Cache Contention and Application Performance Prediction for Multi-Core Systems

Chi Xu[*], Xi Chen[†], Robert P. Dick[†], and Zhuoqing Morley Mao[†]

[*]ECE Department
Univeristy of Minnesota
Minneapolis, MN 55455
xuchi@umn.edu

[†]EECS Department
University of Michigan
Ann Arbor, MI 48109
{chexi@, dickrp@eecs., zmao@eecs.}umich.edu

Abstract—The ongoing move to chip multiprocessors (CMPs) permits greater sharing of last-level cache by processor cores but this sharing aggravates the cache contention problem, potentially undermining performance improvements. Accurately modeling the impact of inter-process cache contention on performance and power consumption is required for optimized process assignment. However, techniques based on exhaustive consideration of process-to-processor mappings and cycle-accurate simulation are inefficient or intractable for CMPs, which often permit a large number of potential assignments. This paper proposes CAMP, a fast and accurate shared cache aware performance model for multi-core processors. CAMP estimates the performance degradation due to cache contention of processes running on CMPs. It uses reuse distance histograms, cache access frequencies, and the relationship between the throughput and cache miss rate of each process to predict its effective cache size when running concurrently and sharing cache with other processes, allowing instruction throughput estimation. We also provide an automated way to obtain process-dependent characteristics, such as reuse distance histograms, without offline simulation, operating system (OS) modification, or additional hardware. We tested the accuracy of CAMP using 55 different combinations of 10 SPEC CPU2000 benchmarks on a dual-core CMP machine. The average throughput prediction error was 1.57%.

I. INTRODUCTION

In recent chip multiprocessor (CMP) architectures, last-level caches are often shared among cores. This can improve performance by supporting on-chip inter-process communication and allowing heterogeneous allocation of cache to processes running on different cores. However, a process may cause the eviction of data belonging to other processes with which it shares cache space. This contention for shared cache space can cause simultaneously running processes to influence each other's performance. Moreover, the performance impact is non-uniform: it depends on the memory access behaviors of all processes with which it shares cache space.

The importance of inter-process cache contention for CMPs has been recognized in prior work [1], [2], [3]. However, the problem of predicting the impact of cache sharing on application performance during process assignment has been considered by only a few researchers [4], [5]. Knowing the performance implications of alternative assignment decisions can improve their quality. We therefore seek to build a cache contention model that permits fast and accurate performance prediction of processes on CMPs.

The construction of such a model should be easy and automatic; it should not require modifications to existing operating systems (OS) or hardware. Exhaustive offline simulation of process combinations is computationally intractable and should therefore be avoided. Moreover, prior work does not permit accurate prediction of the steady-state cache partition among arbitrary combinations of processes, which is a prerequisite for accurate performance prediction during assignment.

The paper describes a fast and accurate shared cache aware performance model for multi-core processors (called CAMP). This model uses non-linear equilibrium equations in a least-recently-used (LRU) or pseudo-LRU last-level cache, taking into account process reuse distance histograms, cache access frequencies, and miss rate aware performance degradation. CAMP models both cache miss rate and performance degradation as functions of process effective cache size, which in turn is a function of the memory access behavior of other processes sharing the cache. CAMP can be used to accurately predict the effective cache sizes of processes running simultaneously on CMPs, allowing performance prediction with an average error of only 1.57%. We also propose an easy-to-implement method of obtaining the reuse distance histogram of a process without offline simulation or modification to commodity hardware or OS. In contrast with existing techniques, the proposed technique uses only commonly available hardware performance counters. Finally, we evaluate the generality of CAMP by profiling processes on one CMP and using the resulting models to accurately predict process performance when run on two other CMPs having different cache sizes. All the measurements are performed on real processors.

76

The rest of this paper is organized as follows. Section II presents related work. Sections III and IV motivate and describe CAMP. Section V introduces an automated way to characterize process memory access behavior to permit later prediction of cache contention. Section VI presents and discusses the experimental validation process and results. Finally, Section VII summarizes our work.

II. RELATED WORK

Past work [6], [7], [8], [9] has considered the problem of adjusting cache partitioning during run time after process assignment decisions have already been made. In contrast, the goal of our work is to predict the performance implications of process assignment decisions before execution. Other researchers have developed performance prediction models to guide process assignment. However, most [10], [11] addressed cache contention only for uniprocessors on which only a single process may run at a time. The move to CMPs will aggravate the cache contention problem since multiple processes can run on different cores simultaneously.

Resource contention models for simultaneous multithreading (SMT) uniprocessors should be applicable to CMP systems due to the similarity in inter-process resource contention. However, existing work on resource contention modeling for SMT processors either suffers from large performance prediction error (20% of the predicted instruction throughput deviates by more than 20% from the actual instruction throughput) [12] or requires modifications to the underlying hardware [13]. To the best of our knowledge, existing performance models for SMT processors do not support accurate runtime performance prediction. Although the similarity of cache effects for CMPs and SMT processors suggests that the modeling technique described in this paper might also be accurate for SMT processors, we have not yet experimentally tested this hypothesis.

Researchers have also considered addressing the performance prediction problem using offline simulation [14] or modifications to the existing hardware or operating system [15]. For example, Suh et al. [8] proposed to add a hardware counter to each cache way and use them to determine the reuse distance histogram. Our goal in this work is runtime prediction of the performance of a process concurrently running on a shared-cache CMP, without requiring prior characterization.

Tam et al. [16] previously developed a technique to predict miss rate as a function of cache size by using built-in hardware performance counters, with a primary goal of supporting on-line optimization of cache partitioning among processes. They do not explain how to use miss rate curves to predict instruction throughputs for processes sharing cache space. Their approach relies on performance counters peculiar to the POWER5 architecture.

Chandra et al. [5] proposed three analytical models to predict miss rates for processes sharing the same cache. Their
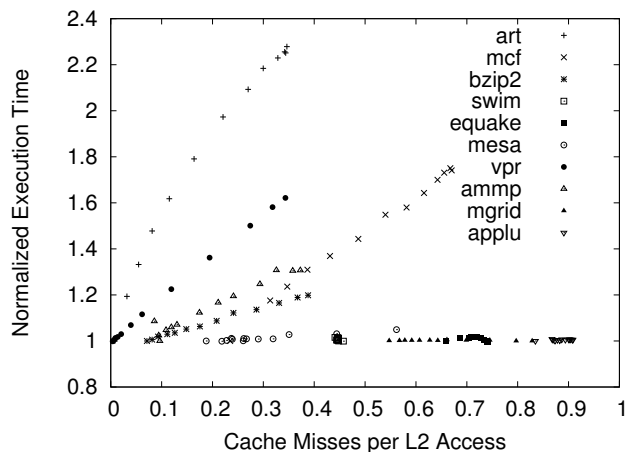

Figure 1. Impact of stressmark on performance of processes sharing case with it.

models use the reuse distances and/or circular sequence profiles for each thread to predict inter-thread cache contention. These models require knowledge of the steady-state L2 cache access frequency of a process when concurrently running with other processes. In reality, obtaining this information without running or simulating all potential combinations of concurrent cache-sharing processes is impractical.

