# Online Cache Modeling for Commodity Multicore Processors

## VMware Technical Report VMware-TR-2010-002*

Richard West
Boston University
Boston, MA 02215 USA
richwest@cs.bu.edu

Puneet Zaroo
VMware, Inc.
Palo Alto, CA 94304 USA
puneetz@vmware.com

Carl A. Waldspurger
VMware, Inc.
Palo Alto, CA 94304 USA
carl@vmware.com

Xiao Zhang
University of Rochester
Rochester, NY 14627 USA
xiao@cs.rochester.edu

## ABSTRACT

Modern chip-level multiprocessors (CMPs) contain multiple processor cores sharing a common last-level cache, memory interconnects, and other hardware resources. Workloads running on separate cores compete for these resources, often resulting in highly-variable performance. It is generally desirable to co-schedule workloads that have minimal resource contention, in order to improve both performance and fairness. Unfortunately, commodity processors expose only limited information about the state of shared resources such as caches to the software responsible for scheduling workloads that execute concurrently. To make informed resource-management decisions, it is important to obtain accurate measurements of per-workload cache occupancies and their impact on performance, often summarized by utility functions such as miss-ratio curves (MRCs).

In this paper, we first introduce an efficient online technique for estimating the cache occupancy of individual software threads using only commonly-available hardware performance counters. We derive an analytical model as the basis of our occupancy estimation, and extend it for improved accuracy on modern cache configurations, considering the impact of set-associativity, line replacement policy, and memory locality effects. We demonstrate the effectiveness of occupancy estimation with a series of CMP simulations in which SPEC benchmarks execute concurrently on multiple cores. Leveraging our occupancy estimation technique, we also introduce a lightweight approach for online MRC construction, and demonstrate its effectiveness using a prototype implementation in the VMware ESX Server hypervisor. We present a series of experiments involving SPEC benchmarks, comparing the MRCs we construct online with MRCs generated offline in which various cache sizes are enforced via static page coloring.

## Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*measurements, modeling and prediction*

## General Terms

Measurement

## Keywords

Multicore processors, CMPs, shared cache resource management

## 1. INTRODUCTION

Advancements in processor architecture have led to a proliferation of multicore processors, commonly referred to as chip-level multiprocessors (CMPs). Commodity client and server platforms contain one or more CMPs, with each CMP consisting of multiple processor cores sharing a common last-level cache, memory interconnects, and other hardware resources [1, 12]. Workloads running on separate cores compete for these shared resources, often resulting in highly-variable or unpredictable performance [9, 14].

Operating systems and hypervisors are designed to multiplex hardware resources across multiple workloads with varying demands and importance. Unfortunately, commodity CMPs typically manage shared hardware resources, such as cache space and memory bandwidth, in a manner that is opaque to the software responsible for higher-level resource management. Without adequate visibility and control over performance-critical hardware resources, it is extremely difficult to optimize for efficient resource utilization or to enforce quality-of-service policies. For example, accurate measurement of workload cache occupancies is needed to make informed decisions about suitable cache partition sizes, and to enable co-scheduling decisions which can potentially reduce shared cache conflicts.

Many hardware-based resource management approaches have been proposed, including low-level architectural mechanisms to support cache occupancy monitoring and / or the ability to partition cache space and other memory system hardware among multiple workloads [2, 5, 7, 8, 13, 14, 17, 22, 23, 25, 27]. However, none of these techniques provide a method to accurately estimate workload cache occupancies on commodity processors, which expose only limited information to software. To further understand the impact of shared caches on workload performance, methods have also been devised to construct cache utility functions, such as miss-ratio curves (MRCs), which capture miss ratios at different cache occupancies [3, 21, 26, 27, 28]. However, existing techniques for generating MRCs either require custom hardware support, or incur non-trivial software overheads.

Both an accurate method of per-workload cache occupancy estimation and a generic approach to constructing cache utility curves are important for effective cache management. These techniques provide the basis for more efficient cache usage, allowing higher-level resource management policies to provide differential quality of service for workloads. For example, schedulers can exploit cache

VMware Technical Report VMware-TR-2010-002, July 2010. Also appears as Boston University BUCS-TR-2010-15.

performance information to make better co-runner placement decisions [4, 26, 29, 33], improving cache efficiency or fairness. Unfortunately, strict quality-of-service enforcement generally requires hardware support. While software-based page-coloring techniques have been used to provide isolation [6, 15, 16], such hard partitioning is inflexible, and generally prevents efficient cache utilization. Moreover, without special hardware support [24], dynamically recoloring a page is expensive, requiring updates to page mappings and a full page copy, making this approach unattractive for dynamic workload mixes in general-purpose systems.

We focus on several cache modeling techniques that are useful for more informed resource management decisions, such as co-runner selection. This paper does not propose specific policies for improved cache-aware scheduling decisions, but instead lays the foundations for such software-based performance management operations. We identify two key contributions: first, an efficient online technique for per-thread cache occupancy estimation and, second, an online technique for identifying the utility or performance benefits to each thread as a function of cache occupancy. In the first case, we leverage only commonly available hardware performance counters, found on most commodity CMPs in use today. We demonstrate the accuracy of our online cache occupancy estimations using Intel's CMPSched$im simulator [19], applied to sets of co-running SPEC benchmarks. We also extend the basic occupancy estimation model with cache-hit information, to more accurately reflect factors such as execution locality, hardware cache-line replacement policies, and set-associativity. Leveraging our online occupancy estimation technique, we show how to construct miss-ratio curves efficiently online, as threads compete dynamically for shared hardware resources. We demonstrate the effectiveness of online MRC construction by presenting experimental results using a prototype implementation in VMware's ESX Server hypervisor [30].

The next section presents our cache occupancy estimation approach, including a detailed description of its mathematical basis, together with simulation results demonstrating its effectiveness. Section 3 builds on this foundation, introducing our method for online construction of cache utility curves. Using a prototype implementation in the VMware ESX Server hypervisor, we examine the accuracy of our online MRCs by comparing them with MRCs for the same workloads collected via static page coloring. Related work is examined in Section 4. Finally, we summarize our conclusions and highlight opportunities for future work in Section 5.

## 2. CACHE OCCUPANCY ESTIMATION

In this section, we present our approach for estimating cache occupancy. We begin with a formal explanation of our basic model, which requires only cache miss counts for each co-running thread. We then examine the effects of pseudo-LRU set-associativity as implemented in modern processors, and extend our model to additionally incorporate cache hit counts to improve accuracy for such configurations.

We demonstrate the effectiveness of our cache occupancy estimation techniques with a series of experiments in which SPEC benchmarks execute concurrently on multiple cores. Since real processors do not expose the contents of hardware caches to software[1], we measure accuracy using the Intel CMPSched$im simulator [19] to compare the results of our model with actual cache occupancies in several different configurations.

For the purposes of our model, we consider a shared last-level cache that may be direct-mapped or $n$-way set associative. Our objective is to determine the current amount of cache space occupied by some thread, $\tau$, at time $t$, given contention for cache lines by multiple threads running on all the cores that share that cache. At time $t$, thread $\tau$ may be descheduled, or it may be actively executing on one core while other threads are active on the remaining cores.

