Modeling Cache Coherence Misses on Multicores

Xiaoyue Pan   Bengt Jonsson
Department of Information Technology
Uppsala University
{xiaoyue.pan, bengt.jonsson}@it.uu.se

***Abstract*—While maintaining the coherency of private caches, invalidation-based cache coherence protocols introduce cache coherence misses. We address the problem of predicting the number of cache coherence misses in the private cache of a parallel application when running on a multicore system with an invalidation-based cache coherence protocol. We propose three new performance models (uniform, phased and symmetric) for estimating the number of coherence misses from information about inter-core data sharing patterns and the individual core's data reuse patterns. The inputs to the uniform and phased models are the write frequency and reuse distance distribution of shared data from different cores. This input can be obtained either from profiling the target application on a single core or by analyzing the data access pattern statically, and does not need a detailed simulation of the pattern of interleaving accesses to shared data. The output of the models is an estimated number of coherence misses of the target application. The output can be combined with the number of other kinds of misses to estimate the total number of misses in each core's private cache. This output can also be used to guide program optimization to improve cache performance. We evaluate our models with a set of benchmarks from the PARSEC benchmark suite on real hardware.**

I. Introduction

The cache system on modern multicore architectures typically consists of a shared last-level cache for multiple cores and private cache(s) for each core, which are kept coherent by a coherence protocol. For parallel applications in which several cores access common shared data, the cache system affects performance in many ways. On the one hand, the shared last-level cache reduces the number of accesses to off-chip memory, since data that is brought into the shared cache by one thread can subsequently be accessed by other threads without going to off-chip memory. On the other hand, shared data accessed by several cores, must be moved between the private caches of the corresponding cores, causing some overhead. Furthermore, whenever a core modifies shared data, copies in other private caches are invalidated; when those copies are later accessed by their cores, this triggers a so-called *coherence miss* in the private cache of that core, typically forcing the data to be fetched from the last-level cache.

To illustrate how coherence misses can cause performance downgrade, consider the "Deduplication" stage of the *dedup* benchmark in the PARSEC benchmark suite [1, version 3.0]. In this stage, a large set of data chunks are processed in parallel by the threads. During the processing of each chunk, the threads access a shared hash table with global data. Figure 1 shows how the number of misses in the private caches (both L1 and L2) increases with the number of cores, for 1 to 8 threads (each thread is pinned to a separate core with private L1 and L2 caches, and all cores, being on the same socket, share a last-level cache). The average number of private cache misses for each core increases by 34% when going from 1 to 8 threads. The cache misses mainly come from the critical section where the shared data is accessed. This increase in the number of private cache misses is caused by an increase in coherence misses, and causes the execution time in the critical section that protects the shared data to increase by 30%. The increased critical section time leads to further performance-harming effects, including increased waiting time at lock accesses. The end result is that the speedup obtained by parallelizing the stage is far from linear. The actual measured speedup for this stage is 4.72 with 8 threads. Without the coherence misses, the execution time in the critical section would not increase, avoiding the further performance-harming effects. By a detailed calculation, one can conclude that the speedup achieved without the coherence misses would be around 7.5. This loss of speedup is purely caused by coherence misses.

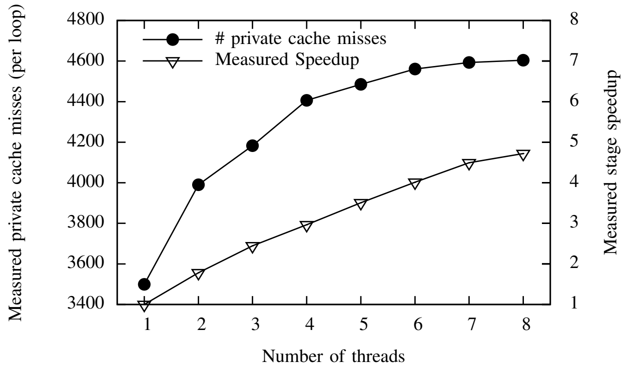

Fig. 1: Private cache misses (L1 + L2) and speedup of the Deduplication stage in *dedup*

The example shows that when parallelizing an application onto several cores that share common data, it is important to understand how the cache system affects the performance of the application and to be able to predict the number of expected cache misses of different forms. It furthermore shows that an increase in coherence misses can be an important bottleneck which must be considered, when deciding how to distribute code and data over the cores. For instance, the cost of coherence misses can be a reason to choose a parallelization that minimizes data sharing between cores, even if this would incur other overheads. The cost of coherence misses might also be something to consider when parallelizing code in a compiler. For example, in the aforementioned *dedup* benchmark, we can reduce the number of coherence misses by padding the shared data's data structure.

The cost of coherence misses should also be considered when predicting which speedup can be obtained by parallelization.

Modeling and predicting coherence misses can of course be done using detailed simulation. Such analysis can be very costly for long-running parallel applications. For predicting other types of cache misses than coherence misses, several less expensive approaches have therefore been developed for modeling cache performance. There are, e.g., approaches that use sampling of native executions (e.g., [2], [3]) with overheads of only a few 10%. This approach has also been extended to model how separate applications compete for a shared cache (e.g., [4], [5], [6]). However, to the best of our knowledge, there is no work on modeling caches which can predict the number or cost of coherence misses in multi-threaded applications.

In this paper, we present three models for modeling and predicting coherence misses in parallel applications. They all rely on input that can be obtained by low-overhead profiling techniques, such as quick sampling techniques that capture reuse distance information (e.g., [2], [3]). The input to the first two models (the uniform and the phased model) are reuse distance distributions and write frequencies by the different threads to shared data. The first (uniform) model uses this data to predict the number of coherence misses of the application. Its precision relies on the assumption that accesses to shared data by different threads occur uniformly and temporally uncorrelated. This assumption holds for applications that access shared data in a uniform way, e.g., a common storage, lookup table, or similar.

For applications that use synchronization to force a particular pattern of access to shared data, the assumption of temporally uniform access to shared data does not hold. The second (phased) model can cope with this problem by dividing the execution into phases, and using as input the reuse distance distributions and write frequencies for each phase. Typically the boundary of a phase is marked by synchronization primitives (conditional variables, barriers, etc). These synchronization primitives are used to divide the execution. The model is first applied in each phase of the execution. Then we sum up the number of coherence misses in each phase to obtain the total number of coherence misses in the whole execution.

Our third (symmetric) model can use less detailed input data than the two other ones. It assumes that the shared data of a parallel application is accessed symmetrically and uniformly by the threads. Under this assumption, it needs only the number of private cache misses for two runs with a different number of threads. The model predicts the expected number of private cache misses for any number of threads.