Chen et al. [4] proposed a two-phase approach for performance prediction. In the first phase, the access frequency of a process running alone is used to estimate performance. In the second phase, the performance estimates from the first phase are refined to consider the implications of cache contention. The models proposed in each paper require processing circular memory access sequences, which must be obtained by tracing execution with an instruction-set simulator or non-standard detailed access tracing hardware.

III. MOTIVATION

Cache sharing among processes running on different cores of a CMP can hide inter-process communication latency and improve cache utilization. This improvement is undermined by cache contention among concurrently running processes. To illustrate this effect, we wrote a synthetic *stressmark* that accesses the last-level cache very frequently. The stressmark is intentionally designed to exhibit extreme memory access behavior, for use in characterization. The stressmark is run concurrently with the process under evaluation, on another core sharing the same cache. By varying the memory access behavior of the stressmark, we can change the number of last-level cache misses per cache access (MPA) for the stressmark, thereby controlling and measuring the performance impact on the other concurrently running process.

Figure 1 illustrates the relationship between the execution time, normalized to that when running the process alone, and MPA of the stressmark when it is run concurrently with each of 10 SPEC CPU2000 benchmarks. The relationship between MPA and execution time depends on the application. For example, with an MPA value of 0.35, the normalized
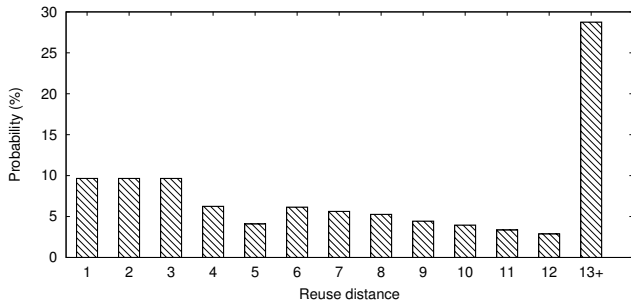
77

Figure 2. Cache line reuse distance histogram for *mcf* application.

execution time of *art* increased by 120% while that of *mesa* only increased by 1.5%. This demonstrates that the impact of cache contention on performance is application-dependent. Accurately predicting the performance and power consumption implications of assigning a particular set of processes to a CMP therefore requires a model that captures the variation in cache access and contention behavior among processes.

IV. ANALYTICAL MODEL

This section describes the main components in CAMP, namely its performance model, effective cache size estimation technique, and steady-state condition estimator.

IV.A. Background

In this section, we define some basic terms that will be used throughout this paper. Our study will consider a N-core processor with an L2 last-level on-chip cache. In the rest of paper, we refer to "L2 cache" as "cache". An set-associate cache is broken into *sets*, each of which has space for multiple *lines*, i.e., the minimal unit of data fetched by or evicted from a cache. The number of lines per set is the caches associativity, i.e., its number of *ways*. A line at a particular location in memory is associated with a set, and may be fetched into any line in the set.

Effective Cache Size: When multiple processes share a cache, they compete for limited space. The division of cache space among processes is influenced by characteristics of the concurrently running processes such as cache access frequency and sequential data access patterns. We define *effective cache size* of process i to be the average number of ways occupied by the process in a set, denoted as S_i. Therefore,

$$\sum_{i=1}^{N} S_i = A, \tag{1}$$

where N is the total number of processes sharing the cache and A is the number of ways in the cache. Note that S_i is a real value in our model because it represents the average number of ways process i occupies in a set during prolonged execution. If the cache access behavior of all processes is static, then S_i will be stable. We define this as the *steady-state* condition.

Reuse Distance: We define the *reuse distance*, R_j, of cache line j to be the number of distinct cache lines within the same set accessed between two consecutive accesses to line j. A *reuse distance histogram* represents the distribution of cache line reuse distances for an entire shared cache. Given an A-way set-associative cache, Figure 2 shows a reuse distance histogram for the *mcf* application (see Section VI). The x-axis shows the reuse distance and the y-axis shows the normalized frequencies of the associated reuse distances. The first bar in the histogram, i.e., $hist_1$, gives the probability that a most-recently-used line will be accessed again, while the last bar, i.e., $hist_{13+}$, gives the probability that the data for the next cache access does not exist in the most-recently-used 12 lines, which can be denoted as $\sum_{k=13}^{\infty} hist_k$. $Hist_\infty$ is the probability that the data in the line is never accessed again. Note that $hist_\infty$ can be very large for some streaming applications. For process i with an effective cache size of S_i, all accesses to the cache lines with a reuse distance larger than S_i result in cache misses. Hence, the probability of a cache access resulting in a miss for process i with an effective cache size of S_i, can be expressed as follows.

$$\text{MPA}_i(S_i) = \int_{S_i}^{\infty} \text{hist}_i(x)\, dx. \tag{2}$$

Note that $\text{hist}_i(x)$ is a continuous function derived using linear interpolation of the discrete histogram to support estimation for non-integer average reuse distances.

IV.B. Problem Formulation and Assumptions

The cache contention prediction problem can be formulated as follows: given N processes assigned to cores sharing the same A-way set-associative last-level cache, predict the steady-state cache size occupied by each process during concurrent execution. Note that the steady-state cache size can be directly translated to performance, as illustrated by Equation 2. Solving this problem is helpful for process assignment and migration in a CMP environment because it allows one to predict the consequences of tentative process assignment and migration decisions. However, accurate prediction of process performance is challenging because there are many combinations of processes that may share the same cache.

We make the following assumptions.

1) For each process, we assume that data accesses are uniformly distributed across all cache sets. The temporal cache access behaviors such as number of cache accesses per second (APS) and the reuse distance histogram (see Section IV-A) are assumed to be stationary. In the case of multiple non-repeating phases with distinct memory access patterns [17], non-repeating phases should be modeled separately.

2) We assume no hardware prefetching. Hardware prefetching complicates the model by predictively fetching cache lines based on access patterns. The model might therefore be inaccurate for systems using

78

prefetching. However, we argue that prefetching is of limited value on CMPs with constrained processor–memory bandwidth. For the 10 benchmarks used in this work, the average improvement was 3.25%, and only *equake* benefitted significantly.