### 2.1 Basic Cache Model

Since hardware caches reveal very little information to software, in order to derive quantitative information about their state, we must rely on inference techniques using features such as hardware performance counters. Virtually all modern processors provide performance counters through which information about various system events can be determined, such as instructions retired, cache misses, cache accesses and cycle times for execution sequences. Using two events, namely the *local* and *global* last-level cache misses, we estimate the number of cache lines, $E$, occupied by $\tau$ at time $t$. By global cache misses, we mean the cumulative number of such events across all cores that share the same last-level cache.

We assume that the shared cache is accessed uniformly at random. Results show this to be a reasonable assumption, given the unbiased nature of memory allocation, and the desire for all cache lines to be used effectively across multiple workloads and execution phases. We also assume each cache line is allocated to a single thread at any point in time. Data sharing is not considered in this paper, although it is part of our ongoing work in this area.

Cache occupancy is effectively dictated by the number of misses experienced by a thread because cache lines are allocated in response to such misses. Essentially, the current execution phase of a thread $\tau_i$ influences its cache investment, because any of its lines that it no longer accesses may be evicted by conflicting accesses to the same cache index by other threads. Evicted lines no longer relevant to the current execution phase of $\tau_i$ will not incur subsequent misses that would cause them to return to the cache. Hence, the cache occupancy of a thread is a function of its misses experienced over some interval of time.

For subsequent discussion, we introduce the following notation:

- Let $C$ represent the number of cache lines in a shared cache, accessed uniformly at random.
- Let $m_l$ represent the number of misses experienced by the *local* thread, $\tau_l$, under observation over some sampling interval. This term also represents the number of cache lines allocated due to misses.
- Let $m_o$ represent the aggregate number of misses by every thread *other* than $\tau_l$, on all cores of a CMP that cause cache lines to be allocated in response to such misses. We use the notation $\tau_o$ to represent the aggregate behavior of all other threads, treating it as if it were a single thread.

**Theorem.** *Consider a cache of size $C$ lines, with $E$ cache lines belonging to $\tau_l$ and $C - E$ cache lines belonging to $\tau_o$ at some time, $t$. If, in some interval, $\delta t$, there are $m_l$ misses corresponding to $\tau_l$ and $m_o$ misses corresponding to $\tau_o$, then the expected occupancy of $\tau_l$ at time $t + \delta t$ is:* $E' = E + (1 - \frac{E}{C}) \cdot m_l - \frac{E}{C} \cdot m_o$

[1] Current processor families do not allow software to inspect cache tags, although the MIPS R4000 [10] did provide a `cache` instruction with this capability.

**Proof.** First, at time $t$, it is assumed that $\tau_l$ and $\tau_o$ are sufficiently memory-intensive, and have executed for enough time, to collectively populate the entire cache. Now, considering any single cache line, $i$, at time $t + \delta t$ we have:

$$\begin{aligned} &Pr\{i\ belongs\ to\ \tau_l\} \ = \\ &Pr\{i\ belongs\ to\ \tau_l \mid i\ belonged\ to\ \tau_l\}{\cdot}Pr\{i\ belonged\ to\ \tau_l\} + \\ &Pr\{i\ belongs\ to\ \tau_l \mid i\ belonged\ to\ \tau_o\}{\cdot}Pr\{i\ belonged\ to\ \tau_o\} \end{aligned}$$

This follows from the prior probabilities, at time $t$:

$$Pr\{i\ belonged\ to\ \tau_l\} \quad = \quad \frac{E}{C} \tag{1}$$

$$Pr\{i\ belonged\ to\ \tau_o\} \quad = \quad 1 - \frac{E}{C} \tag{2}$$

Additionally, after $m_l + m_o$ misses, the probability that $\tau_l$ replaces line $i$, previously occupied by $\tau_o$, is one minus the probability that $\tau_l$ does not replace $\tau_o$ after $m_l + m_o$ misses. More formally,

$$\begin{aligned} &Pr\{\tau_l\ replaces\ \tau_o\ on\ line\ i\} = \\ &1 - [1 - \frac{m_l}{C(m_l + m_o)}]^{(m_l+m_o)} \end{aligned} \tag{3}$$

In Equation 3, $\frac{m_l}{C(m_l+m_o)}$ represents the probability that a miss by $\tau_l$ will result in an arbitrary line, $i$, being populated by contents for $\tau_l$. We know that the probability of a particular line being replaced by a single miss is $1/C$, and the ratio $\frac{m_l}{m_l+m_o}$ corresponds to the probability of that miss being caused by one of $\tau_l$'s accesses.

It follows from Equation 3 that the probability of $\tau_o$ replacing $\tau_l$ on line $i$ at the end of $m_l + m_o$ misses is:

$$\begin{aligned} &Pr\{\tau_o\ replaces\ \tau_l\ on\ line\ i\} = \\ &1 - [1 - \frac{m_o}{C(m_l + m_o)}]^{(m_l+m_o)} \end{aligned} \tag{4}$$

Therefore,

$$\begin{aligned} &Pr\{i\ belongs\ to\ \tau_l \mid i\ belonged\ to\ \tau_l\} \ = \\ &\quad 1 - Pr\{\tau_o\ replaces\ \tau_l\ on\ line\ i\} = \\ &\quad\quad [1 - \frac{m_o}{C(m_l + m_o)}]^{(m_l+m_o)} \end{aligned} \tag{5}$$

$$\begin{aligned} &Pr\{i\ belongs\ to\ \tau_l \mid i\ belonged\ to\ \tau_o\} \ = \\ &\quad Pr\{\tau_l\ replaces\ \tau_o\ on\ line\ i\} = \\ &\quad\quad 1 - [1 - \frac{m_l}{C(m_l + m_o)}]^{(m_l+m_o)} \end{aligned} \tag{6}$$

From Equations 1, 2, 5 and 6, we have:

$$\begin{aligned} Pr\{i\ belongs\ to\ \tau_l\} \ &= \frac{E}{C}{\cdot}[1 - \frac{m_o}{C(m_l + m_o)}]^{(m_l+m_o)} + \\ &(1 - \frac{E}{C}){\cdot}[1 - [1 - \frac{m_l}{C(m_l + m_o)}]^{(m_l+m_o)}] \end{aligned} \tag{7}$$

Ignoring the effects of quadratic and higher-degree terms, the first-degree linear approximation of Equation 7 becomes:

$$\begin{aligned} &Pr\{i\ belongs\ to\ \tau_l\} \ = \\ &E/C(1 - m_o/C) + (1 - E/C)m_l/C \end{aligned} \tag{8}$$

This is a reasonable approximation given that $1/C$ is small. Consequently, the expected number of cache lines, $E'$, belonging to $\tau_l$ at time $t + \delta t$ is:

$$\begin{aligned} &E' = E(1 - m_o/C) + (1 - E/C)m_l = \\ &\quad E + (1 - \frac{E}{C}){\cdot}m_l - \frac{E}{C}{\cdot}m_o \end{aligned} \tag{9}$$

This follows from Equation 8 by considering the state of each of the $C$ cache lines as independent of all others. □

Observe that the recurrence relation in Equation 9 captures the changes in cache occupancy for some thread over a given interval of time, with known local and global misses. The terms $[1 - \frac{m_o}{C(m_l+m_o)}]^{(m_l+m_o)}$ and $[1 - \frac{m_l}{C(m_l+m_o)}]^{(m_l+m_o)}$ in Equation 7, approximate to $e^{-m_o/C}$ and $e^{-m_l/C}$, respectively. Thus, for situations where $m_l + m_o >> 1$, Equation 9 becomes

$$E' = Ee^{-m_o/C} + C(1 - E/C)(1 - e^{-m_l/C}) \tag{10}$$

Equation 10 is significant in that it shows the cache occupancy of a thread (here, $\tau_l$) mimics the charge on an electrical capacitor. Given some initial occupancy, $E$, a growth rate proportional to $(1 - e^{-m_l/C})$ applies to lines currently unoccupied by $\tau_l$. Similarly, the rate of reduction in occupancy (*i.e.*, the equivalent discharge rate in a capacitor) is proportional to $e^{-m_o/C}$.