We have implemented the uniform and phased models in the PIN framework [7]. This implementation is not efficient but is intended to evaluate our models. We expect that an efficient implementation can be realized by low-overhead sampling techniques. Given a target program, we first trace the memory accesses by each thread. The number of cold misses is estimated by counting the number of distinct cache lines accessed. We used the conventional set stack distance to pick out all the capacity and conflict misses. To calculate the the number of coherence misses, first we get the reuse distance distribution and write frequency to shared data from the trace. Then these parameters are fed to our model to get the number of coherence misses. Finally, the numbers of all four kinds of misses are combined to estimate the total number of cache misses in the private cache.

We evaluate our models on a set of applications from the PARSEC benchmark suite [1] on actual hardware. Since the current hardware performance counters do not distinguish coherence misses from the rest of the cache misses, we compare the modeled and measured *total* number of cache misses in the private cache. The average relative error for the uniform model is 5.80%. The average relative error of the phased model is 8.02%. We evaluated the symmetric model on one benchmarks (*dedup*), which gives an error rate of 5.4%.

Our proposed models provide a quantitative estimate of the number of coherence misses. This information can be used to optimize the program. If a program is likely to have too many coherence misses, redesigning the data access and sharing pattern may reduce the number of coherence misses. The model can also be used to predict the number of private cache misses with different cache sizes.

The rest of the paper is organized as follows: Section II introduces the intended target multicore systems and describes the different kinds of private cache misses on the target system. Section III describes the assumptions, terms and notations used throughput the paper. Section IV reviews the methods to model cold, capacity and conflict misses. Section V presents our cache coherence models in detail. Section VI discusses the implementation of two of our proposed models (model uniform and phased). Section VII shows the model evaluation results. Section VIII discusses related work and compares it to this work. Section IX concludes the paper.

II. Cache miss categorization

We consider a multicore system with a hierarchical cache system: multiple cores share a last level cache and each core has a private (possibly multi-leveled) cache. In the shared cache, the sharing effect could be both constructive and destructive. On the one hand, cores may bring in the shared data, which reduces the number of cache misses for other cores. On the other hand, cores may evict other cores' data, which increases the number of cache misses for other cores. In a core's private cache, other cores cannot bring in or directly evict data. However, the shared data in one core's private cache can be invalidated due to another core's write to it (assuming an invalidation-based cache coherence protocol).

Modeling and predicting both the constructive and destructive sharing effects of cache sharing has been studied previously [4], [5], [6]. However, we are not aware of any proposed model for efficiently modeling and predicting the effect of data sharing on private caches. In this section, we discuss the cache misses in the private cache with invalidation-based cache coherence protocols.

In a single-core system, there are three kinds of cache misses according to Hill et al's cache miss categorization [8]: *compulsory misses* (also known as *cold misses*), *capacity* *misses*, and *conflict misses*. Cold misses occur when the data is accessed for the first time. Both capacity misses and conflict misses are triggered by data eviction. The eviction for capacity misses is caused by limited cache capacity, while for conflict misses it is caused by limited cache associativity.

In a multicore system, all cores still experience these three kinds of cache misses. In addition, cores that share data may also suffer from *cache coherence misses* (also known as communication misses [2]). A cache coherence miss (from now simply *coherence miss*) is caused by the invalidation of a cache line by the cache coherence protocol. When the cache line is subsequently accessed, it will result in a coherence miss. Figure 2 illustrates how a coherence miss occurs. $Core_0$ and $Core_1$ share a data item X, $Core_0$ reads X at t_1 and reads it again (reuse of X) at t_2 where $t_2 > t_1$. Between these two accesses to X by $Core_0$, another core, $Core_1$, writes to X at t ($t_1 < t < t_2$). With an invalidation-based cache coherence protocol, $Core_1$'s write will cause an invalidation of the cache line containing X in $Core_0$'s private cache. The invalidated cache line causes $Core_0$'s reuse of X at t_2 to become a coherence miss.

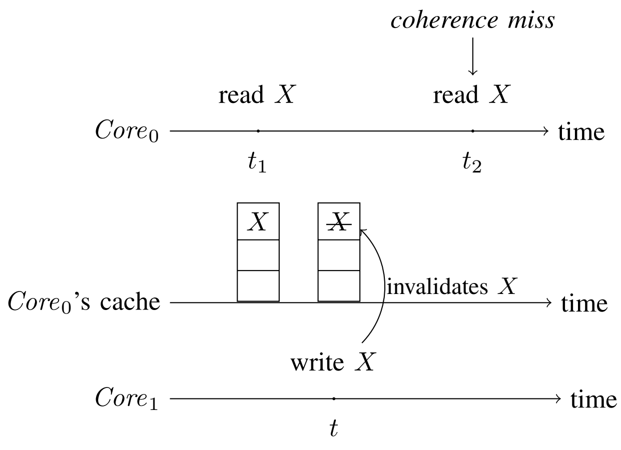

Fig. 2: Cache coherence miss example

Note that if X is evicted from $Core_0$'s private cache between t_1 and t_2 because of a cache access by $Core_0$, the access at t_2 will a capacity or conflict miss instead of a coherence miss. By convention, we classify such a miss as a capacity miss, even if the cache line is invalidated by another core before the evicting access by $Core_0$. Thus, invalidations of cache lines that would later anyway have been evicted for capacity or conflict reasons do not give rise to coherence misses. In other words, a necessary condition for being a coherence miss is that without the invalidation, the next access would have been a cache hit (in the private cache). This convention is consistent with previous work [2].

III. Notations

In this section, we introduce the assumptions, terms and notations used throughout this paper. We consider a multi-threaded program (target program) running with N threads. We use i to index the cores and threads. Each thread runs on a designated core ($Thread_i$ runs on $Core_i$). The terms and notations used in this model are shown in Table I.

Symbol	Description
Overall parameters	
N	number of running threads of the target application
C	private cache size (in terms of the number of cache lines)
$\mathbb{L}$	set of shared cache lines of the target application
Parameters of misses in the private cache	
$M_i(N)$	number of memory accesses by $Core_i$, running with N threads
$M_i^{miss}(N)$	number of private cache misses by $Core_i$, running with N threads
$M_i^{cold}(N)$	number of cold misses of $Core_i$, running with N threads
$M_i^{cap}(N)$	number of capacity misses of $Core_i$, running with N threads
$M_i^{conf}(N)$	number of conflict misses of $Core_i$, running with N threads
$M_i^{coh}(N)$	number of coherence misses by $Core_i$, running with N threads
$M_{i,X}(N)$	number of memory accesses to shared cache line X by $Core_i$, running with N threads

TABLE I: Table of notations

There are two kinds of notations we use in this paper: overall parameters (about the cache system and target application) and cache misses. The overall parameters include the number of running threads in the application, cache size and set of shared cache lines. We use M to denote the number of cache misses: $M_i^{cold}(N)$, $M_i^{cap}(N)$, $M_i^{conf}(N)$ and $M_i^{coh}(N)$ for the number of cold, capacity, conflict and coherence misses, respectively.

