB the miss ratio does not saturate to 1 even for large stack distances. To account for this we set the miss ratio corresponding to $DMAX_{arch}$ as 1 explicitly. For a given value of d, r varies from $d+1$ to infinity. Therefore, there are theoretically an infinite number of unknowns. We reduce the number of unknowns by dividing the r values into N ranges, the rationale being that the impact of any aggressor is similar for all r values in a particular range. Thus, the total number of unknowns is the product of $DMAX_{arch}$ and N.

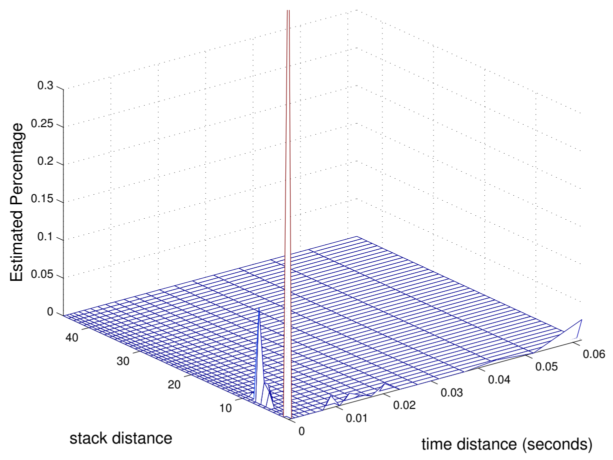
To solve for these unknowns we need $DMAX_{arch} * N$ independent equations involving these terms. To produce these many equations, we design a benchmark, PSDAR (Figure 5), which is a variant of the PSD benchmark. The only difference between PSDAR and PSD is a new input parameter, t, which controls the memory access rate of PSDAR by introducing a certain number of NOPs between two consecutive memory accesses made by PSDAR. By varying the values of the input parameters d and t, we are able to produce $DMAX_{arch} * N$ variants of PSDAR, leading to $DMAX_{arch} * N$ independent instances of Equation 2. We solve these equations using a standard constraint based solver to derive the circular sequence profile.

The specific values of $DMAX_{arch}$ and N are platform dependent and were tuned manually. We set these values as 45 and 21 for the Sandy Bridge machine and 100 and 9 for the Ivy Bridge machine. The requisite number of experiments can be lowered as we found that the cache miss ratio of an application is independent of δ after a particular threshold. The threshold varied from one application to the other. Based on this insight, we are working towards a strategy for minimizing the number of experiments.

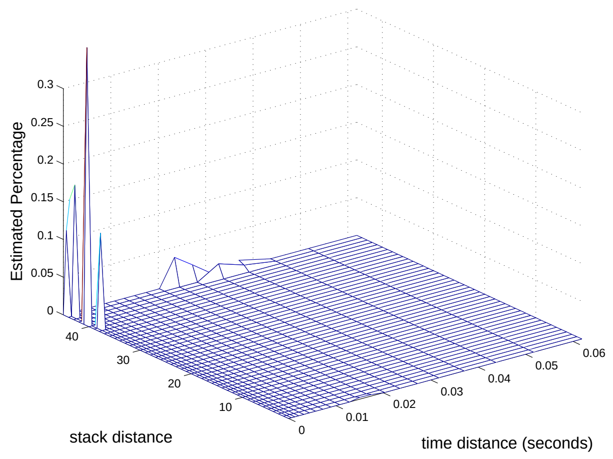
The circular sequence profile derived by solving the system of equations for two SPEC CPU2006 benchmarks, *Sphinx3* and *Milc* on the Sandy Bridge machine are depicted in Figure 6a and Figure 6b, respectively. Most of the circular sequences have a low stack distance for *Sphinx3* while most of the circular sequences have a high stack distance for *Milc.* As confirmation of this fact, the solo miss ratio of the *Sphinx3* benchmark was found to be lower than that of *Milc.*

4.2 Estimating the Cache Footprint

The methodology for estimating the cache footprint of an application is in some ways the reverse of the methodology for estimating the circular sequence profile. The unknown application is now considered as an aggressor while the PSDAR benchmark is the victim. The circular sequence profile of PSDAR reduces to a single number because it only posts memory accesses with a single stack distance d, and all the circular sequences have the same time distance, $NS * d$. In other words, $P(d, NS * d) = 1$. Substituting this value into Equation 2, we find the value of $\xi(d, NS)$, substituting $\hat{MR}$ with observed values. Substituting,



(a) Sphinx3.