The linear model in Equation 9 is practical for online occupancy estimation, since it consists of an inexpensive computation that requires only the ability to measure per-core and per-CMP cache misses, which is provided by most modern processor architectures. For example, in the Intel Core architecture [11] used for our experiments in Section 3, the performance counter event `L2_LINES_IN` represents lines allocated in the L2 cache, in response to both on-demand and prefetch misses. A mask can be used to specify whether to count misses on a single core or on both cores sharing the cache.

## 2.2 Extended Cache Model for LRU Replacement Policies

So far, our analysis has assumed that each line of the cache is equally likely to be accessed. Over the lifetime of a large set of threads, this is a reasonable assumption. However, commodity CMP configurations feature $n$-way set associative caches, and lines within sets are not usually replaced randomly. Rather, victim lines are typically selected using some approximation to a least recently used (LRU) replacement policy. We modified Equation 9 and to additionally incorporate cache *hit* information, modeling the reduced replacement probability due to LRU effects when lines are reused. Equation 9 can be rewritten as

$$E' = E(1 - m_o p_l) + (C - E)m_l p_o \tag{11}$$

where $p_l$ is the probability that a miss falls on a line belonging to $\tau_l$, and $p_o$ is the probability that a miss falls on a line belonging to $\tau_o$. Since Equation 9 does not model LRU effects, each line is equally likely to be replaced and $p_l = p_o = 1/C$. In order to model LRU effects, we calculate

$$r_l \quad = \quad (h_l + m_l)/E \tag{12}$$

$$r_o \quad = \quad (h_o + m_o)/(C - E) \tag{13}$$

to quantify the frequency of reuse of the cache lines of $\tau_l$ and $\tau_o$, respectively. $h_l$ and $h_o$ represent the number of cache hits experienced by $\tau_l$ and $\tau_o$, respectively, in the measurement interval. As with miss counts, these hit counts can be obtained using hardware performance counters available on most modern processors.

When the cache replacement policy is an LRU variant, $r_o$ and $r_l$ approximate the frequency of reuse of the cache lines belonging to $\tau_0$ and $\tau_1$, respectively, since we are unable to precisely know which line is the most recently accessed. Since the probability that a miss evicts a line belonging to a thread is inversely proportional to its reuse frequency, we assume the following relationship:

$$p_o/p_l = r_l/r_o \tag{14}$$

Furthermore, since a miss must fall on some line in the cache with probability 1:

$$p_l E + p_o(C - E) = 1 \tag{15}$$

Solving Equations 14 and 15, we obtain:

$$p_o = r_l/[r_o E + r_l(C - E)] \tag{16}$$

$$p_l = r_o/[r_o E + r_l(C - E)] \tag{17}$$

The values of $p_o$ and $p_l$ obtained from Equations 16 and 17 can be substituted in Equation 11 to obtain the hit-adjusted occupancy estimation model which handles LRU cache replacement effects.

## 2.3 Experiments

We evaluated the cache estimation models on Intel's CMPSched$im simulator [19], which supports binary execution and co-scheduling of multiple workloads. This enabled us to measure the accuracy of our cache occupancy models by comparing the estimated occupancy values with the actual values returned by the simulator. The ability to control scheduling allowed us to perform experiments in both under-committed and over-committed scenarios.

By default, the Intel simulator implements a CMP architecture using a pseudo-LRU policy used in modern processors, although it is also configurable to simulate random and other replacement policies. We configured the simulator to use a 3 GHz clock frequency, with private per-core 32 KB 4-way set-associative L1 caches, and a shared 4 MB 16-way set-associative L2 cache. All caches used a 64-byte line size. The number of hardware cores and software threads was varied across different experiments to test the effectiveness of our occupancy estimation models under diverse conditions.

During simulation, the per-core and per-CMP performance counters measuring L2 misses and hits were sampled once per millisecond, after which the occupancy estimates were updated for each software thread. Since cache occupancies exhibit rapid changes at this time scale, we averaged occupancies over 100 millisecond intervals. We plot one value per second for both the estimated and actual occupancy values, in order to display results more clearly over longer time scales. We refer to the miss-based occupancy estimation technique using the basic cache model presented in Section 2.1 as method *Estimate-M*. The extended cache model presented in Section 2.2 that also incorporates hit information to better model associativity is referred to as method *Estimate-MH*.

Our first experiment tests the effectiveness of the basic Estimate-M method in a dual-core configuration where a 16-way set-associative L2 cache is configured to use a simple random cache line replacement policy instead of pseudo-LRU. Figure 1 plots the estimated and actual cache occupancies over time when the two cores were running `mcf` and `omnetpp` from the SPEC CPU2006 benchmark suite. The estimated occupancy for each benchmark tracks its actual occupancy very closely, which is expected since the random replacement policy is consistent with our assumption of random cache access.

Our next experiment evaluates the same workload with the default pseudo-LRU line replacement policy which is used by actual processor hardware. Figures 2(a) and 2(b) plot the estimated and actual