The total number of misses in the private cache of $Core_i$ is the sum of all four kinds of cache misses: cold misses, capacity misses, conflict misses and coherence misses:

$$M_i^{miss}(N) = M_i^{cold}(N) + M_i^{cap}(N) + M_i^{conf}(N) + M_i^{coh}(N)$$

IV. Modeling cold, capacity, and conflict misses

In this section, we review existing methods to model and predict the number of cold, capacity and conflict misses, which will be used in our evaluation.

A. Modeling cold misses

A cold miss in the private cache of $Core_i$ is triggered when a cache line is accessed for the first time by $Core_i$. In this paper, we therefore use the number of distinct cache lines accessed by $Core_i$ as an estimate of $M_i^{cold}(N)$. Note that some of the accessed cache lines may have been brought into the private cache by prefetching, and thus this estimate in general overapproximates $M_i^{cold}(N)$. However, we will still use this estimate (in the evaluation part) since it is difficult to estimate the effect of prefetching, and since cold misses are not the focus of this paper.

B. Modeling capacity and conflict misses

Both capacity and conflict misses are caused by eviction of cache lines. They differ in the cause. If a cache line is evicted because the cache is full, it causes a capacity miss. The eviction for conflict misses is due to the mapped set being full (in a non-fully-associative cache).

The number of capacity misses depends on a program's data locality, cache size and replacement policy. For a fully-associative cache with the Least Recently Used (LRU) replacement policy, *stack distance analysis* [9] [10] [11] is a common way to model the number of capacity misses. The stack distance is defined as the number of distinct cache lines accessed between the reuse of a cache line. For example, given

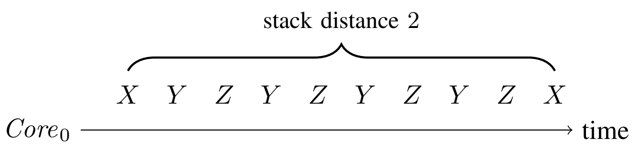

Fig. 3: Example memory access sequence: letters represent cache lines

a memory access sequence in Figure 3, the stack distance of the second access to X is two since there are two distinct cache lines accesses (Y and Z) in between the two accesses of X. In a fully-associative cache with cache size $C < 2$, X would have been evicted before it is reused, resulting in a capacity miss. If $C \geq 2$, the second access to X would be a capacity hit. In a fully-associative cache with the LRU replacement policy, the stack distance distribution can be estimated with the reuse distance distribution [3]. The advantage is that reuse distance distributions can be obtained efficiently by sampling, even on native executions, whereas obtaining exact stack distances requires exact recording of memory access sequences.

Conflict misses can be modeled with set stack distance [12]. The set stack distance of a memory access is the distinct number of cache lines accessed within the same set between the reuse of the memory access. In the previous example in Figure 3, assuming X and Y map to the same set, the set stack distance of the second access to X would be one. The sets are isolated and accesses to one set cannot evict data from another set. If the set stack distance is bigger than the set associativity, the memory access will result in a conflict miss. In a non-fully-associative cache, the conflict misses contain capacity misses. In the evaluation in Section VII, we adopt set stack distance analysis to calculate the number of conflict misses.

V. MODELING CACHE COHERENCE MISSES

So far we have discussed how to analyze the number of cold misses, capacity misses and conflict misses in the private cache. These three kinds of misses only depend on the data locality of the memory access to the core and the cache parameters (cache size, cache replacement policy and set associativity). Given a memory access sequence and cache parameters, the numbers of these misses are deterministic. However, the number of coherence misses is non-deterministic, since it depends on the pattern of interleaving accesses to shared data between cores. One way to analyze the coherence misses is to enumerate and model all possible interleavings, which is time consuming. In this section, we describe our two probabilistic models (uniform and phased) and a third model (symmetric) for estimating the number of coherence misses.

The first two models (uniform and phased) are based on the following reasoning. Assume that the target program is running on N cores. Consider a particular core $Core_i$ and a particular memory access x to a cache line X by $Core_i$. It follows from the description in Section II that in order for the memory access x to be a coherence miss, the following conditions must be satisfied:

1) x is not a cold miss, i.e., the access to X is a reuse.
2) x is not a capacity or conflict miss.
3) $Core_i$ shares cache line X with at least one other core, which may write to X.
4) Another core performs a write access to X before the access x, but after the previous access to X by $Core_i$.

The access x has a well-defined *reuse distance* in the sequence of memory accesses by $Core_i$, which is defined as the number of memory accesses between the last access to X by $Core_i$ and the access x. For example, in Figure 4, the last access to

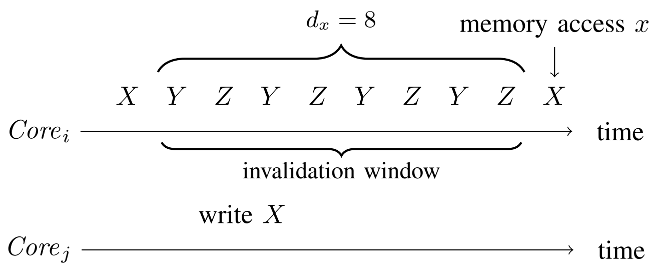

Fig. 4: Coherence miss example: $Core_i$ and $Core_j$ share a cache line X. $Core_i$ reuses X with reuse distances d_x.

cache line X by $Core_i$ has reuse distance 8. Let d_x denote the reuse distance of access x. Between its previous access to X and the access x, the core $Core_i$ leaves a window for other cores to perform write accesses to X that invalidate $Core_i$'s copy of X. The reuse distance d_x is a measure of the size of this window. Note that the relevant measure of window length is the reuse distance, not the stack distance, since it is not relevant whether the accesses in the window are to different cache lines or not.

Let us now consider all accesses to cache line X by $Core_i$ with some fixed reuse distance d. Let $P^{cap}_{i,d,X}(N)$ denote the probability that such an access (i.e., an access by $Core_i$ to cache line X with reuse distance d) is a capacity or conflict miss. Let $P^{inv}_{i,d,X}(N)$ denote the probability that another core performs a write access to X before such an access, but after the previous access to X by $Core_i$. We can then compute the probability that an access by $Core_i$ to cache line X with reuse distance d is a coherence miss, denoted $P^{coh}_{i,d,X}(N)$, as

$$P^{coh}_{i,d,X}(N) = P^{inv}_{i,d,X}(N)(1 - P^{cap}_{i,d,X}(N)) \tag{1}$$

The probability $P^{cap}_{i,d,X}(N)$ can be obtained by stack distance analysis. As described in Section IV-B, it can also be efficiently approximated from a reuse distance histogram that can be obtained by sampling native executions, using the technique of [3].