3) We do not explicitly model the effect of kernel thread and instruction accesses on cache contention, but note that the resulting technique remains accurate in the presence of these accesses.

4) The cache uses an LRU replacement policy. Although most modern caches use pseudo-LRU policies, assuming LRU still permits high prediction accuracy.

Although these assumptions simplify the explanation of our analysis, we do not rely on them but instead "close the loop" by evaluating the resulting prediction technique on systems for which the assumptions may not hold. Finally, we consider a multi-programmed environment and therefore neglect communication among processes. Our analysis will hold for applications in which there is little communication among processes assigned to separate cores.

IV.C. Performance Model

The average number of cache accesses per second (APS) reflects how aggressively a process competes for cache space. All other things being equal, a process with high APS will generally take up more space in a shared cache than a process with low APS.

$$\text{APS} = \frac{\text{API}}{\text{SPI}}, \quad (3)$$

where API is the number of cache accesses per instruction and SPI is the number of seconds per instruction. API is a process property: given the same input data, the API of a process is fixed. On the other hand, SPI is largely affected by the number of cache misses per second (MPS). The latency per instruction, i.e., seconds per instruction, can be decomposed into two parts: on-chip latency due to computation and off-chip latency caused by main memory and disk accesses. When the CPU frequency remains constant, the on-chip latencies per instruction are approximately constant for a process. As shown in Figure 1 we experimentally determined that SPI can be expressed as a linear function of MPA.

$$\text{SPI} = \alpha \cdot \text{MPA} + \beta, \quad (4)$$

where α and β are parameters that can be obtained during offline characterization.

IV.D. Estimate Effective Cache Size After n Accesses

In this section, we use the reuse distance histogram of a process to derive its effective cache size. Consider the number of the distinct cache lines, s, (i.e., the effective cache size of the process) after n accesses in one set. Note that s is essentially effective cache size, S_i, as defined in Section IV-A. Given that $P_{s,n}$ is the probability of having s distinct cache lines after n consecutive cache accesses, $P_{hit,s}$ is the probability that a cache access will result in a cache

hit when the process already has s cache lines, and $P_{miss,s}$ is the probability that a cache access will result in a miss when the process has s cache lines, noting s can never be greater than n, the following recursive equation can be derived:

$$P_{s,n} = P_{s,n-1} \cdot P_{hit,s} + P_{s-1,n-1} \cdot P_{miss,s-1}, 1 < s \le n. \quad (5)$$

This can be explained as follows. The fact that n cache accesses result in an effective cache size of s can only be the result of one of the following scenarios.

1) In scenario A, the first $n-1$ cache accesses led to an effective cache size of s and the nth access resulted in a cache hit. Since the nth access did not lead to an increase in the effective cache size, it remains s. The probability of this scenario, $P(A)$, is $P_{s,n-1} \cdot P_{hit,s}$.

2) In scenario B, the first $n-1$ cache accesses lead to an effective cache size of $s-1$ and the nth access causes a cache miss. In this case, the effective cache size is increased by one, relative to the $s-1$ lines resulting from the first $n-1$ accesses. Thus, the effective cache size will be s after n cache accesses. The probability of this scenario, $P(B)$, is $P_{s-1,n-1} \cdot P_{miss,s-1}$.

Noting that $P_{s,n} = P(A) + P(B)$, we can derive Equation 5.

Given that $\text{MPA}(s)$ is the probability of a cache access missing, given an effective cache size of s, Equation 5 can be written as

$$P_{s,n} = P_{s,n-1} \cdot (1 - \text{MPA}(s)) + P_{s-1,n-1} \cdot \text{MPA}(s-1). \quad (6)$$

Note that $P_{1,1} = 1$ because the first cache access always uses a cache line and $1 < s \le n$. Assuming the process reaches steady state after n accesses, and given that $G_i(n)$ is the effective cache size for process i after n accesses, we have

$$G_i(n) = \sum_{s=1}^{n} (P_{s,n} \cdot s). \quad (7)$$

Note that $G_i(n)$ is a monotonically increasing function of n. Therefore, given the effective cache size of process i, S_i, we can deduce the number of cache accesses n needed for the process to reach steady state using the inverse function of $G_i(n)$, i.e., $n = G_i^{-1}(S_i)$.

IV.E. Steady-State Conditions

Given a cache with an LRU-like replacement policy, it is reasonable to assume that at time t, we can always find a duration T such that data accessed before time $t-T$ have been evicted and data accessed during $[t-T, t]$ are presently in the cache. To determine the effective cache size, we are only interested in data accessed during $[t-T, t]$. Since none of these accesses will evict any data lines accessed during $[t-T, t]$, it is as if the data were written to an empty cache with no cache misses during $[t-T, t]$, thus Equations 6 and 7 hold. Note that these accesses may still evict cache lines accessed before $t-T$. We assume the partition among processes resulting from data accesses from all co-running processes within $[t-T, t]$ is the same as that when all the

79

processes reach steady state. By computing the cache size of each process resulting from data accesses within $[t - T, t]$, we can determine process effective cache sizes. Hence, the effective cache size of process i, denoted as S_i, corresponds to the expected cache size determined by the most recent $\text{APS} \cdot T$ cache accesses for process i. Thus, the effective cache S_i is written as $G_i(\text{APS}_i \cdot T)$. Conversely, APS_i can be expressed as $G_i^{-1}(S_i)/T$. From Equation 3 and 4, we can derive the following equation:

$$\text{APS}_i = \frac{G_i^{-1}(S_i)}{T} = \frac{\text{API}_i}{\alpha_i \cdot \text{MPA}_i(S_i) + \beta_i}. \qquad (8)$$

Therefore,

$$T = \frac{G_i^{-1}(S_i) \cdot (\alpha_i \cdot \text{MPA}_i(S_i) + \beta_i)}{\text{API}_i}. \qquad (9)$$

Note that Equation 9 holds for any process i, where $i \in \{1, 2, \cdots, N\}$, given that N is the total number of processes. Combined with Equation 1, we have

$$\frac{G_1^{-1}(S_1)}{G_j^{-1}(S_j)} - \frac{\text{API}_1 \cdot (\alpha_j \text{MPA}_j(S_j) + \beta_j)}{\text{API}_j \cdot (\alpha_1 \text{MPA}_1(S_1) + \beta_1)} = 0, \forall_{j=1}^{N}, \quad (10)$$

$$\text{and } \sum_{i=1}^{N} S_i - A = 0, \qquad (11)$$

where $G_i^{-1}(S_i)$ and $\text{MPA}_i(S_i)$ are application-dependent non-linear functions of S_i. We solve Equation 10 using Newton–Raphson iteration, a standard numerical method for finding the roots of non-linear equations. Note that the number of ways in a cache (A) and number of cores (N) are each fewer than 10. $G_i^{-1}(S_i)$ and $\text{MPA}_i(S_i)$ are monotonic functions of S_i, so we can solve S_i for process i accurately within several iterations, where i ranges from 1 to N. The initial guess also affects the computational cost. In our experiments, we find that initially guessing that the effective cache size of a process i is proportional to its APS allows quick convergence to an accurate solution.