(b) Milc.

Figure 6: Estimated circular sequence profiles for 2 SPEC CPU2006 benchmarks. Notice that Sphinx3 forms a large peak for small stack distances as compared to Milc. Thus, Sphinx3 is estimated to have circular sequences with small stack distances as compared to Milc.

Table 1: Shared cache configuration for SNB and IVB

Machine	L3 size	Associativity	Number of sets
SNB	20MB	20	16384
IVB	6MB	12	8192

$r = NS * d$, in Equation 3, we compute r_{agg}, since AR_v and AR_{agg} are known through observation. Substituting d, r_{agg} and $\xi(d, NS)$ in Equation 5, we find that the only unknowns are $dist(r_{agg})$ and $S(r_{agg})$. As an added complication, $dist(r_{agg})$ depends on $S(r_{agg})$. We solve the equation iteratively by assuming a starting value of $S(r_{agg})$, using it to compute $dist(r_{agg})$, substituting $S(r_{agg})$ and $dist(r_{agg})$ into Equation 5, computing the difference between the lhs and rhs of Equation 5, using this difference to refine the value of $S(r_{agg})$ and repeating the process until the difference falls below a certain threshold.

To compute $S(r_{agg})$ and $dist(r_{agg})$ for different values of r_{agg}, we vary the configuration of the PSDAR benchmark and repeat the entire process. The requisite number of configuration changes was tuned manually and was set to 40 for both Sandy Bridge and Ivy Bridge micro-architectures.

5 Experimental Setup

The metrics used by us are discussed in Section 5.1. Section 5.2 describes the configuration of the machines and Section 5.3 enumerates the applications used as test cases. Finally, the experimental methodology for conducting the experiments is described in Section 5.4.

5.1 Metrics

In order to validate the prediction made by the cache contention model we need to compare the observed cache miss ratio with the predicted value. For this, we compute the ratio of the absolute difference between the two values to the observed value. It may be noted that even for small differences in the absolute error, the error percentage is quite high if the observed miss ratio is very small. Thus, a simple arithmetic mean of all error percentages is misleading because it is overly sensitive to outliers. The geometric mean however does not suffer from this problem as it indicates the central tendency of a set of numbers and is not affected by outliers. Thus, we consider the geometric mean of the error percentages. We report the arithmetic mean for the observations also.

We quantify the ability of the prediction to rank different workloads for a given victim using a weighted rank inversion metric (RIM). This metric quantifies the degree to which the predicted ranks differ from an oracle predictor. It is computed as follows: First all workloads corresponding to a victim are sorted according to their predicted miss ratio in an ascending order to obtain the predicted rank for each workload. Next, for all pairs of workloads (W1,W2) such that the predicted rank of W1 is lower than W2, if the observed miss ratio corresponding to W1 is greater than that of W2 then this pair is counted as an inverted rank pair. For each such pair a weight is computed as the absolute percentage difference between the observed miss ratio of W1 and W2, the rational being that the seriousness of a rank inversion is proportional to the absolute difference of the observed values . The RIM metric is computed as the weighted average of all pairs.

5.2 Test Machine Configuration

We used two machines, *SNB* and *IVB* for our experiments. SNB is a Dell R720 Server with a 8-core, Sandy Bridge Intel Xeon E5-2665 processor, a 2.4 GHz clock rate and 64 GB of main memory. IVB is a Dell OptiPlex 3010 workstation, with a 4-core, Ivy Bridge Intel i5-3470 processor, a 3.2 GHz clock rate and 8 GB of main memory.

The L1 cache size for all the three machines is 32KB and L2 cache size is 256 KB. The cache line size is 64 bytes. The L3 cache parameters are described in Table 1.