Let $P_{i,d,X}(N)$ denote the probability that when the application executes on N cores, an access by $Core_i$ accesses cache line X with reuse distance d. By summing over all reuse distances, we can obtain the probability $P^{coh}_{i,X}(N)$ that an access by $Core_i$ to cache line X is a coherence miss as

$$P^{coh}_{i,X}(N) = \sum_{d=1}^{\infty} P_{i,d,X}(N) P^{coh}_{i,d,X}(N) \tag{2}$$

Substituting $P^{coh}_{i,d,X}(N)$ by Eq. 1,

$$P^{coh}_{i,X}(N) = \sum_{d=1}^{\infty} P_{i,d,X}(N) P^{inv}_{i,d,X}(N)(1 - P^{cap}_{i,d,X}(N)) \quad (3)$$

Let $M_{i,X}(N)$ denote the total number of memory accesses to X by $Core_i$ (with N running threads), and let $\mathbb{L}$ be the set of cache lines that are shared between cores. The expected number of coherence misses for $Core_i$ is then

$$E^{coh}_i(N) = \sum_{X \in \mathbb{L}} (M_{i,X}(N) \cdot P^{coh}_{i,X}(N)) \quad (4)$$

Now the remaining problem is to calculate the probability $P^{inv}_{i,d,X}(N)$ of an invalidating access by another core during the reuse window. In this paper, we propose two models for this calculation, *uniform* and *phased*. The *uniform* model assumes that accesses by each thread to each data item are spread uniformly throughout the execution. The *phased* model works even without this restriction, by dividing the execution of the program into phases, and considering accesses in different phases separately.

A. The uniform model

This model assumes that we can approximate the accesses of each thread as occurring uniformly throughout the execution, and without correlation between threads. Such an assumption is valid for applications that access and update some common data in a way that is not tightly coordinated.To calculate $P^{inv}_{i,d,X}(N)$, assume that a core $Core_j$ different from $Core_i$ writes to cache line X with a frequency of $F^{write}_{j,i,X}(N)$ per memory access of core i. This frequency shows the number of write accesses to X by $Core_j$ for each memory access of $Core_i$. In many applications the frequency of memory accesses is approximately equal for all cores, and $F^{write}_{j,i,X}(N)$ does not depend on i.

For a reuse distance d by $Core_i$, the probability of $Core_j$ *not* writing to X within a reuse window of length d is $(1 - F^{write}_{j,i,X}(N))^d$. Since the memory accesses of different cores are assumed to be independent, the probability of at least one of the cores writing to X within d is

$$P^{inv}_{i,d,X}(N) = 1 - \prod_{j \neq i} (1 - F^{write}_{j,i,X}(N))^d \quad (5)$$

By substituting for $P^{inv}_{i,d,X}(N)$ Eq. (3) and for $P^{coh}_{i,X}(N)$ in Eq. (4), we obtain

$$E^{coh}_i(N) = \sum_{X \in \mathbb{L}} (M_{i,X}(N) \cdot P^{coh}_{i,X}(N)) \ , \quad (6)$$

where $P^{coh}_{i,X}(N))$ is

$$\sum_{d=1}^{\infty} P_{i,d,X}(N) \left[(1 - \prod_{j \neq i} (1 - F^{write}_{j,i,X}(N))^d)(1 - P^{cap}_{i,d,X}(N)) \right] \quad (7)$$

B. The phased model

The uniform model assumes that all shared cache lines are written uniformly by the cores throughout the execution. This is not the case in all real applications. Some applications show different patterns of shared data access in different parts of the execution, or coordinate access to shared data by synchronization. A change of phase is typically marked by synchronization primitives (e.g., conditional variables, barriers, etc). For example, the *bodytrack* benchmark in the PARSEC benchmark suite has a producer-consumer structure. The master threads (the main thread and the I/O thread) first generate the shared data by writing to it: thereafter they signal the worker threads (by signaling a conditional variable) to start processing the shared data. Although a significant amount of data is shared, the master and worker threads' accesses to shared data are separated by the signaling of the conditional variable. Such a separation prevents interleaved accesses to shared data by the master and worker threads. Without these interleaved accesses, the master threads' writes to the shared data do not cause any coherence misses in the worker threads.

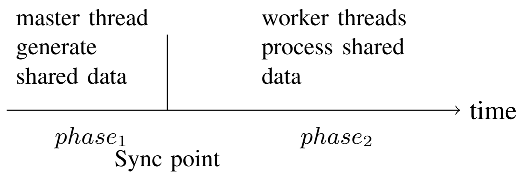

Fig. 5: The division of phases for *bodytrack*

The phased model generalizes the uniform model to handle programs with phase-dependent pattern of accesses to shared cache lines. The phased model divides the execution into phases, by letting synchronization primitives (conditional variable, barriers, etc) determine that a new phase is entered. There are two kinds of coherence misses with the phase division: intra-phase coherence misses where the coherence miss is caused by an invalidating write in the same phase as the two accesses of the core that experiences the coherence miss, and inter-phase coherence misses, where one of the accesses by the core with coherence miss occurs in a different phase from the invalidating write access (cf. Figure 6).

The intra-core coherence misses are analyzed using the uniform model in each phase. Inter-core coherence misses are analyzed by the phased model as follows. For each phase, we keep track of which shared cache lines are written to by each core. In addition, for each core we keep track of the distance d_f between each phase boundary and the first access to each cache line X, and symmetrically the distance d_l from the last access to each cache line X and the following phase boundary. The distribution of these distances are summarized in the probabilities $P^{first}_{i,d_f,X}(N)$ and $P^{last}_{i,d_l,X}(N)$, respectively, for each d_f and d_l.

For a target core $Core_i$, and for all pairs of phases $phase_{first}$ and $phase_{last}$ such that $Core_i$ does not access X in the phases between $phase_{first}$ and $phase_{last}$, the phased model predicts (i) one coherence miss if X is written to by another core in a phase between $phase_{first}$ and $phase_{last}$, and

(ii) a coherence miss with probability

$$\sum_{d_f}\sum_{d_l} P^{first}_{i,d_f,X}(N) P^{last}_{i,d_l,X}(N) \left[1 - \prod_{j\neq i}(1 - F^{write}_{j,i,X}(N))^{d_f+d_l} \right]$$

if X is not written to by another core between phases $phase_{first}$ or $phase_{last}$. This formula gives is the probability for an invalidating access in either $phase_{first}$ or $phase_{last}$. For simplicity, we have assumed that $F^{write}_{j,i,X}(N)$ is the same in both phases, otherwise the formula must be refined to consider two different frequencies in the obvious way. Since the probability in the second case is typically small, and can add at most one coherence miss, we often omit it in the analysis.