V. AUTOMATED PROFILING

In this section, we first explain how to obtain the reuse distance histogram of a process. We then describe how to derive other parameters such as API and MPA. After that, we give details about the automated profiling process. Finally, we indicate possible sources of prediction error.

V.A. Reuse Distance Profiling

Process reuse distance histograms play a central role in the proposed performance modeling technique. It would be possible to extract the reuse distance histograms of processes via simulation, and CAMP would dramatically improve estimation speed even if simulation were used for initial characterization; however, there is a faster alternative.

Most modern processors have built-in hardware performance counters (HPCs) that record information about architectural events such as the number of instructions retired, number of last-level cache accesses, and number of last-level cache misses [18]. Therefore, we can gather information about parameters such as SPI and MPA accurately. However, existing hardware or software resources do not directly provide reuse histogram data. We now explain the process of deriving reuse histogram data from directly monitored parameters.

Consider two processes running on separate cores sharing an A-way last-level cache. We assume if one process occupies l ways in a cache set, the concurrently running process will occupy $A - l$ ways. Based on Equation 2, we can compute the effective cache size of a *stressmark* with a controlled MPA and a known reuse distance histogram. We obtain the reuse distance histogram of a process (denoted as B) as follows. Run the stressmark along with B multiple times. In the lth run, we tune the parameters in the stressmark to change the effective cache size, denoted as $S_{stress,l}$. Record B's MPA in each run, denoted as $MPA_{B,l}$, where $l \in \{1, 2, \cdots, A\}$. Given that $S_{B,l}$ is process B's effective cache size in the lth run, and considering the lth and the $l + 1$st runs, we have

$$
\begin{aligned}
MPA_{B,l+1} &= \int_{S_{B,l+1}}^{\infty} \text{hist}_B(x)dx \text{ and} \\
MPA_{B,l} &= \int_{S_{B,l}}^{\infty} \text{hist}_B(x)dx. \qquad (12)
\end{aligned}
$$

See the discussion after Equation 2 for the definition of $\text{hist}(x)$. Hence, we can estimate the probability of process B having an effective cache size of $S_{B,l}$ as

$$\text{hist}_B(S_{B,l}) \approx MPA_{B,l+1} - MPA_{B,l}. \qquad (13)$$

By varying $S_{B,l}$ from 1 to A, we can estimate the probability at each effective cache size, thus allowing us to construct the reuse distance histogram. Since we can not control $S_{B,l}$ directly, in practice we adaptively tune the effective cache size of the stressmark from run to run. $S_{B,l} + S_{stress,l} = A$. Therefore, varying $S_{stress,l}$ changes $S_{B,l}$.

As indicated above, the stressmark should have the following properties.

1) High cache access frequency, i.e., high API. API is related to the degree to which a process competes for cache space. In order to estimate the probability of a process having a small effective cache size, the concurrently running stressmark should occupy a large portion of the cache with few cache misses.

2) A uniform reuse distance histogram, i.e., the probability is the same across all possible reuse distances. This makes it easy to compute the effective cache size given an MPA value. In addition, given a pseudo-LRU cache replacement policy, cache lines other than the least recently used will sometimes be evicted. Having a uniform reuse distance histogram minimizes the impact of this potential problem because the replacement noise will affect cache lines with all reuse distances equally.

Algorithm 1 Stressmark with k-Way Occupation

1: *Set* is the number of cache sets.
2: *Step* is the number of integers per cache line.
3: $S[Set \cdot Step \cdot k]$ is an array of integers.
4: $Index \leftarrow \{s_1, s_2, \cdots, s_n\}$
5: The following loop loads a predefined random sequence into *Index*.
6: **for** $j = 0 : n - 1$ **do**
7:    $flag \leftarrow Index[j]$
8:    $T \leftarrow \&S[flag \cdot Set \cdot Step]$
9:    **for** $i = 0 : Set - 1$ **do**
10:      read $T[i \cdot Step]$
11:    **end for**
12: **end for**

The pseudo-code of the stressmark is shown in Algorithm 1, where *Set* is the number of sets in the cache, *Step* is the number of integers per cache line. *Index[n]* is an integer array whose elements are uniformly distributed from $[1, k]$, which contains a random access location sequence. In order to maintain high cache access frequency for the stressmark, we pre-generate these arrays. Note that in Line 10 in Algorithm 1, two consecutive reads are *Step* elements apart to ensure an 100% L1 cache miss rate. Since the stressmark randomly accesses k cache lines within a cache set, the effective cache size of the stressmark is expected to be k. However, this may not be very accurate due to conflict misses between the stressmark and the process of interest. In reality, we use Equation 2 to estimate the effective cache size of the stressmark, i.e., $S_{stress} = \text{MPA}^{-1}(\text{MPA}_{stress})$, where MPA_{stress} is the MPA of the stressmark and $\text{MPA}^{-1}()$ is the inverse function for MPA in Equation 2 that converts MPA to an effective cache size, i.e., $\text{MPA}^{-1}(\text{MPA}(x)) = x$.

V.B. Automated Parameter Estimation

In this section, we describe how we calculate parameters such as API and SPI for a process. Given an A-way associative cache, in order to get the reuse distance histogram for a process, we run the stressmark concurrently with the process A times. In the lth run, we set k to l for the stressmark in Algorithm 1. Since API is fixed for a process with the same input data, given that API_l is the process's API in the lth run, the average API of the process can be estimated as

$$\text{API} = \frac{\sum_{l=1}^{A} API_l}{A}. \tag{14}$$

Similarly, we can get A pairs of a process's MPA and SPI values from the A runs. Given that MPA_l and SPI_l are the average MPA and SPI of the process in the lth run, the α and β in Equation 4 can be determined using linear regression, i.e.,

$$\alpha = \frac{A \cdot (\sum_{l=1}^{A} MPA_l \cdot SPI_l) - (\sum_{l=1}^{A} MPA_l)(\sum_{l=1}^{A} SPI_l)}{A \cdot (\sum_{l=1}^{A} MPA_l{}^2) - (\sum_{l=1}^{A} MPA_l)^2} \tag{15}$$

$$\text{and } \beta = \frac{(\sum_{l=1}^{A} SPI_l) - \alpha \cdot (\sum_{l=1}^{A} MPA_l)}{A}. \tag{16}$$

Note that most programs have repeating phases with periods ranging from 200 ms to 2,000 ms [17]. Numerous work exists on phase detection, i.e., finding the time at which the process switches from one phase to another. Since the process behavior is by definition similar within a phase, one set of parameters per phase is sufficient. In the rest of the paper, we will treat processes as having a single phase each to simplify explanation. Note that the proposed technique is also suitable for multi-phase processes, for which each phase may have a different set of extracted parameters.