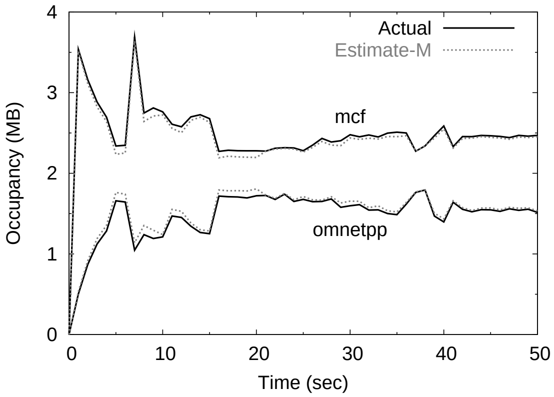


**Figure 1: Accuracy of basic Estimate-M method on dual-core system with random line replacement policy.**

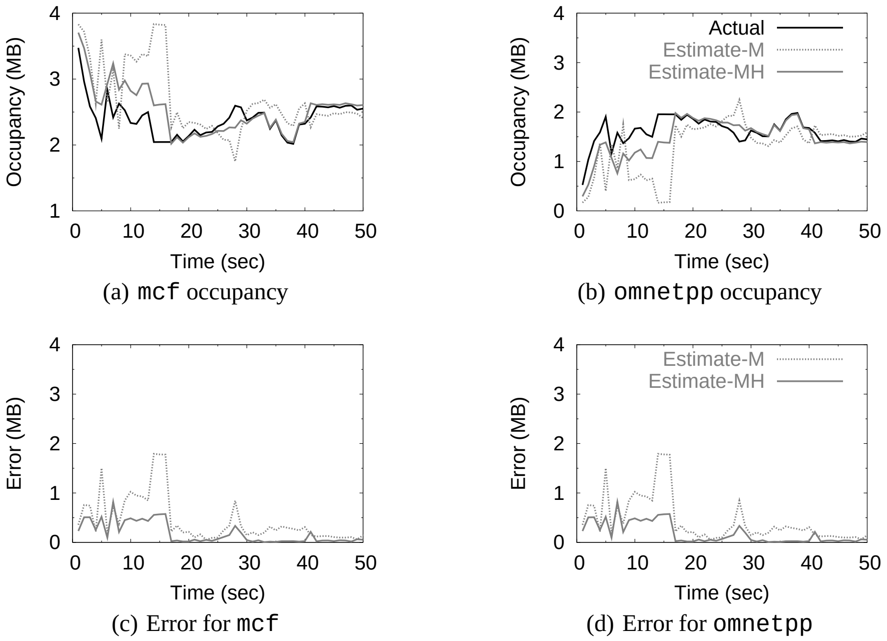
cache occupancies over time, for `mcf` and `omnetpp` respectively, using both the basic Estimate-M and extended Estimate-MH methods. Figures 2(c) and 2(d) present the absolute error between the actual and estimated values. The workloads in this experiment were selected to highlight the difference in accuracy between the two estimation methods, which generally agreed more closely for other workload pairings. In this case, the Estimate-M method is considerably less accurate, often showing a substantial discrepancy relative to the actual occupancies, especially during the interval between 8 and 18 seconds. On the other hand, the hit-adjusted Estimate-MH method, designed to better reflect LRU effects, is much more accurate, and tracks the actual occupancies fairly closely. The remaining experiments focus on the more accurate Estimate-MH method with various sets of co-running workloads. Figure 3 presents the results of two separate experiments with different co-running SPEC CPU2006 benchmarks with a dual-core configuration. Figures 3(a) and 3(b) show `gcc` running with `mcf` on the two cores; `omnetpp` and `perlbmk` are co-runners in Figures 3(c) and 3(d). The estimated occupancies match the actual values very closely.

Figure 4 presents the results of three separate experiments. Each row plots occupancy over time for four different co-running benchmarks from the SPEC CPU2000 and CPU2006 suites in a quad-core configuration. As with the dual-core results, the estimated occupancies in the quad-core system match the actual values with reasonably good accuracy.

We also evaluated the effectiveness of occupancy estimation in an over-committed system, in which many software threads are time-multiplexed onto a smaller number of hardware cores. In such a scenario, some threads will be descheduled at various points in time, waiting in a scheduler run queue to be dispatched onto a processor core. In our experiments, we used a 100 millisecond scheduling time quantum, with a simple round-robin scheduling policy selecting threads from a global run queue.

Figure 5 plots the actual and estimated occupancies over time for a quad-core system with ten software threads running various benchmarks from the SPEC CPU2000 and CPU2006 suites. In this experiment, the ten threads are scheduled to run on the four cores sharing the L2 cache. The accuracy of occupancy estimation remains high, despite the time-sliced scheduling.

In order to look at the estimation accuracy over shorter time intervals, Figure 6 zooms in to examine the first three seconds of execu-



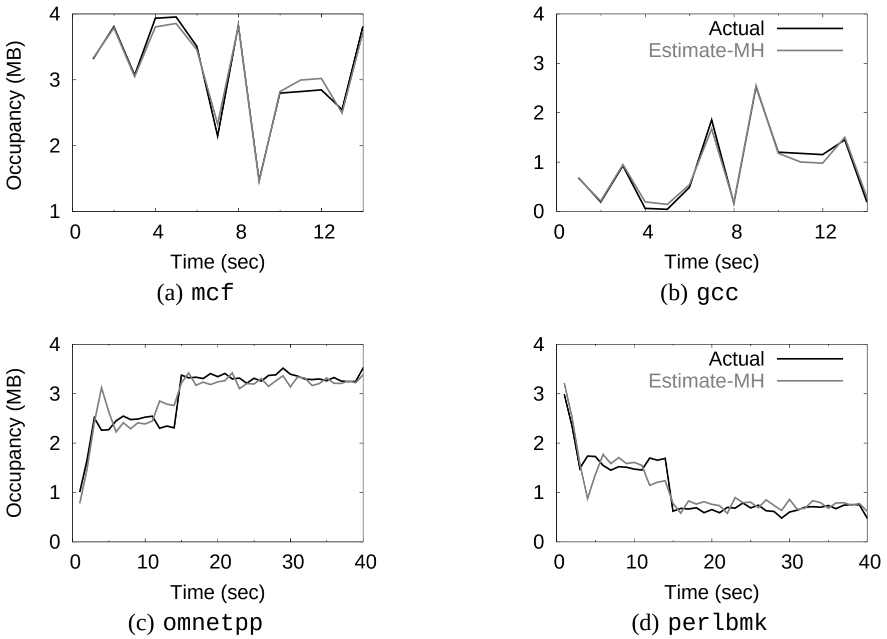
(a) `mcf` occupancy

(b) `omnetpp` occupancy

(c) Error for `mcf`

(d) Error for `omnetpp`

**Figure 2: Occupancy and estimation error for the Estimate-M and Estimate-MH methods.**



(a) `mcf`

(b) `gcc`

(c) `omnetpp`

(d) `perlbmk`

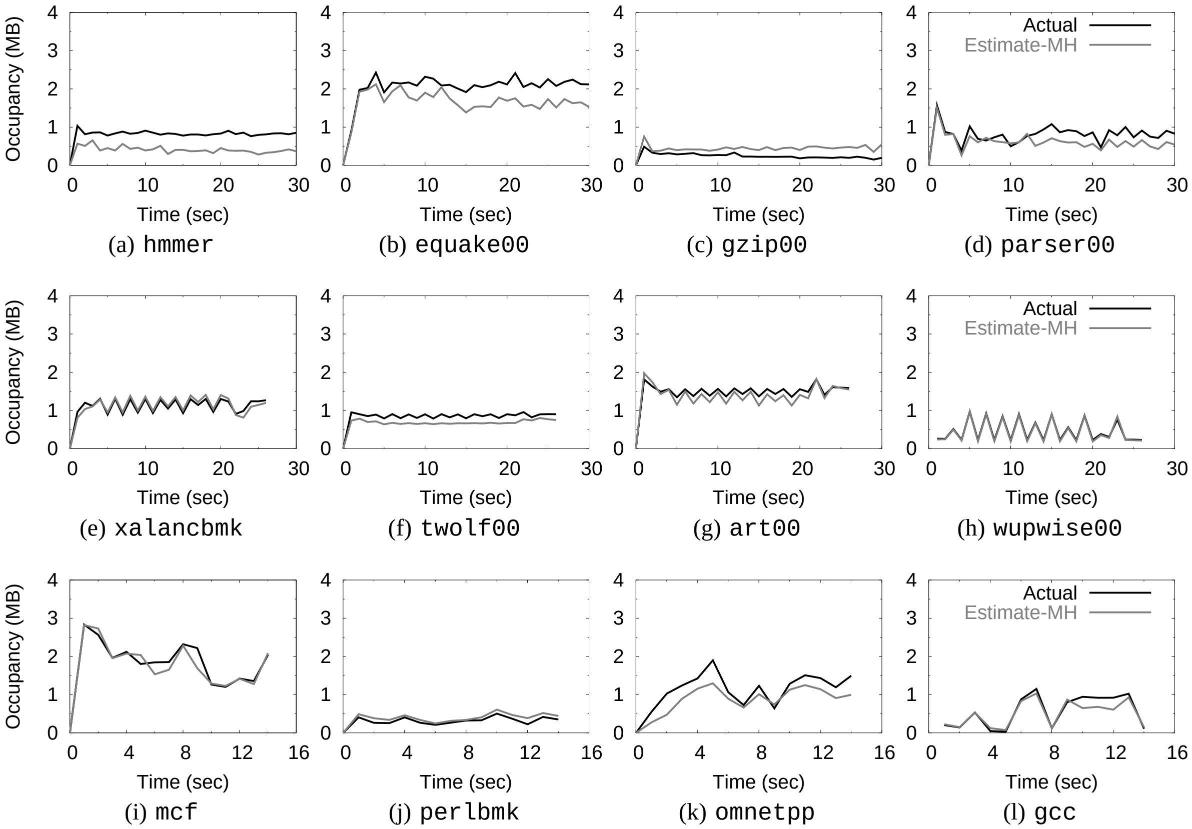
**Figure 3: Two pairs of co-runners in dual-core systems: `mcf` vs. `gcc`, and `omnetpp` vs. `perlbmk`.**