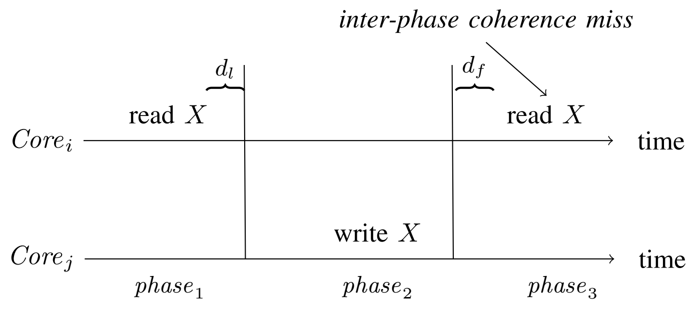

Fig. 6: Inter-phase coherence miss, phases divided by synchronization points

C. *The symmetric model*

For multi-threaded applications where threads access both local and global data in a symmetric way, we propose our third model, called the symmetric model. Applications to which this model can apply include network packet processing, streaming applications, etc. Due to the symmetry assumption, we can avoid applying uniform or phased, which needs information about reuse distance distributions and write frequencies to shared data. The symmetric model, in that it assumes that all threads can be treated symmetrically, needs less information about the behavior of each thread.

We consider programs that process a large set of input data items, which are evenly divided among threads. The threads maintain some shared data structure (e.g. a hash table) in order to manage the data items. All threads access the shared data structure randomly and independently, i.e., the threads have symmetric access patterns to the shared data. In addition, subsequent accesses to shared data are not correlated.

To present our analysis, let us introduce the following notations:

L	is the size (number of cache lines) of the input data to be processed. L only includes the part of the input data *not* having false-sharing effects with the shared data.
S	is the size of shared data
k_S	is the total number of accesses to S by all threads during the whole execution.

We make the following assumptions, which holds in typical applications. The input data size is much larger than the size of the shared data and cache size ($L >> S$ and $L >> C$).

Let us describe how the symmetric model models the different kinds of private cache misses.

Cold misses: When the application executes with only one thread, it accesses L local data and S shared data. Thus, the number of cold misses is $M^{cold}_1(1) = L+S$. When parallelized with N threads, the assumption on symmetry makes sure the input data is distributed evenly. Each thread processes $\frac{1}{N}$ of the total input data but all threads access the shared data: $M^{cold}_i(N) = \frac{1}{N}\cdot L + S$. Assuming $L >> S$,

$$M^{cold}_i(N) \approx \frac{1}{N} M^{cold}_1(1) \tag{8}$$

The point of this approximation is that we do not need to know L or S.

Capacity and conflict misses: Let k_L be all reuses to L that are cache misses, which means the accessed cache line have been evicted prior to the access. Since each access in k_L is a reuse of the non-shared data item, the eviction must be due to the limited cache capacity, which makes the access a capacity misses. Running with one thread, there are k_L capacity misses and with N threads, there are $\frac{1}{N}\cdot k_L$ misses for each thread due to the symmetry assumption.

The shared data may be reused. Let r be the miss ratio for the shared data running with one thread. When parallelized to N threads, due to the symmetry, each core has the same cache size and each thread accesses both the local and shared data in the same pattern as the one-core case. Thus the miss ratio r does not change with the number of threads. Therefore the number of capacity misses for the shared data is $k_S \cdot r$ for the one-thread case and $\frac{1}{N}\cdot k_S \cdot r$ for each thread in the N-thread case. To sum up, the total number of capacity misses is $k_L + k_S\cdot r$ and $\frac{1}{N}\cdot k_L + \frac{1}{N}\cdot k_S \cdot r$ for one thread and N threads respectively. Thus we have

$$M^{cap}_i(N) = \frac{1}{N} M^{cap}_i(1) \tag{9}$$

Similarly for the conflict misses:

$$M^{conf}_i(N) = \frac{1}{N} M^{conf}_i(1) \tag{10}$$

Coherence misses: To convert a $Core_i$'s potential cache hit into a coherence miss, the thread on a foreign core needs to write to the shared data before its reuse by $Core_i$. The probability of the last write to the shared data not being the same thread is $1 - \frac{1}{N}$. Then the probability of invalidating a shared data item for each thread is simply $P_{inv}(N) = (1 - \frac{1}{N}) \cdot F^{write}_{j,i,X}(N)$ in steady state.

After knowing the number of each kind of cache misses in the private cache, we can calculate the total number of cache misses. In the one-thread run, there are only cold, capacity and conflict misses:

$$M^{miss}_1(1) = M^{cold}_1(1) + M^{cap}_1(1) + M^{conf}_1(1) \tag{11}$$

If there are $M_1^{hit}(1)$ cache hits of shared data in the one-thread case, the probability of converting them into coherence misses is $P_{inv}(N)$. Thus we have

$$M_i^{coh}(N) = M_1^{hit}(1) \cdot P_{inv}(N) \tag{12}$$

and the total number of cache misses is the sum of the four different kinds of misses. Putting equations (8), (9), (10), (12), and (11) together, we get

$$M_i^{miss}(N) \approx \frac{1}{N} M_1^{miss}(1) + M_1^{hit}(1) \cdot P_{inv}(N) \tag{13}$$

Then we measure the number of cache misses for one thread and two threads to get $M_i^{miss}(2)$ and $M_1^{miss}(1)$. We solve equation (13) to get $M_1^{hit}(1)$:

$$M_1^{hit}(1) = \frac{M_i^{miss}(2) - \frac{1}{2} M_1^{miss}(1)}{P_{inv}(2)}$$

With $M_1^{hit}(1)$ and $M_1^{miss}(1)$ known, we now can predict the number of private cache misses for any number of threads with Eq. (13).

VI. Implementation

In this section, we describe how we have implemented our models, for the purpose of evaluating the applicability and accuracy of our models by comparing predictions obtained by the models with measurements from executions on actual hardware. The goal is to compare the modeled number of coherence misses in the private cache with the measured number. However, it is difficult to measure the number of coherence misses on actual hardware. One way would be to count the number of invalidation messages sent by the cache coherence protocol. However, not all invalidation messages will trigger future coherence misses. Therefore, using the number of invalidation messages as the number of coherence misses would be very inaccurate. However, the *total* number of private cache misses can be easily measured by reading the hardware performance counters. To compare with the measured total number of cache misses, we sum up the number of all four kinds of misses to estimate the total number of cache misses. Then this estimated value is compared with the measured one.