Process characterization can be automated as follows. First, run the stressmark along with the process A times, varying the effective cache size. After A runs, API, α, β, and the reuse distance histogram can be estimated using Equations 13–16. These four parameters form the *feature vector* of a process. Given the feature vectors of two processes, we can predict their effective cache sizes when sharing cache, which in turn can be translated to SPI values using Equations 2 and 4. Note that the SPIs for the two processes are predicted without actually running them concurrently. Hence, given N processes for assignment to N cores, only N feature vectors are needed ($\mathcal{O}(N)$ complexity). These vectors can be used to predict the performance of any subset of the N processes during assignment ($2^N - 1$ combinations). Thus, the proposed technique is dramatically more efficient than one requiring simulation or execution of $2^N - 1$ combinations of processes.

V.C. Potential Sources of Error

There are two primary sources of error in the proposed technique: error in histogram estimation and error in linear regression analysis. We will explain these error sources now, but note that even with these error sources, the proposed technique is highly accurate (see Section VI).

When estimating the reuse distance histogram for a process, it is very difficult to capture the probability corresponding to a reuse distance close to 0 because the concurrently running stressmark cannot consume all of the cache space. Similarly, the estimation for a reuse distance close to A may also have some error. In practice, we assume a uniform distribution for reuse distances close to 0 or A. Linear interpolation, given an assumed miss rate of 1 at an effective cache size of zero, is used for very small effective cache sizes. In addition, the probability of reuse distances larger than A cannot be captured by our technique. Hence, we extrapolate this probability based on the derivative of the probability density function at a sample point close to A.

Error may also be introduced due to noise in sample parameters. When the <MPA, SPI> pairs gathered during profiling are clustered within a small region, linear regression may lead to inaccurate estimation of coefficients due to noise. We addressed this problem by bounding the step size during

81

Item	Specification
Number of chips	1
Number of cores per chip	2
Frequency	2.40 GHz
L1 ICache (Private)	32 KB, 64 B line, 8-way associative
L1 DCache (Private)	32 KB, 64 B line, 8-way associative
L2 Cache (Shared)	3 MB, 64 B line, 12-way associative

Newton–Raphson iteration when solving for the effective cache size (see Equation 10), permitting convergence.

VI. EVALUATION METHODOLOGY AND RESULTS

In this section, we first describe our experimental setup. We then present the experimental results for model validation. We contrast the proposed technique with other potential methods of predicting CMP cache contention among processes and indicate which features of the proposed approach permit high prediction accuracy.

VI.A. Experimental Setup

We evaluated our technique on a computer equipped with an Intel Core 2 Duo P8600 processor and the Mac OS X 10.5 operating system. The system parameters are listed in Table I. We used Shark, a built-in profiling tool, to sample performance counters at a period of 2 ms. The samples are used for calculating parameters (e.g., API, MPA, and SPI) on each core. We used the SPEC CPU2000 benchmark suite, which contains 26 benchmarks. Since validating all 351 pairwise combinations would be costly, we instead selected a subset containing five CPU-intensive and five memory-intensive benchmarks, and considered all pairwise combinations of these ten. We recorded the program phase information for each benchmark during pre-characterization. Experimental results indicate that all but two benchmarks have only one significant phase, as defined by our parameters of interest. The longest phases in *art* and *mcf* were used. We can thus address the prediction problem one phase at a time using phase detection algorithms, as described in Section V-B.

VI.B. Pre-Characterization

As indicated in Section V-B, we first run the stressmark concurrently with each benchmark on two different cores 12 times to derive various parameters such as API, MPA, and SPI. Each run lasts 10 s, which has proven sufficient for characterizing these parameters. Note that the working data set size of the stressmark is incremented by 1 way after each run to construct the reuse distance histogram for each benchmark, as described in Section V-A.

Analyzing API, α, and β: Hardware performance counter readings are analyzed to determine API, α, and β in Equations 3 and 4. Table II shows the value for each benchmark. API indicates an application's capability to compete for cache space. It also indicates whether an application is memory-intensive because higher API is usually associated with more misses per instruction, resulting in more off-chip memory

transactions. As indicated in Table II, benchmarks such as *art*, *mcf*, *vpr*, *swim*, and *ammp* are memory-intensive. Their APIs are significantly higher than those of the other benchmarks. α indicates sensitivity to cache misses in terms of performance. Equation 4 implies that for the same amount of change in MPA, a larger α indicates a larger change in SPI. As shown in Table II, the performance of memory-intensive applications tends to be more sensitive to cache misses than that of CPU-intensive applications, with *art* being the most cache-miss sensitive benchmark and *mgrid* being the least cache-miss sensitive benchmark. Note that α is negative for *swim*. This is because cache contention has little impact on this benchmark's MPA value, resulting in inaccurate estimation during linear regression when building the SPI model. However, this introduces little error in performance estimation because, as we show later in Figure 3, both MPA and SPI are insensitive to effective cache size for this benchmark.

Analyzing Cache Miss Rate: We use the approach explained in Section V-A to build the reuse distance histogram for each benchmark, which is then used to predict its cache miss rate as a function of effective cache size. Figure 3 illustrates the relationship between cache miss rate and effective cache size for each benchmark. The cache miss rate curves for benchmarks *bzip2* and *equake* are not shown because they are similar to that of *mgrid*. The results, from execution on hardware, are consistent with those obtained from simulation [19]. Note that linear approximation is used for leftmost segment of each miss rate curve, for the reasons given in Section V-C. However, for the benchmarks with high APIs such as *swim* and *applu*, the solutions of Equation 10 always lie outside this linear region. Therefore, we do not consider this region when analyzing the sensitivities of the cache miss rate curves for any benchmarks. As indicated in Figure 3, the cache miss rates of benchmarks such as *swim* and *applu* are insensitive to their effective cache sizes in the effective range. Therefore, their performance is only slightly affected when run together with other benchmarks. However, cache miss rates of benchmarks such as *art* and *vpr* are very sensitive to their effective cache sizes. Therefore, their performances will be significantly affected by cache contention, although the impact on their performances highly depends on the memory access patterns of the processes running concurrently with them. This indicates the importance of considering application behavior and cache contention during performance prediction on CMPs.