tion for the `mcf`, `equake00`, and `xalancbmk` workloads from Figures 5(a), 5(c), and 5(e). The actual and estimated occupancies are plotted every 100 milliseconds. Estimated occupancy tracks actual occupancy very closely, even during periods when a thread is de-scheduled and its occupancy falls to zero. Although these fine-grained results are reported for only three of the ten workloads from Figure 5, we observed similar behavior for the remaining benchmarks.

## 3. CACHE UTILITY CURVES

Our work on online cache occupancy prediction is intended to lay a foundation for improved resource management on multicore processors. In particular, we are currently investigating ways to improve fair and efficient scheduling of workloads (such as software threads, processes, tasks, or virtual CPUs) on hardware contexts (*e.g.*, cores or SMT threads) that compete for shared resources such as cache lines and memory bus bandwidth. In this paper, we do not propose any new cache-aware scheduling algorithms. Instead, we show how our method for online cache occupancy estimation can be used to produce workload-specific cache utility curves, which have proved valuable in prior research [3, 21, 26, 27, 28, 33].

These curves are presented with cache occupancy as the independent variable on the $x$-axis, and a dependent performance metric



(a) `hmmer` (b) `equake00` (c) `gzip00` (d) `parser00`

(e) `xalancbmk` (f) `twolf00` (g) `art00` (h) `wupwise00`

(i) `mcf` (j) `perlbmk` (k) `omnetpp` (l) `gcc`

**Figure 4: Three sets of co-runners in quad-core systems: occupancy over time for different sets of co-running SPEC CPU2000 and CPU2006 benchmarks in (a)–(d), (e)–(h), and (i)–(l).**

on the $y$-axis, such as the number of cache misses per reference, instruction, or cycle at different occupancies. In this section we explain our technique for lightweight online construction of cache utility curves, yielding information about the effect of cache size on expected performance for running workloads. We then present experimental MRC results for a series of benchmarks, using a prototype implementation, and compare them to MRCs collected for the same workloads using static page coloring.

All experiments were conducted on a Dell PowerEdge SC1430 host, configured with two 2.0 GHz Intel Xeon E5535 processors and 4GB RAM. Each quad-core Xeon processor actually consists of two separate dual-core CMPs in a single physical package. The two cores in each CMP share a common 4MB L2 cache. We implemented our techniques for cache utility curve generation in the VMware ESX Server 4.0 hypervisor [30]. Each benchmark application was deployed in a separate virtual machine, configured with a single CPU and 256MB RAM, running an unmodified Red Hat Enterprise Linux 5 guest OS (Linux 2.6.18-8.e15 kernel).

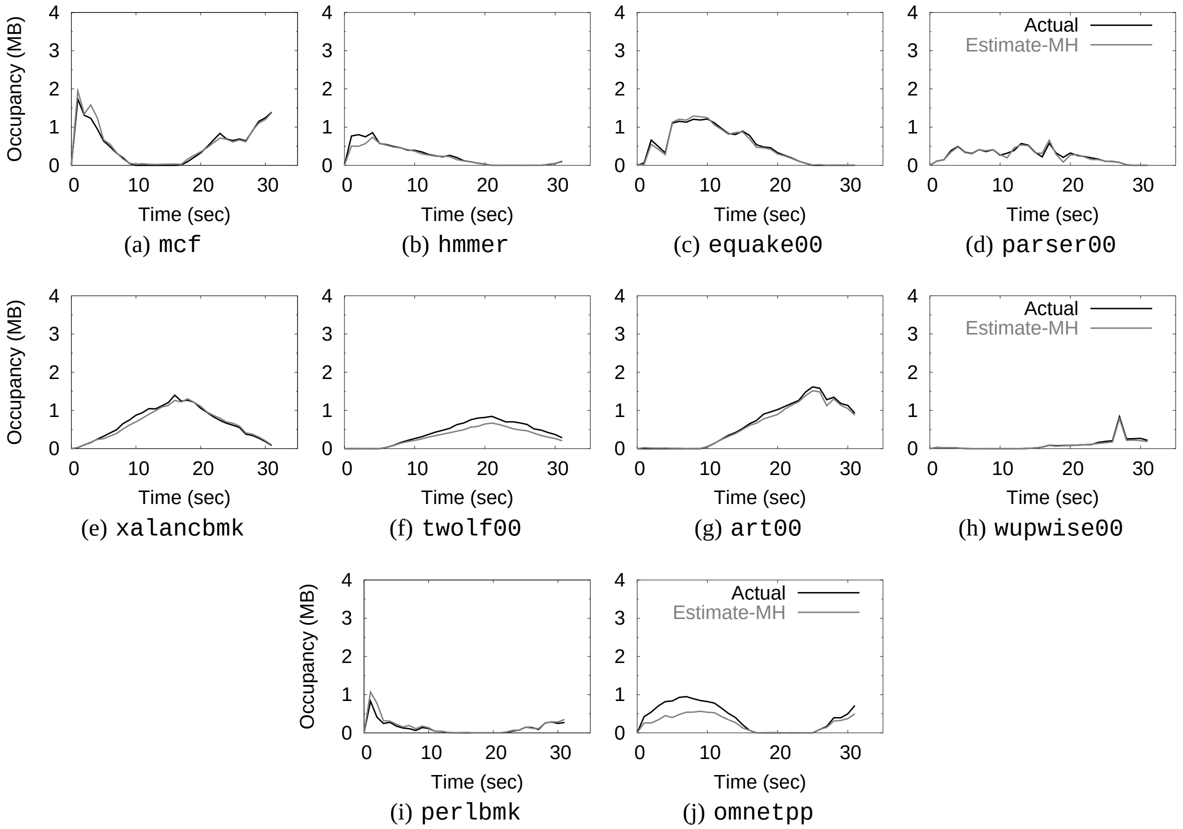
## 3.1 Curve Types

Most work in this area has focused on per-thread *miss-ratio curves* that plot cache misses per memory reference at different cache occupancies [3, 21, 26, 27, 28]. Another type of miss-ratio curve plots cache misses per instruction retired at different cache occupancies. We refer to miss-ratio curves in units of misses per kilo-reference as *MPKR* curves, and to those in units of misses per kilo-instruction as *MPKI* curves.