A. Trace the target program

We used Intel's PIN [7] tool to trace memory instructions. As a tool that introduces heavy profiling overhead (base overhead without any user-inserted instrumentation is 30% [13]), PIN does not guarantee to keep the traced program's thread interleaving as when the program runs on real hardware. Luckily, our models do not rely on capturing the exact pattern of interleaving of threads. We can even run the target application with PIN on a single core.

For each memory access, we record the *memory address*, *thread id*, *operation* (either memory read or write), and a time stamp which increments by 1 with each memory access. The time stamp shows the relative position of this memory access in all the memory accesses by the same thread. When a synchronization primitive (`pthread_cond_signal()` `pthread_cond_broadcast()`or `pthread_barrier()`) is encountered, the time stamps of all threads are recorded. This is used to divide the whole execution into phases for the phased model.

PIN records virtual addresses with the first few bits indicating the tag and index of the cache line and the last few bits indicating the offset in the cache line (Figure 7). In our target platform, each cache line is 64B, meaning the last 6 bits of the virtual address is the offset in the cache line. Truncating these bits gives an address identifying the cache line of the virtual address. This allows us to account for the coherence misses caused by both true and false sharing.

Tag	Index	Offset

Fig. 7: Segments of a virtual address

B. Extract input to model from trace

By profiling the target program, we get a trace of the memory accesses. Now we need to extract input to our models from this trace. uniform and phased need the following inputs for each kind of cache misses:

- Cold misses: the number of accessed cache lines
- Capacity and conflict misses: the set stack distance of all memory accesses.
- Coherence misses: reuse distance distribution for all shared cache lines, and other threads' write frequencies to the shared cache lines

First, we generate a set of all the distinct cache lines accesses by each core. By searching through all the sets, we find all the shared cache lines. The number of cold misses can be estimated by counting the number of distinct cache lines. The stack distance of each memory access is measured by counting the number of distinct memory accesses in the same set between the last and current access to the same cache line. If the set stack distance is bigger than the set associativity, the memory access is counted as a conflict miss. Otherwise, if the accessed cache line is a shared cache line, we record its reuse distance for the analysis for coherence misses.

To calculate the number of coherence misses for a target core $Core_i$, we need the reuse distance distribution of all shared cache lines by $Core_i$ and the other core's write frequency to all the shared cache lines. We collect all reuse distance for all the shared cache lines obtained from the previous step. Then we generate a reuse distance histogram. Next, we get the frequency of other cores (other than $Core_i$) write to X. For each $Core_j$ where $j \neq i$, the frequency of $Core_j$ writes to X is calculated as

$$F_{j,X}^{write}(N) = \frac{\#\ writes\ to\ X}{\#\ of\ total\ accesses\ of\ i}$$

This frequency is a relative frequency of $Core_j$ to $Core_i$ where $Core_i$ is our target core.

VII. Evaluation of our Models

A. Experiment setup

We evaluate our models on a 32-core system with 2.7GHz Intel Xeon E5-4650 CPUs. There are 4 sockets with 8 cores on each socket. The L2 cache is non-inclusive and non-exclusive of the L1 cache. A cache line is 64B. The experiment machine implements a MESIF coherence protocol[14].

Cache	Specification
L1	32KB, private, 8-way
L2	256KB, private, 8-way
L3	20MB, shared among cores on the same socket

TABLE II: The cache hierarchy of the experiment machine

B. Obtaining reference cache misses

All the multi-threaded benchmarks run with each thread pinned to a designated core. We read the hardware performance counter of each core for L2 misses with the Performance Application Programming (PAPI) library [15]. The performance counter starts right before a thread is created and stops right after a thread finishes (The Pthread library's `pthread_create()` function was overridden to control the counters). After collecting each core's L2 cache miss count, we take the average among all threads, which is used as a reference for the number of cache misses in the private cache.

C. Benchmarks

To validate our model, we chose 7 benchmarks (*blackscholes*, *bodytrack*, *fluidanimate*, *streamcluster*, *raytrace*, *swaptions* and *dedup*) from the PARSEC 3.0 benchmark suite. [1] Table III shows the input size and applied model for each benchmark. All benchmarks are compiled with gcc version 4.7.2 and optimization level O3. We ran all the benchmarks with $1-8$ thread(s), which utilizes all the cores on one socket. For *dedup*, which implements the pipeline parallelism, all threads in a parallel stage are assigned to the same socket.

benchmark	input	method
blackscholes	$4,096$ options	uniform
bodytrack	100 particles	uniform, phased
fluidanimate	$5,000$ particles	uniform
streamcluster	128 input points	uniform, phased
raytrace	teapot.env	uniform
swaptions	$10,000$ swaptions	uniform, phased
dedup	640MB random input data	symmetric

TABLE III: Benchmarks and inputs

D. Results (all benchmarks except for dedup)

Figure 8 shows the evaluation results for the uniform and phased models. Each vertical bar in the histogram represents the modeled number of private cache misses. It is further divided into four kinds of misses: capacity (including conflict) misses, cold misses, coherence misses and inter-phase coherence misses. The reference dots are the measured number of L2 misses. By comparing the number of different cache misses as the number of cores scales up, we can decide which kind of miss is the scalability bottleneck for the benchmarks.

Benchmarks analyzed with the uniform model: *blackscholes* and *fluidanimate* are analyzed with the uniform model. These two benchmarks do not have phase-wise behaviors in accessing the shared cache lines. The average relative error for uniform is 5.8%. In *blackscholes*, the number of coherence misses increases with more threads, which is due to the fact that the number of shared cache lines fluctuates in a small range (between 642 and 730) regardless of the number of threads. Sharing the same number of cache lines with more threads will increase the number of coherence misses. This can be shown in our model: the invalidation probability $P^{inv}_{i,d,X}(N)$ increases if there are more foreign cores write to shared data according to Eq. (5).

[1]We excluded benchmarks that do not compile, have no input, incompatible with the PIN framework and with no Pthread implementation.

Benchmarks analyzed with both the uniform and phased models: *bodytrack*, *streamcluster*, *raytrace* and *swaptions* are analyzed with the phased model combined with the uniform model. The run of *streamcluster* with one thread does not have any coherence misses. Taking out this instance run, the average relative error of all these benchmarks is 8.02%. The phased model divides *bodytrack*'s whole execution into two phases, where the master threads generate the shared data in $phase_1$ and the worker threads process the shared data in $phase_2$. All the coherence misses come from $phase_2$ since the accesses to shared data from $phase_1$ do not interleave with the shared data accesses in $phase_2$. The number of capacity misses decreases with more threads. This is because the amount of data for each thread decreases as there are more threads sharing the data to process. For *streamcluster*, the number of coherence misses dominates the total number of cache misses. The inter-phase coherence misses contribute to a noticeable part of the cache misses. This is due to the frequent barrier calls in the benchmark (over $19,000$ barriers). Most shared variables are used in multiple phases.