VI.C. Model Validation

In this section, we validate our technique by using the *feature vector*, i.e., $<API, \alpha, \beta>$, and reuse distance histogram of a benchmark to predict the performance when run concurrently with another benchmark. Note that feature vectors are determined during pre-characterization. We compare the performances of the two benchmarks during the evaluation period to the predicted performances using the feature vectors of the benchmarks. Note that the approach

TABLE II
API, α, AND β FOR DIFFERENT BENCHMARKS

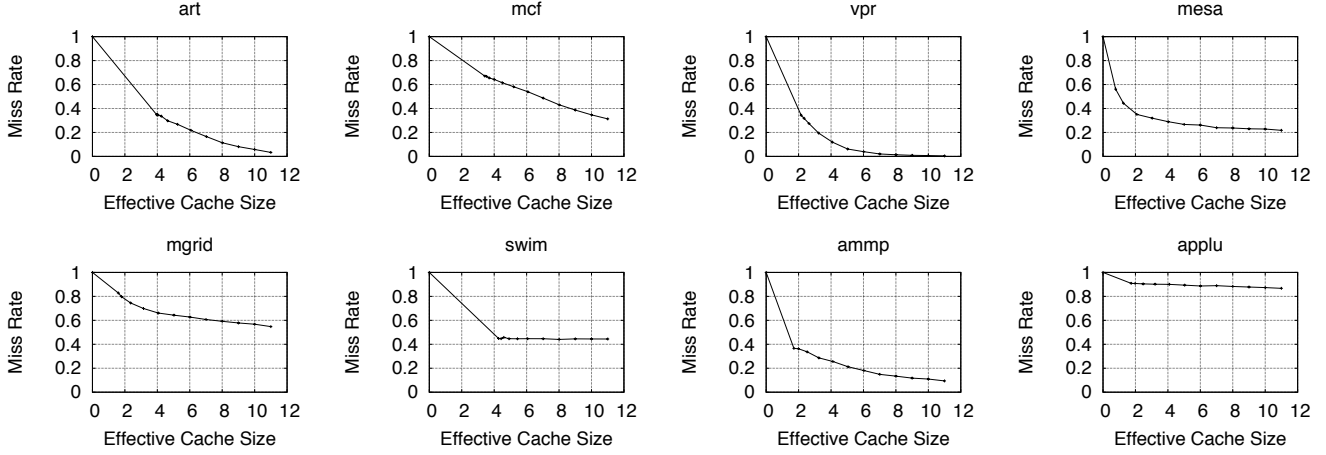| Benchmark	art	mcf	bzip2	swim	equake	mesa	vpr	ammp	mgrid	applu
API	0.0225	0.0733	0.0044	0.0116	0.0074	0.0013	0.0102	0.0092	0.0018	0.0018
α $(\times 10^{-9})$	446	134	99.9	-99.6	60.5	30.7	306	243	0.609	3.12
β $(\times 10^{-7})$	1.34	5.86	1.50	1.97	2.28	1.55	1.65	1.83	1.28	1.15

Figure 3. Profiled cache miss rate corresponding to effective cache size.

proposed by Chandra et al. [5] requires the steady-state cache access frequency of a process to be known a priori. We see no practical way to accurately predetermine this value for concurrently running processes. In contrast, our technique determines the steady-state cache access frequency using analysis of performance counter readings, i.e., the proposed technique works correctly using only inputs that are readily available in real systems.

In addition to the proposed technique, we considered and evaluated two alternatives. The first, called Accesses Based (AB), assumes the effective cache size of a process is proportional to APS. Given two processes running on two cores with effective cache sizes of S_1 and S_2, the formula to determine effective cache sizes can be written as

$$\frac{\mathrm{APS}_1}{\mathrm{APS}_2} = \frac{S_1}{S_2} = \frac{\mathrm{API}_1 \cdot (\alpha_2 \mathrm{MPA}_2(S_2) + \beta_2)}{\mathrm{API}_2 \cdot (\alpha_1 \mathrm{MPA}_1(S_1) + \beta_1)}. \quad (17)$$

Note that this model only considers APS. It may be inaccurate if the concurrently running processes have different MPAs or reuse frequencies. The second model, known as Misses Based (MB), assumes that S_i is proportional to MPS. Therefore, the equation used to determine S_1 and S_2 is

$$\frac{\mathrm{MPS}_1}{\mathrm{MPS}_2} = \frac{\mathrm{MPA}_1(S_1) \cdot \mathrm{API}_1 \cdot (\alpha_2 \mathrm{MPA}_2(S_2) + \beta_2)}{\mathrm{MPA}_2(S_2) \cdot \mathrm{API}_2 \cdot (\alpha_1 \mathrm{MPA}_1(S_1) + \beta_1)}. \quad (18)$$

The model only considers MPS. Thus it may be also inaccurate if the concurrently running processes have different reuse distance profiles.

Analysis of Results: We examined all 55 possible pairwise combinations of 10 benchmarks: each benchmark is paired with every other benchmark (including another instance of itself) and assigned to the two cache-sharing cores. The measured performance data are then compared to those predicted

by AB, MB, and CAMP. AB and MB are not past work. They are in fact alternative prediction models we considered.

Table III presents the average prediction error in cache miss rate and performance for each benchmark when run simultaneously with each of the 10 benchmarks. The first column lists the benchmarks. Columns 2, 6, and 10 show the average magnitudes of cache miss estimation error for CAMP, AB, and MB. Columns 3, 7, and 11 show the percentage of test cases with a cache miss estimation error larger than 5% among all 10 test cases. Similarly, Columns 4, 8, and 12 indicate the average relative estimation error in performance for the three techniques, while columns 5, 9, and 13 indicate the percentage of test cases with a relative performance estimation error larger than 5% among all 10 test cases for the three techniques. The last two rows correspond to the results for the 5 most memory-intensive benchmarks and all 10 benchmarks, respectively.

As indicated in Table III, CAMP has an average of 1.57% performance estimation error over all 10 benchmarks, compared to 3.07% for AB and 4.89% for MB. In addition, only 8% of the cases for CAMP have estimation errors greater than 5%, compared to 21% for AB and 33% for MB. Note that all three models have average performance estimation errors below 5%. This is mainly because all the three models are based on predicting the effective cache size of each benchmark when subject to cache sharing. If one of the two co-running benchmarks are CPU-intensive, e.g., *mesa*, *applu*, or *mgrid*, at least one of the two following conditions holds: (1) its cache miss rate is insensitive to its effective cache size or (2) its performance is insensitive to its cache miss rate. Therefore, the large cache miss estimation error may not be reflected in performance estimation error.