5.3 Benchmark Selection

For validating our model, we considered nine SPEC CPU2006 applications, namely *Omnetpp, Sphinx3, Astar, Xalancbmk, Namd, Zeusmp, GemsFDTD and Milc* with train inputs. The same set of binaries was used for all machines. The applications were compiled using gcc 4.7.2.

5.4 Experimental Methodology

To validate the cache contention model we conducted a large number of experiments on SNB and IVB. SNB has 8 cores, so we considered all workloads with all possible combinations of one, two, four and eight applications. The possible number of such workloads for a set of 9 applications is $C(9,1) + C(9,2) + C(9,4) + C(9,8)$. Note that for each workload of J applications, any of the J applications can be considered as a victim. Thus, the total number of predictions to be made is $C(9,1) + C(9,2) * 2 + C(9,4) * 4 + C(9,8) * 8 = 657$. We ensured that all aggressors were running while the victim was running. If an aggressor finished before the victim then it was restarted.

A similar methodology was followed for IVB. However, the number of co-running workloads is lower as the number of cores is 4. The total number of configurations considered for this machine is $C(9,1) + C(9,2) * 2 + C(9,4) * 4 = 585$.

Applications were pinned to physical processors using numactl or hwloc-bind. For all workloads comprising of multiple applications, each application was run on a different physical core. The Perf utility was used to measure the number of last level cache misses and cache accesses.

App	SNB (%)	IVB (%)
471.omnetpp	2.06	8.06
482.sphinx3	2.08	4.38
473.astar	1.42	3.13
483.xalancbmk	1.18	5.72
444.namd	1.12	4.12
429.mcf	0.69	3.81
434.zeusmp	0.17	1.18
459.GemsFDTD	0.92	2.09
433.milc	0.04	0.34
Average	1.08	3.65

Table 2: Results for the rank inversion metric. The numbers denote the degree to which the predicted ranks differ from an oracle predictor.

6 Results

The cache contention model is validated in Section 6.1. In Section 6.2 we discuss the assumptions and limitations of our model. In Section 6.3, we present an experiment on cross-platform prediction to gain deeper insights into the problem of portability.

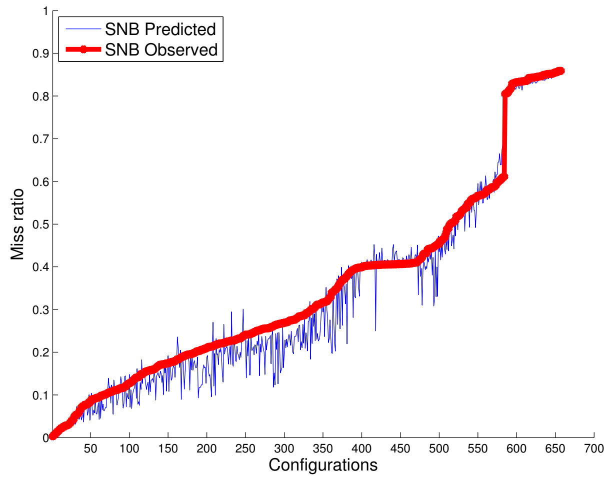
6.1 Cache Contention Experiments

The predictions made by our cache contention modeling strategy are compared with the observed values in Figure 7 for both SNB and IVB. The x-axis corresponds to the different configurations as discussed in Section 5.4. The configurations are sorted with respect to the miss ratio of the victim. The red line plots the observed value while the blue line plots the predicted value. The geometric means of the error percentages were 5.74% and 7.27%, and the arithmetic means were 13.17% and 14.67% for SNB and IVB respectively. The error rate remained stable across all workload sizes, so we do not report these errors separately.