E. Applying symmetric to dedup

In this section, we take the *dedup* benchmark as an example to show how to use the symmetric model to predict the coherence misses. *dedup* implements pipeline parallelism (Figure 9 shows the structure, taken from [16]). There are five stages in this benchmark. The first and last stages are sequential and the other three stages are parallel. Adjacent stages share buffer queue(s) (Q1-Q8). The first stage Fragment splits the input data into coarse-grained chunks, then the FragmentRefine stage divides these chunks into finer-grained chunks. The Deduplicate stage processes the fine-grained chunks and inserts pointers to them into a hashtable. Then the Compress stage compresses the chunks. The last stage Reorder writes the compressed data into an output file. Our analysis focuses on the Deduplicate stage.

In the Deduplicate stage, all threads process their local data chunks and share a hash table. Each thread executes the following loop: 1) fetch a data chunk 2) generate a hash key for the data chunk and 3) search the hash key in the hash table 4) if the key is not found, insert the key into the hash table. There are some false-sharing effects between the local data and the shared data.

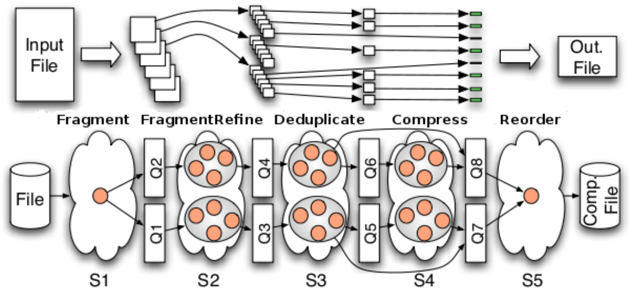

Fig. 9: Structure of the *dedup* benchmark: figure taken from [16] with updated stage names of PARSEC 3.0

By applying the symmetric model, we obtain the invalidation probabilities for $1-8$ threads, as shown in Table IV. Then we measure

threads	1	2	3	4	5	6	7	8
inv prob	0	0.5	0.67	0.75	0.8	0.83	0.86	0.88

TABLE IV: Analyzed invalidation probability for Deduplicate stage in *dedup*

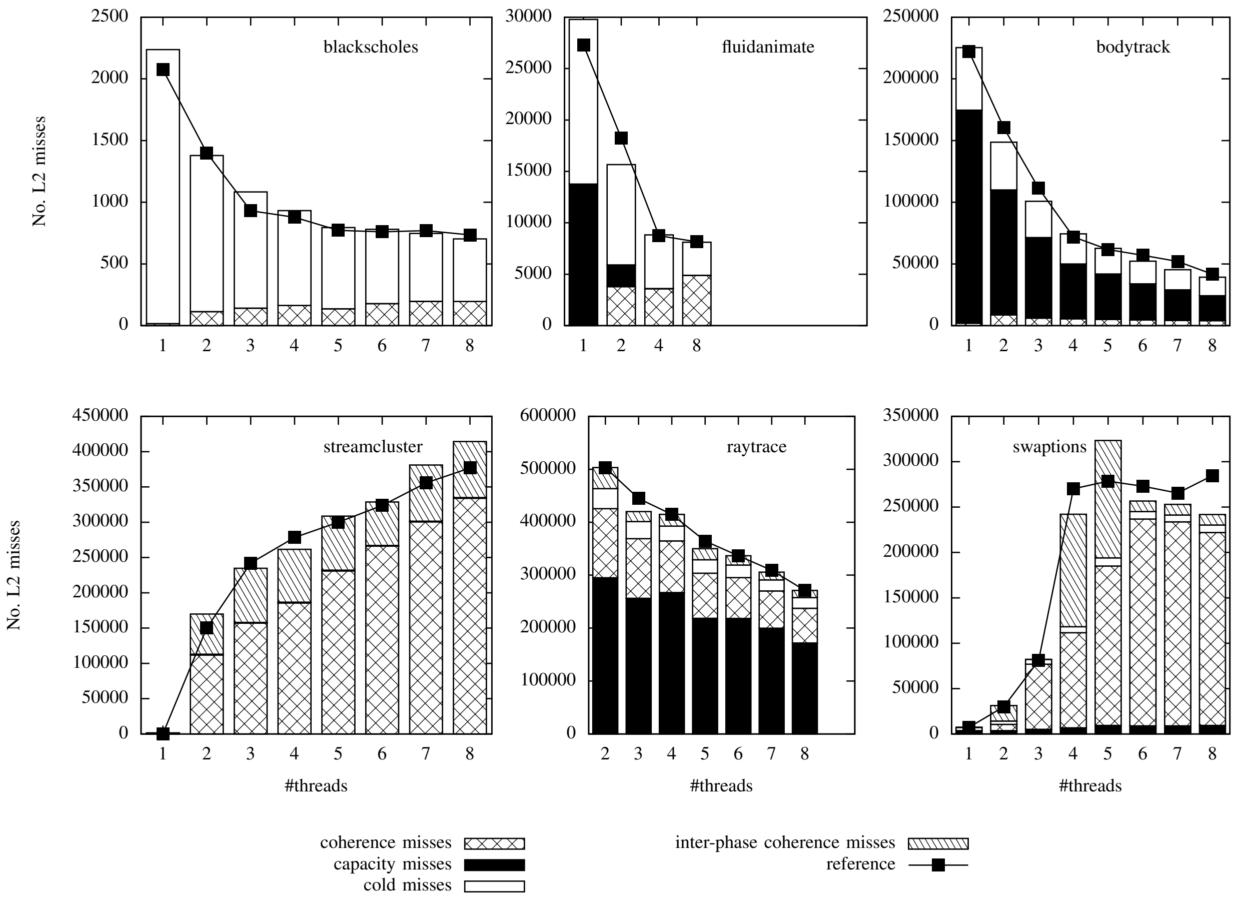

Fig. 8: Evaluation results

the number of private cache misses for 1 and 2 threads, solve Eq. (13) and use the equation to predict the number of private cache misses for $3-8$ threads (we can only evaluate the model with at most 8 threads since there are 8 cores on each socket). Figure 10 shows the prediction results. The average relative error of the symmetric model on *dedup* in predicting the number of L2 misses is 5.4%. We also did the same analysis on the private $L1$ cache and the relative error is 0.658%.