83

Benchmark	CAMP				AB				MB			
	MPA		SPI		MPA		SPI		MPA		SPI	
	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)	Error (%)	>5% (%)
art	1.61	0	3.68	40	4.60	50	10.26	80	5.88	70	18.09	90
vpr	0.88	0	1.48	0	4.70	40	7.67	60	5.89	30	9.24	50
mcf	2.10	10	3.70	20	2.82	10	3.97	40	6.79	40	7.72	70
ammp	2.82	20	3.04	20	4.03	30	4.16	30	5.89	60	6.78	90
bzip2	1.86	10	1.17	0	3.17	20	1.89	0	6.09	60	3.63	30
mesa	4.23	50	0.83	0	4.90	30	0.94	0	7.77	50	1.55	0
swim	0.28	0	0.86	0	0.23	0	0.81	0	0.27	0	0.78	0
equake	0.70	0	0.38	0	0.92	0	0.41	0	1.43	0	0.45	0
applu	1.13	0	0.32	0	0.86	0	0.31	0	1.79	10	0.33	0
mgrid	2.79	10	0.28	0	3.35	20	0.28	0	6.00	40	0.30	0
top 5 average	1.86	8	2.61	16	3.86	30	5.59	42	6.11	52	9.09	66
average	1.86	4	1.57	8	2.94	20	3.07	21	4.78	36	4.89	33

This also explains why memory-intensive benchmarks have larger estimation error than CPU-intensive benchmarks. In Table III, the bottom 5 benchmarks are either CPU-intensive applications or streaming applications with constant high miss rates, e.g., *swim*. Their performance estimation errors are below 1% for all three models. We thus also list the average performance estimation error for the top 5 benchmarks, which are relatively sensitive to the cache misses. CAMP has an average of 2.61% performance prediction error, compared to 5.59% for AB and 9.09% for MB.

Analyzing One Benchmark–Art: We now examine the accuracy of the three models when a specific benchmark, namely *art*, runs simultaneously with other benchmarks. Table IV presents the estimation error for MPA and SPI using CAMP, AB, and MB when *art* runs concurrently with each of the 10 benchmarks. The first column lists the benchmarks. Columns 2 and 3 present the increase in MPA and in SPI of each of the 10 benchmarks due to cache contention, compared to those when it runs alone. Column 4 shows the number of iterations required to solve for the effective cache size. Columns 5, 7, and 9 show the prediction errors for MPA for each of the three models. Columns 6, 8, and 10 show the prediction errors for SPI for each of the three models. The errors relative to measurements are reported. A positive error indicates an over-prediction and a negative error indicates an under-prediction. The last row shows the average results for all 10 cases.

Table IV indicates that CAMP outperforms AB and MB in terms of both MPA estimation error and SPI prediction error. AB over-predicts the effective cache size of *art*, resulting all 10 under-predictions of cache miss rate and 9 over-predictions of SPI. It achieves an average SPI prediction error of 10.26% and a maximum error of 17.47%. MB under-predicts the effective cache size of *art*, resulting in 8 over-predictions of MPS. It achieves an average SPI estimation error of 18.48%. and a maximum error of 41.06%. In contrast, CAMP achieves an average estimation error of 3.68% and a maximum error of 7.15%. Note that the computation overhead of CAMP is also lower than that of AB and MB

because it uses monotonic non-linear functions. This might significantly reduce computational cost when the number of cores is large. In addition, since the three models are based on estimating the effective cache sizes of two processes, they give the same results when two instances of *art* are running together, as indicated in the first row of Table IV.

Analyzing Three Scenarios: We now explain why AB usually leads to over-prediction and MB usually leads to under-prediction of the effective cache size. Figures 4–6 illustrate the predicted and measured normalized SPIs. The black portion shows the SPI when benchmark is run alone. Figure 4 shows the results when benchmarks *art* and *mcf* share cache in a dual-core system, with the left part corresponding to *art* and the right part corresponding to *mcf*. We denote this scenario as <*art, mcf*>. Similarly, Figure 5 represents <*art, vpr*>, and Figure 6 represents <*vpr, mcf*>. As indicated in Figures 4–6, CAMP achieves the best accuracy in all three cases. We take the left figure as an example to explain the reason for variation in accuracy. As indicated in Figure 3, given the same effective cache size, *mcf* has a higher miss rate than *art*, resulting in larger SPI than *art*. Therefore, the APS of *art* is approximately twice that of *mcf* when they run concurrently, even though the API of *mcf* is larger than that of *art*. Thus, *art* has a high APS with low MPS, which indicates that *art* can access the cache very frequently with low reuse distances, resulting in few misses. In this case, MB tends to over-predict the performance of *mcf* because it ignores factors such as APS. On the other hand, AB overestimates *mcf*'s performance due to ignoring its high reuse distances. Note that when two processes share the cache in a dual-core system, under-predicting the performance of one leads to over-predicting the performance of the other. CAMP takes both APS and MPS into consideration, and therefore is most accurate.

VI.D. Generality of Predictor For Different Machines

Figure 7 shows the cache miss rate of *art* corresponding to effective cache size profiled under two other cache configurations differing from that in Figure 3. CAMP was also validated on two other Intel Core 2 Duo Processors with

84

TABLE IV
MPA AND SPI PREDICTION WHEN PROCESSES RUN WITH ART

| Benchmark	Extra MPA	Extra SPI	CAMP			AB		MB	
			Iterations	MPA Error	SPI Error	MPA Error	SPI Error	MPA Error	SPI Error
	(%)	(%)		(%)	(%)	(%)	(%)	(%)	(%)
art	17.40	72.01	1	-1.96	+4.89	-1.96	+4.89	-1.96	+4.89
mcf	16.72	72.62	6	-1.52	+2.38	-7.16	+12.44	+13.60	-41.06
bzip2	6.13	31.48	5	+0.52	-0.13	-2.20	+6.82	+5.97	-17.71
swim	16.20	71.12	6	-4.12	+7.15	-9.35	+15.76	+6.58	-17.39
equake	10.92	48.03	8	+0.60	+0.19	-8.03	+17.47	+10.45	-31.18
mesa	2.33	13.93	4	-0.33	+5.60	-2.56	+11.50	-0.17	+5.18
vpr	8.41	42.24	5	+0.03	-0.66	-0.07	-0.41	+6.00	-18.72
ammp	5.42	32.84	5	-2.33	+4.45	-5.54	+11.80	+3.77	-13.48
mgrid	7.76	37.85	4	+2.17	-5.01	-3.29	+8.67	+5.26	-14.73
applu	9.40	44.74	6	+2.48	-6.38	-5.83	+12.79	+6.90	-20.46
average	10.07	46.69	5	1.61	3.68	4.60	10.26	6.07	18.48
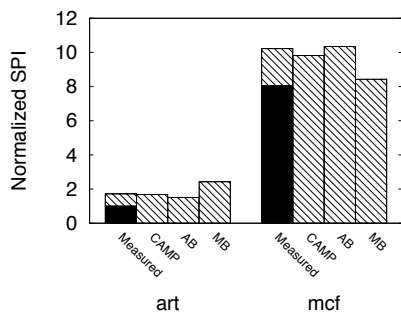
Figure 4. Performance degradation for <art, mcf> pair.
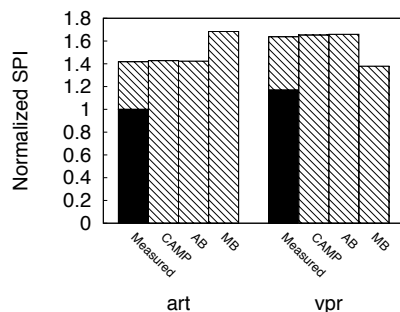