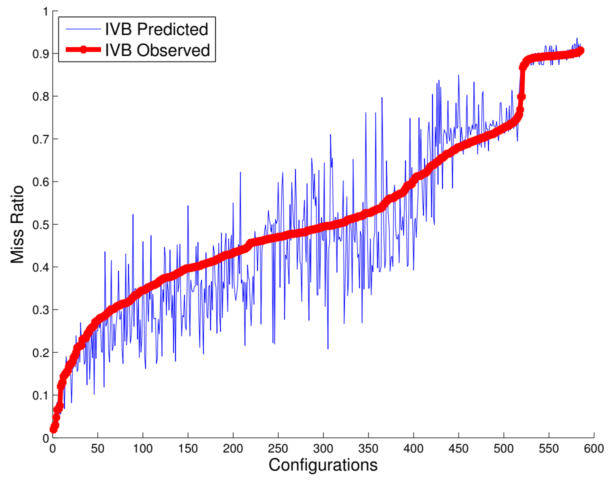
The RIM metric for both platforms is reported in Table 2. The values are 1.08% on Sandy Bridge and 3.65% on Ivy Bridge. Recall that these numbers denote the degree to which the predicted ranks differ from an oracle predictor.

6.2 Discussion

The error in the predicted values may be attributed to the following assumptions of the current model:



(a) Observed versus Predicted Miss Ratio for the SNB machine. There are 657 points. The average error is 5.74%.



(b) Observed versus Predicted Miss ratio for the IVB machine. There are 585 points. The average error is 7.27%.

Figure 7: Predictions for cache contention on two different micro-architectures.

1. Phase Changes: It is well known that the applications exhibit changes in phase. However, our current model does not consider them. The model computes the average behavior of an application. We could have improved accuracy further by considering phases, but accuracy by itself is not the topic of this work.

2. Solo Access Rate: We consider solo access rates of applications. We observed that the access rate decreased if the miss ratio increased, however we did not consider this variation. We found that our predictions were quite accurate, even in the presence of this simplifying assumption.

The limitations of the cache contention model are as follows:

1. No data sharing: We have assumed that there is no data sharing between applications, i.e., the applications were not multi-threaded. However, the proposed model is applicable to the sequential parts of a multi-threaded program. We are working towards extending this work for multi-threaded applications.

2. Prior information on applications: Our approach assumes that the set of applications to be executed is known beforehand. This is a basic limitation of all white-box cache models, as building white-box models takes time and has to be done offline.

6.3 Cross-platform Prediction and Portability

In addition to verifying the applicability of the estimated cache access pattern to cache contention prediction, we performed an interesting experiment to test the feasibility of our approach for cross-platform prediction.

We convolved the LLC-side stack distance profiles of applications on IVB with the behavioral cache model of another machine to obtain a prediction of the solo LLC miss ratio for the new machine. This new machine, named as *SNB2*, is a Dell Precision T3600 workstation, with a 4-core, Sandy Bridge Intel Xeon E5-1620 processor. The L1 and L2 cache sizes of this machine are the same as that of IVB. The LLC cache size of the new machine is 10MB and its associativity is 20. The number of LLC cache sets and line size is the same for both these machines i.e. any memory address will reach the same cache set on both these machines. If for a given executable file, the same memory addresses reach the IVB and SNB2 LLCs, then they will give rise to the same circular distance profile. However, the interfering hardware mechanisms may differ, and therefore the memory addresses reaching the two LLCs may be different. The goal was to compare predictions made on fully portable application models with those made on platform-dependent application models. But the platform-dependent application models were impaired by the fact that they were being used for cross-platform prediction.

The results of this experiment are depicted in Figure 8. The observed miss ratios are depicted with blue stars. The red triangles denote the predictions

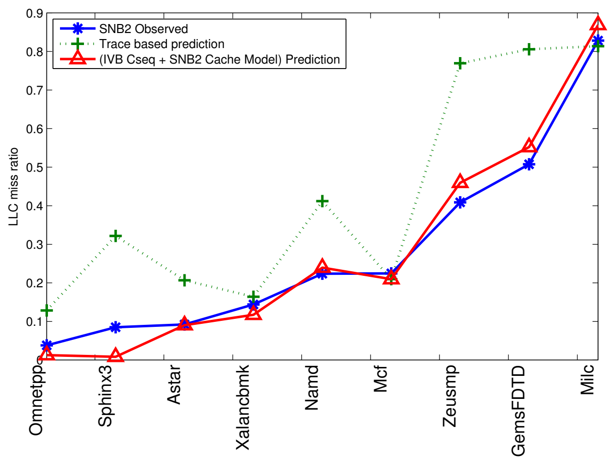


Figure 8: Prediction for miss ratios on the SNB2 machine. The two methodologies used were: 1) the circular sequence profiles derived from IVB were convolved with the behavioral cache model for SNB2 (red line), and, 2) a trace-based approach (green line). The red line approximates the blue line better than the green line.