It is also possible to construct *miss-rate curves*, defined in terms of misses per kilo-cycle. Such *MPKC* curves are attractive for use with cache-aware scheduling policies, since they indicate the number of misses expected over a real-time interval for a workload with a given cache occupancy. However, a problem with MPKC curves is that they are sensitive to contention for memory bandwidth from co-running workloads. Under high contention, workloads start experiencing more memory stalls, throttling back their instruction issue rate, thereby decreasing their cache misses per unit time. Consequently, a cache utility function based on miss rates is dependent on dynamic memory bandwidth contention from co-running workloads. In contrast, MPKR and MPKI curves measure cache metrics that are intrinsic to a workload, independent of co-runners and timing details.

Figure 7 illustrates the problem of MPKC sensitivity to memory bandwidth contention using the SPEC2000 `mcf00` workload. Miss-rate curves for `mcf00` were collected using page coloring, but with different levels of memory read bandwidth contention generated by a micro-benchmark running on a different CMP sharing the same memory bus, but not the same cache. For a given cache occupancy value, the miss rates are higher when there is less memory bandwidth contention, resulting in variable miss-rate curves.

One can also generate *CPKI* curves, which measure the impact of cache size on the cycles per kilo-instruction efficiency of a workload. The CPKI metric has the advantage of directly showing the impact of cache size on a workload's performance, reflecting the effects of instruction-level parallelism that help tolerate cache miss latency. However, like MPKC curves, CPKI curves suffer from

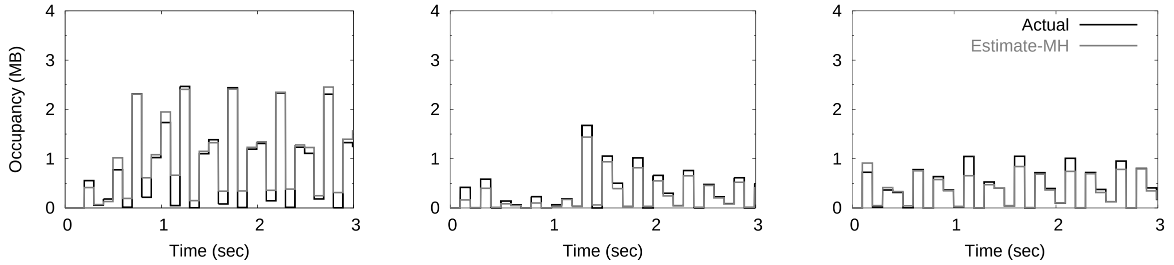


(a) `mcf` (b) `hmmer` (c) `equake00` (d) `parser00`

(e) `xalancbmk` (f) `twolf00` (g) `art00` (h) `wupwise00`

(i) `perlbmk` (j) `omnetpp`

**Figure 5: Occupancy estimation for over-committed quad-core system.**



(a) `mcf` (b) `equake00` (c) `xalancbmk`

**Figure 6: Fine-grained occupancy estimation in over-committed quad-core system.**

the problem of co-runner variability due to contention for memory bandwidth or other shared hardware resources.

Since MPKI and MPKR curves do not vary based on memory contention caused by co-runners, they are good candidates for determining a workload's intrinsic cache behavior. In some cases, however, it is also useful to infer the impact on workload performance due to the combined effects of cache and memory bandwidth contention. Our utility curve generator can produce both MPKI and CPKI curves to guide higher-level scheduling policies.

## 3.2 Curve Generation

We implemented our online cache-utility curve generator in ESX Server. Utilizing the occupancy estimation method described in Section 2, curve generation consists of two components at different time scales: fine-grained occupancy updates, and coarse-grained curve construction.

### *3.2.1 Occupancy Updates*

Each core updates the cache occupancy estimate for its currently-running thread every two milliseconds, using the linear occupancy model in Equation 9 to implement the Estimate-M method presented in Section 2.1. We were unable to use the more accurate hit-adjusted Estimate-MH method described in Section 2.2, due to the limited number of hardware performance counters available on our experimental platform.[2] A high-precision timer callback reads hardware performance counters to obtain the number of cache misses for both the local core and the whole CMP since the last update. In addition to this periodic update, occupancy estimates are also updated whenever a thread is rescheduled, based on the number of intervening cache misses since it last ran.

[2] The Intel E5535 processor supports only two programmable counters, which we used for counting cache misses by the local core and the whole CMP. More recent processors from Intel and AMD support at least four programmable counters, sufficient for supporting our more accurate model.

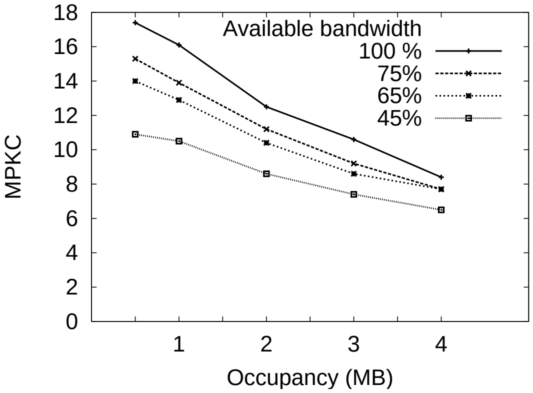


**Figure 7: Effect of memory bandwidth contention on MPKC miss-rate curve for `mcf00` workload.**

While always maintaining a precise cache occupancy estimate, our current implementation additionally quantizes cache occupancy in discrete levels equal to one-eighth of the total cache size, in order to support efficient curve generation. We construct discrete curves to bound the space and time complexity of their generation, while providing sufficient accuracy to be useful in cache-aware CPU scheduling enhancements. During each occupancy update for a thread, we record the change in values since the previous update for several hardware performance counters, including cache misses, instructions retired, and elapsed CPU cycles. These changes are added to aggregate values associated with the current discrete occupancy level. However, if an update spans multiple occupancy levels, the current incremental changes are not added to the aggregate values, because they cannot be attributed to a single level. Since occupancy updates are invoked very frequently, we tuned the timer callback carefully, and measured its cost as approximately 320 cycles on our experimental platform.

### *3.2.2 Generating Miss-Ratio Curves*

Miss-ratio curves are generated after a configurable time period, typically several seconds spanning thousands of fine-grained occupancy updates. For each discrete occupancy level, an MPKI value is computed by dividing the its associated aggregate cache misses (accumulated during the fine-grained occupancy updates) by the accumulated retired instructions for that level.

MPKI values are expected to be monotonically decreasing with increasing cache occupancy; *i.e.*, more cache leads to fewer misses per instruction. Our utility curve generator enforces this monotonicity property explicitly by adjusting MPKI values. Preference is given to those occupancy points that have the most updates, since we have more confidence in the performance metrics corresponding to these points. Starting with the most-updated occupancy point with MPKI value $m$, any lower MPKI values to its left or higher MPKI values to its right are set to $m$. Interestingly, monotonicity violations are good indicators of phase changes in workload behavior, although we do not yet exploit such hints. We instrumented our MRC generation code, including monotonicity enforcement, and found that it takes approximately 2850 cycles to execute on our experimental platform. The overheads for occupancy estimation and MRC construction are sufficiently low that these techniques can remain enabled at all times in production systems.

Our cache utility curve generator is extremely flexible. By recording appropriate statistics with each discrete occupancy point, a variety of different cache performance curves can be constructed. By default, we collect cache misses, instructions retired, and elapsed cycles, enabling generation of MPKI, MPKC, and CPKI curves.