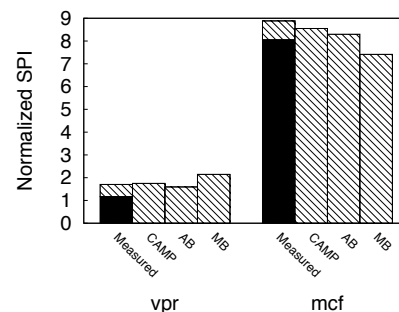Figure 5. Performance degradation for <art, vpr> pair.

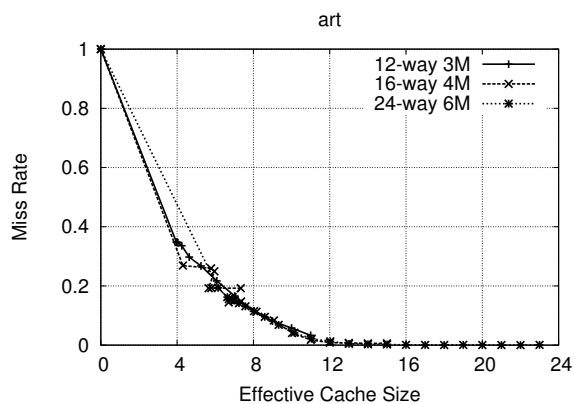Figure 6. Performance degradation for <vpr, mcf> pair.

Figure 7. Profiled cache miss rate corresponding to effective cache size for different cache configurations.

4 MB and 6 MB of L2 unified cache. The three cache miss rate curves closely match each other, suggesting that process characterization data derived on one machine might be used to accurately predict the performance of cache-sharing processes on different types of processors with different cache structures.

VII. CONCLUSION

Cache contention among processes running on different CMP cores heavily influences performance. A cache-contention aware assignment algorithm can help improve system throughput and reduce power consumption. However, this requires a model of cache contention behavior that can quickly and accurately determine the impact of different assignments on performance. This is challenging due to the large numbers of potential assignments of processes to CMPs. We have described CAMP, a predictive model that allows fast and accurate estimation of system performance degradation due to cache contention. More specifically, it first determines a process-dependent feature vector and reuse distance histogram via (potentially on-line) pre-characterization. The feature vectors of cache-sharing processes are supplied into a group of non-linear equations that determine the steady-state effective cache size and performance of each process. We also described a method to automate the profiling and performance prediction process. We evaluated the proposed technique on 55 different combinations of 10 SPEC CPU2000 benchmarks on a dual-core machine. The average performance prediction error is 1.57%. We also tested the generality of the proposed technique by profiling processes on one CMP and using the profiling information for performance prediction on two other CMPs with different cache sizes. In contrast with existing work, the proposed approach requires access only to infor-

85

mation that is readily available from processor performance counters.

REFERENCES

[1] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving performance isolation on chip multiprocessors via an operating system scheduler," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2007, pp. 25–38.

[2] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2006, pp. 13–22.

[3] T. Qiming, P. F. Sweeney, and E. Duesterwald, "Understanding the cost of thread migration for multi-threaded Java applications running on a multicore platform," in *Proc. Int. Conf. Performance Analysis of Systems and Software*, Apr. 2009, pp. 123–132.

[4] X. E. Chen and T. M. Aamodt, "A first-order fine-grained multithreaded throughput model," in *Proc. Int. Symp. High-Performance Computer Architecture*, Mar. 2009, pp. 329–340.

[5] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proc. Int. Symp. High-Performance Computer Architecture*, Feb. 2005, pp. 340–351.

[6] R. Iyer, "CQoS: A framework for enabling QoS in shared caches of CMP platforms," in *Proc. Annual International Conference on Supercomputing*, June 2004, pp. 257–266.

[7] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2004, pp. 111–122.

[8] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. Int. Symp. High-Performance Computer Architecture*, Feb. 2002, pp. 117–128.

[9] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. Int. Symp. Microarchitecture*, Dec. 2006.

[10] G. E. Suh, S. Devadas, and L. Rudolph, "Analytical cache models with applications to cache partitioning," in *Proc. Annual International Conference on Supercomputing*, June 2001, pp. 1–12.

[11] B. B. Fraguela, R. Doallo, and E. L. Zapata, "Automatic analytical modeling for the estimation of cache misses," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Oct. 1999, pp. 221–231.

[12] T. Moseley, J. L. Kihm, D. A. Connors, and D. Grunwald, "Methods for modeling resource contention on simultaneous multithreading processors," in *Proc. Int. Conf. Computer Design*, Oct. 2005, pp. 373–380.

[13] J. L. Kihm and D. A. Connors, "Implementation of fine-grained cache monitoring for improved SMT scheduling," in *Proc. Int. Conf. Computer Design*, Oct. 2004, pp. 326–331.

[14] "Dinero IV trace-driven uniprocessor cache simulator," http://www.cs.wisc.edu/~markhill/DineroIV.

[15] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, and D. Newell, "CacheScouts: Fine-grain monitoring of shared caches in CMP platforms," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sept. 2007, pp. 339–352.

[16] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm, "RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Mar. 2009, pp. 121–132.

[17] C. Isci, A. Buyuktosunoglu, and M. Martonosi, "Long-term workload phases: Duration predictions and applications to DVFS," *IEEE Micro*, no. 25, pp. 39–51, Oct. 2005.

[18] "Intel 64 and IA-32 Architectures Software Developer's Manual," http://www.intel.com/products/processor/manuals/.

[19] A. Kleinosowski, J. Flynn, N. Meares, and D. J. Lilja, "Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research," in *Proc. Int. Wkshp. Workload Characterization*, Sept. 2000, pp. 83–100.