made by the experiment described above, while the green dots represent the predictions made from a simple trace-based approach. It is observed that the predictions obtained by convolving the LLC access pattern with the cache model have better accuracy than a trace-based approach. The geometric means of the error were 12.15% and 43.10% respectively. The arithmetic means were 24.10% and 99.38% respectively. This experiment validates that some notion of platform interfering mechanisms is needed to get accuracy. At the same time, modeling each application for each machine is time-consuming. Therefore, we believe that determining LLC-side memory access patterns is important as it paves the way to portability.

7 Related Work

In this section we review the existing work on cache contention modeling. We classify existing cache contention models as : 1) co-runner-proof, 2) white-box, 3) input-independent, 4) portable, 5) protection-proof, and, 6)micro-architecture proof. The summary of the comparison of popular proposals for cache contention and our model is presented in Table 3.

First we describe two popular "online" models for cache contention modeling. These models utilize aggregate metrics such as the cache miss ratio to optimize the performance of programs at run-time. Examples of this approach include the DI policy proposed by Fedorova et al [4], and the bubble-flux model proposed by Mars et al [3]. The DI policy classifies each application on the

Work	Co-runner Proof	White-box	Input Independent	Portable	Protection Proof	Micro-architecture proof
Bubble-Flux [3]	Yes	No	Yes	No	Yes	Yes
DI Policy [4]	Yes	No	Yes	No	Yes	Yes
Chandra et al. [1] and Chen et al. [2]	Yes	Yes	Yes	Yes	No	No
All Window Profiling [9]	Yes	Yes	Yes	Yes	No	No
Bubble-UP [13, 6]	Partial	Yes	No	No	Yes	Yes
Empirical model [5]	Yes	Yes	No	No	Yes	Yes
CAMP [7]	Yes	Yes	No	No	Yes	Yes
Sandberg et al. [8, 14]	Yes	Yes	No	No	Yes	Yes
Our Model	Yes	Yes	Yes	Partial	Yes	Yes

Table 3: A comparison of existing cache contention models with our model.

basis of its miss ratio. The key assumption is that cache contention is more for applications with a high miss ratio. However, it has been proved that cache contention is dependent upon other factors such as the cache access rate [6]. The bubble-flux model [3] probes each system dynamically to estimate the amount of pressure that a machine executing a sensitive application can withstand along with the pressure exerted by an application to identify "safe" co-locations for the latter. The major limitations of such "online" models are: 1) they are not white-box, i.e., they do not provide insights into the causes for contention that could be used for contention-aware application optimization, 2) lack of portability as application behavior is platform-dependent and the models need to be recomputed for each platform under consideration, and, 3) they necessitate monitors on each machine being considered and thus have a run-time overhead.

Offline models on the other hand do not suffer from these limitations as they characterize application behavior a priori. Stack distance profiling (or circular sequence profiling) is one of the major techniques for characterizing the application behavior for cache contention modeling. Chandra et al proposed a cache contention model on the basis of circular sequence profiles [1]. However, they assumed that applications access all the cache sets equally. This assumption was addressed in a later work by Chen et al [2]. The added advantage of these models is input-independence [15]. In addition, they provide the basis for platform independence. However, for true platform independence, it is required that the application model is platform-independent. For example, Ding et al [9] demonstrate that a theoretical model captures the high order trends for cache contention modeling. However they also mention that given the complexity of the hardware mechanisms it is difficult to estimate LLC-side memory access patterns. In this report, we have overcome this limitation and have proposed a methodology for predicting such LLC-side memory access patterns.

Another category of cache contention models learn the aggregate behavior of an application and the LLC on a specific platform. While these techniques succeed at the goal of finding the cache contention behavior of applications, they do not attempt to separate application and architecture behavior at all. Performing this separation is a pre-cursor to portability and input independence. We examine some popular strategies in this category below.