### *3.2.3 Obtaining Full Curves*

A key challenge with our approach is obtaining performance metrics at all discrete occupancy points. In the steady state, a group of threads co-running on a shared cache achieve equilibrium occupancies. As a result, the cache performance curve for each thread has performance metrics concentrated around its equilibrium occupancy, leading to inaccuracies in the full cache performance curves.

In addition to passive monitoring, we have explored ways to actively perturb the execution of co-running threads to alter their relative cache occupancies temporarily. For example, varying the group of co-runners scheduled with a thread typically causes it to visit a wider range of occupancy points. An alternative approach is to dynamically throttle the execution of some cores, allowing threads on other cores to increase their occupancies. Our utility curve generator cannot use frequency and voltage scaling to throttle cores, since in commodity CMPs, all cores must operate at the same frequency [20]. However, we did have some success with duty-cycle modulation techniques [11, 31] to slow down specific cores dynamically.

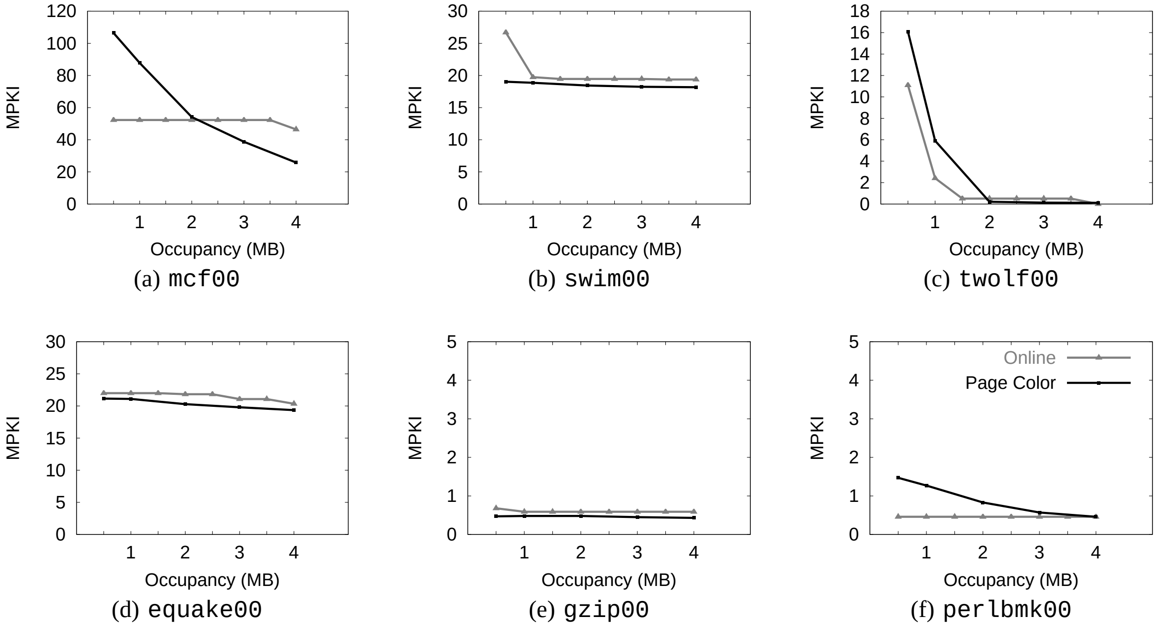
For thermal management, Intel processors allow system code to specify a multiplier (in discrete units of $12.5\%$) specifying the fraction of regular cycles during which a core should be halted. When a core is slowed down, its co-runners get an opportunity to increase their cache occupancy, while the occupancy of the thread running on the throttled core is decreased. To limit any potential performance impact, we enable duty-cycle modulation during less than $2\%$ of execution time. Experiments with SPEC CPU2000 benchmarks did not reveal any observable performance impact due to cache performance curve generation with duty-cycle modulation.

## 3.3 Experiments

We evaluated cache curve construction techniques using our ESX Server implementation. We first collected miss-ratio curves for various SPEC CPU2000 benchmarks (`mcf00`, `swim00`, `twolf00`, `equake00`, `gzip00` and `perlbmk00`), by running them to completion with access to an increasing number of page colors in each successive run. We then ran all six benchmarks together on a single CMP of the Dell system, using our online approach to generate the miss-ratio curves at benchmark completion time.

Figure 8 compares the miss-ratio curves of the benchmarks obtained online with those obtained by page coloring. In most cases, the MRC shapes and absolute MPKI values match reasonably well. However, in Figure 8(a), the MRC generated online for `mcf00` is flat at lower occupancy points, differing significantly from the page-coloring results. Even with duty-cycle modulation there is insufficient interference from co-runners to push `mcf00` into lower occupancy points. Since there are no updates for these points, the miss-ratio values for higher occupancy points are used as the best estimate due to monotonicity enforcement.

To analyze this further, Figure 9 shows separate MRCs generated online for `mcf00` with different co-runners, `swim00` and `gzip00`. The MRC generated when `mcf00` is running with `gzip00` is flat because `mcf00` only has updates at the highest occupancy point. The miss ratio of `mcf00` at the highest occupancy point is a factor



(a) mcf00 (b) swim00 (c) twolf00

(d) equake00 (e) gzip00 (f) perlbmk00

**Figure 8: Miss-ratio curves (MRCs) for SPEC CPU2000 workloads, obtained both online and offline.**

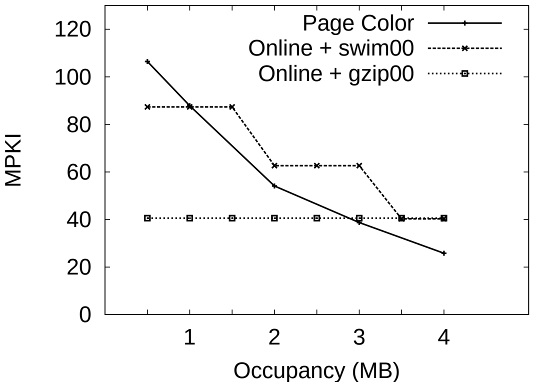


**Figure 9: MRC for mcf00 with different co-runners.**

of sixty more than the miss ratio of gzip00, which renders duty-cycle modulation ineffective, since it can throttle a core by at most a factor of eight. In contrast, the MRC generated with co-runner swim00 matches the MRC obtained by page-coloring closely.

### 3.4 Discussion

Our online technique for MRC construction builds upon our cache occupancy estimation model. While the MRCs generated for a working system in Section 3.3 are encouraging, there remain several open issues. By using only commodity hardware features, our MRCs may not always yield data points across the full spectrum of cache occupancies. Duty cycle modulation addresses this problem to some degree, but some sensitivity to co-runner selection may still remain. Although an MPKI curve is intrinsic to a workload, and does not vary based on contention from co-runners, the workload may be prevented from visiting certain occupancy levels due to co-runner interference, as observed in Figure 9. In practice, it may be necessary to vary co-runners selectively during some execution intervals, in order to allow a workload to reach high cache occupancies, or alternatively, to force a workload into low occupancy states, depending on the memory demands of the co-runners.

While the experiments in Section 3.3 compare offline MRCs with our online approach, they are produced at the time of benchmark completion. This introduces some potential differences between the online and offline curves, since online we plot MPKI values based on the time *during workload execution* at which a given occupancy is reached. We are currently investigating MRCs at different time granularities. Early investigations yield curves that remain stable for an execution phase, but which fluctuate while changing phases. We intend to study how MRCs can be used to identify phase changes as part of future work.