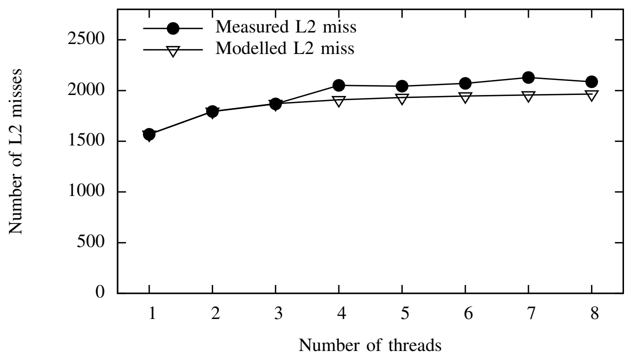

Fig. 10: Modeling the number of L2 misses for each loop of the Deduplicate stage in *dedup*

VIII. Related work

Stack distance: To quantify the data locality of a program, Mattson et al [17] introduced the stack distance as a measure to describe a program's data locality. It was used later by other researchers to analyze program locality [18] [19] [20] [21].

In the multicore era, Chandra et al [4], Xu et al [5] and Eklov et al [6] used the stack distance histogram to model cache contention in the shared cache. While predicting the cache contention effect accurately, none of these methods consider inter-core data sharing. Jiang et al [11] introduced the concurrent reuse distance (CRD) to consider interleaved memory accesses by all cores. Wu and Yeung [22] analyzed how the CRD changes with multiple threads. Dilation, spreading and distortion effects are observed in the CRD distribution compared to the non-concurrent reuse distance distribution. This observation is used to predict the CRD distribution with a number of threads.

Modeling coherence misses: So far the work taking coherence misses into account has been sampling-based. Schuff et al [10] presented a sampling-based approach to speed up detailed online simulations. It keeps track of the stack distance profile of a multi-threaded application and simulates the cache behavior with the profile. It models both the shared and private caches without distinguishing the kinds of cache misses. Each thread keeps a private stack and the cache line invalidation is propagated to other threads at synchronization points. Berg et al [2] also present a sample-based method to analyze the data locality of a multi-threaded program. To capture coherence misses, a cache line is monitored until it is reused by the same core. During the monitoring, it maintains a writer list to catch the foreign cores' write to the cache line. On reuse by the same core, a non-empty writer list means at least a core invalidated the cache line, which makes the reuse a coherence miss. Both of the sampling methods rely on capturing the exact thread interleavings. Sampling-based approaches are sensitive to interference from other processes. In addition, it cannot be used to model cache misses for another hardware configuration (e.g., different cache size). Our profiling approach only collects software-specific data, which makes our profiling process insensitive to interference. Another advantage of analytical-based approaches including ours is the ability to evaluate performance in another system's settings. For example, our model can be used to predict the cache misses with a different cache size.

Optimization with inter-core data sharing: Zhang et al [23] point out that the inter-core data reuse is not fully exploited by the current on-chip cache hierarchy or the state-of-the-art optimizations. An optimization scheme that balances the inter-core and intra-core data reuse is proposed. Demetriades et al [24] propose a run-time coherence miss prediction scheme. The scheme is based on the observation that the coherence misses and synchronization points in a program are usually correlated.

IX. Conclusion

In this paper, we proposed three new analytical models to analyze cache coherence misses for a multi-threaded application on multicore. The model builds on the observation that the occurence of a coherence miss is caused by a foreign write interleaving with the reuse of a shared cache line. The model quantifies the cache coherence misses of a core with the reuse distance distribution and the frequency of other cores's writing to the shared cache lines. The predicted cache coherence misses can then be added to the cold misses and capacity misses (calculated with existing methods) to model the total number of cache misses in the private cache. The model we proposed can be used to predict the private cache misses of a multi-threaded application for different cache sizes, to guide program optimizations in order to better utilize the private cache. We evaluated our models with a set of benchmarks in the PARSEC benchmark suite.

X. Acknowledgment

This work is supported by the Swedish Foundation for Strategic Research through CoDeR-MP, and by the Swedish Research Council through UPMARC. We are very grateful to Andreas Sembrant, Andreas Sandberg and Alexandra Jimborean for the valuable discussions. We would like to thank the anonymous reviewers for their helpful suggestions.

References

[1] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton, NJ, USA, 2011, aAI3445564.

[2] E. Berg, H. Zeffer, and E. Hagersten, "A statistical multiprocessor cache model." in *ISPASS*, 2006, pp. 89–99.

[3] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of lru caches," in *ISPASS*. IEEE Computer Society, 2010, pp. 55–65.

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture." in *HPCA*, 2005, pp. 340–351.

[5] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao, "Cache contention and application performance prediction for multi-core systems." in *ISPASS*, 2010, pp. 76–86.

[6] D. Eklov, D. Black-Schaffer, and E. Hagersten, "Fast modeling of shared caches in multicore systems." in *HiPEAC*, 2011, pp. 147–157.

[7] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation." in *PLDI*, 2005, pp. 190–200.

[8] M. D. Hill and A. J. Smith, "Evaluating associativity in cpu caches." 1989, pp. 1612–1630.

[9] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," in *In Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2001, pp. 617–662.

[10] D. L. Schuff, M. Kulkarni, and V. S. Pai, "Accelerating multicore reuse distance analysis with sampling and parallelization." in *PACT*, 2010, pp. 53–64.

[11] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *CC*, 2010, pp. 264–282.

[12] G. Balakrishnan and Y. Solihin, "West: Cloning data cache behavior using stochastic traces." in *HPCA*, 2012, pp. 387–398.

[13] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. M. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil, and A. Tal, "Analyzing parallel programs with pin." 2010, pp. 34–41.

[14] "The common system interface: Intels future interconnect." [Online]. Available: http://www.realworldtech.com/common-system-interface/5/

[15] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[16] A. G. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical modeling of pipeline parallelism." in *PACT*, 2009, pp. 281–290.

[17] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies." 1970, pp. 78–117.

[18] C. Cascaval and D. A. Padua, "Estimating cache misses and locality using stack distances." in *ICS*, 2003, pp. 150–159.

[19] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis." in *PLDI*, 2003, pp. 245–257.

[20] Y. Zhong, S. Dropsho, and C. Ding, "Miss rate prediction across all program inputs." in *IEEE PACT*, 2003, pp. 79–90.

[21] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance." 2009.

[22] M.-J. Wu and D. Yeung, "Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs." in *PACT*, 2011, pp. 264–275.

[23] Y. Zhang, M. T. Kandemir, and T. Yemliha, "Studying inter-core data reuse in multicores." in *SIGMETRICS*, 2011, pp. 25–36.

[24] S. Demetriades and S. Cho, "Predicting coherence communication by tracking synchronization points at run time." in *MICRO*, 2012, pp. 351–362.