The *Bubble Up* Model [13] measures the sensitivity of applications, by calibrating their behavior in the presence of a benchmark that exerts an incremental pressure on the memory system. The contentiousness of an application is measured by finding the "pressure" exerted by the application itself. However, this model has been proved to be applicable for only two co-running applications and is hence not co-runner proof. The *empirical* model [5] is a similar model without this limitation. The model quantifies applications on the basis of their solo cache and memory pressure. The application is then profiled for varying cache and memory pressure. The variation in cache and memory pressure is obtained by co-running different applications.

The CAMP model [7] aims to solve for the "cache size" or the average number of "ways" occupied by an application by profiling it in the presence of a benchmark with special memory access properties and the "Cache Pirating" based model in [8] profiles an application to obtain the amount of "sticky" and "volatile" data. Though these models are based on input data similar to ours, their proposed methodology is not aimed at portability or input independence.

Cache models corresponding to various cache replacement policies have also been proposed [10, 11]. However, the application of these models in the context of commercial multicore processors is difficult as the actual cache replacement policy in an modern multicore is often protected by the IP rights of the processor manufacturer. Even if this policy is known, for example for open architectures like Sun Niagra, a lot of effort in terms of code testing and verification needs to be invested for every new generation of processors. Our behavioral cache model in contrast is easily portable to each new generation of processors. .

8 Conclusions and Future Work

To conclude, we propose an inference-based approach to estimate the LLC-side memory access patterns, and an approach to build the behavioral cache models for cache contention modeling on two recent commercial processors. The error percentages of the predictions were 5.74% and 7.27% for the machines with Sandy Bridge and Ivy Bridge micro-architectures respectively. We confirm that platform-dependent models perform better than the platform-independent ones, even when the prediction is cross-platform. We believe that our techniques pave the way for building truly portable application models.

The directions for future research are:

1. Building truly portable application models by correlating application-side memory access patterns with LLC-side memory access patterns.

2. Using the information gained by the memory access patterns for application optimization.

3. Extending the model for multi-threaded applications.

References

[1] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 340–351, Feb 2005.

[2] X. Chen and T. Aamodt, "Modeling cache contention and throughput of multiprogrammed manycore processors," *Computers, IEEE Transactions on*, vol. 61, pp. 913–927, July 2012.

[3] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 607–618, ACM, 2013.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, (New York, NY, USA), pp. 129–142, ACM, 2010.

[5] J. Zhao, H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang, "An empirical model for predicting cross-core performance interference on multicore processors," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 201–212, IEEE Press, 2013.

[6] L. Tang, J. Mars, and M. L. Soffa, "Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, (New York, NY, USA), pp. 12–21, ACM, 2011.

[7] C. Xu, X. Chen, R. Dick, and Z. Mao, "Cache contention and application performance prediction for multi-core systems," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 76–86, March 2010.

[8] A. Sandberg, D. Black-Schaffer, and E. Hagersten, "Efficient techniques for predicting cache sharing and throughput," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 305–314, ACM, 2012.

[9] X. Xiang, B. Bao, T. Bai, C. Ding, and T. Chilimbi, "All-window profiling and composable models of cache sharing," *SIGPLAN Not.*, vol. 46, pp. 91–102, Feb. 2011.

[10] F. Guo and Y. Solihin, "An analytical model for cache replacement policy performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 34, pp. 228–239, June 2006.

[11] R. Sen and D. A. Wood, "Reuse-based online models for caches," in *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, (New York, NY, USA), pp. 279–292, ACM, 2013.

[12] D. Thiebaut and H. S. Stone, "Footprints in the cache," *ACM Trans. Comput. Syst.*, vol. 5, pp. 305–329, Oct. 1987.

[13] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 248–259, ACM, 2011.

[14] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten, "Cache pirating: Measuring the curse of the shared cache," in *Parallel Processing (ICPP), 2011 International Conference on*, pp. 165–175, Sept 2011.

[15] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," *SIGPLAN Not.*, vol. 38, pp. 245–257, May 2003.