## 4. RELATED WORK

The focus of this paper is on efficient online cache modeling, including software techniques for estimating both runtime cache occupancies and performance curves for individual workloads. The intent is for such estimates to inform performance-related resource management decisions, including cache-aware scheduling. To the best of our knowledge, no prior software techniques exist for online estimation of per-thread cache occupancies in commodity processors with shared caches. Other researchers have, however, inferred cache usage and utility of different cache sizes. In CacheScouts [32], for example, hardware support for monitoring IDs and set sampling are used to associate cache lines with different workloads, enabling cache occupancy measurements. The use of special IDs differs from our occupancy estimation approach, which requires only currently-available performance monitoring events common to modern CMPs. For estimation of performance curves, the approach taken by RapidMRC [28] comes closest to our goal of an efficient online software technique, although as detailed below, it still requires uncommon hardware support and incurs significant runtime overhead.

In the area of shared-cache resource management, there is a significant literature on cache partitioning, using either hardware or software techniques [2, 5, 7, 13, 14, 17, 22, 23, 25, 27]. This has been prompted by the observation that multiple workloads sharing a cache may experience interference in the form of conflict misses and memory bus bandwidth contention, resulting in significant performance degradation. For example, several studies have shown significant variation in execution times for SPEC benchmarks, depending on co-runners competing for shared resources [14, 33].

Cache partitioning has the potential to eliminate conflict misses and improve fairness or overall performance. While hardware-based approaches are typically faster and more efficient than those implemented by software, they are not commonly available on current processors [26, 27]. Software techniques such as those based on page coloring require careful coordination with the memory management subsystem of the underlying OS or hypervisor, and are generally too expensive for workloads with dynamically varying memory demands [6, 15, 16].

A significant challenge with cache partitioning is deriving the optimal allocation size for a workload. One approach is to construct cache utility functions, or performance curves, that associate workload benefits (*e.g.*, in terms of miss ratio, miss rate, or CPI) with different cache sizes. In particular, methods for constructing miss-ratio curves (MRCs) have been proposed to capture workload performance impacts at different cache occupancies, but either require special hardware [21, 26, 27], or incur high overhead [3, 28].

The Mattson Stack Algorithm [18] generates MRCs by maintaining an LRU-ordered stack of memory addresses. RapidMRC [28] uses this algorithm as the basis for its online MRC construction, but requires hardware support, in the form of a *Sampled Data Address Register* (SDAR) in the IBM POWER5 performance monitoring unit, to obtain a trace of *all* data accesses to the L2 cache. The total cost of online MRC construction is several hundred milliseconds, with more than 80 milliseconds of workload stall time due to the high overhead of trace collection. This overhead is mitigated by triggering MRC construction only when phase transitions are detected, based on changes in the overall cache miss rate. However, since changes in cache miss rates can be triggered by cache contention caused by co-runners, and not necessarily phase changes, the phase transition detection in RapidMRC does not seem robust in over-committed environments.

In contrast, we deploy an efficient online method to construct MRCs and other cache-performance curves, requiring only commonly-available performance counters. Due to the low overhead of our cache-performance curve construction, it can remain enabled at all times, providing up-to-date information pertaining to the most recent phase. As a result, our technique does not require an offline reference point to account for vertical shifts in the online curves due to phase transitions as in [28], and is also robust in the presence of cache contention from co-runners. We do, however, suffer from the problem of obtaining enough occupancy data points to construct full curves. Using duty-cycle modulation to temporarily reduce the rate of memory access by competing workloads is one technique that has the potential to alleviate this problem.

## 5. CONCLUSIONS AND FUTURE WORK

We have introduced several novel techniques for practical online cache modeling of commodity multicore processors sharing a last-level cache. Using both simulations and empirical results from a prototype implementation, we demonstrated the effectiveness of our software-based approach with quantitative experiments for a variety of workloads and CMP configurations.

Our first contribution is efficient online estimation of cache occupancies for software threads, using only performance counters commonly available on commodity processors. To the best of our knowledge, our technique is the first to enable accurate online cache occupancy monitoring without requiring additional hardware support. We derive a basic statistical model for cache occupancy estimation based on cache miss counts, and extend it to incorporate cache hit counts for improved accuracy with set-associative caches that employ pseudo-LRU replacement policies. Simulation results using Intel's CMPSched$im verify that our estimates track actual occupancies closely, across various sets of co-running SPEC benchmarks on realistic dual-core and quad-core CMP configurations, even in the presence of workload phase changes and descheduling events in over-committed scenarios.

Building on occupancy estimation, we demonstrate how to dynamically generate cache performance curves, such as MRCs, which capture the utility of cache space on workload performance. Empirical results using the VMware ESX Server hypervisor demonstrate that we are able to construct per-thread MRCs online with low overhead, in the presence of interference from co-runners. Comparisons with MRCs generated offline using static page coloring indicate that our lightweight approach is sufficiently accurate to inform online scheduling algorithms. We also highlight remaining challenges, and show how duty cycle modulation can be used to facilitate obtaining a wider range of MRC occupancy points, by dynamically varying the level of cache contention between co-runners.

While we have presented several new online techniques for CMP cache modeling, many interesting research opportunities remain. We plan to enhance our occupancy estimation approach to incorporate the effects of data sharing and constructive interference between threads. We are also examining various approaches to obtain accurate cache performance curves at all cache occupancy points. Additionally, we are exploring ways to extend and integrate our software techniques with future hardware support for cache QoS monitoring and enforcement, and to scale to wide CMPs containing large numbers of cores.

The techniques described in this paper are currently being applied to cache-aware fair and efficient scheduling in the VMware ESX Server hypervisor. Our scheduling-related research is directed at algorithms for optimizing co-runner placement based on estimated performance trade-offs, as well as mechanisms for improving fairness by adjusting thread scheduling priorities to account for co-runner interference.

## 6. REFERENCES

[1] Advanced Micro Devices, Inc. *Multi-Core Processors from AMD*, 2009. http://multicore.amd.com/.

[2] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *ACM/IEEE International Symposium on Microarchitecture (MICRO '99)*, pages 248–259, November 1999.

[3] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '06)*, pages 89–99, 2006.

[4] J. M. Calandrino and J. H. Anderson. Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In *EuroMicro Conference on Real-Time Systems (ECRTS '08)*, pages 299–308, July 2008.

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *International Conference on Supercomputing (ICS '07)*, pages 242–252, June 2007.

[6] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, 2006.

[7] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *High Performance Computing*, volume 4297/2006, pages 22–34, 2006.

[8] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 335–346, March 2010.

[9] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multicore processors. Technical Report TR-17-06, Harvard University, 2006.

[10] J. Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Inc., 1994.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, June 2009.

[12] Intel Corporation. *Intel Multi-Core Technology*, 2009. http://www.intel.com/multi-core/.

[13] R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *the 18th Annual International Conference on Supercomputing*, pages 257–266, 2004.

[14] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Parallel Architectures and Compilation Techniques (PACT '04)*, October 2004.

[15] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *the 3rd IEEE Real-time Technology and Applications Symposium*, 1997.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *the 14th IEEE International Symposium on High Performance Computer Architecture*, pages 367–378, 2008.

[17] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *International Symposium on High-Performance Computer Architecture*, pages 176–185, 2004.

[18] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[19] J. Moses, K. Aisopos, A. Jaleel, R. Iyer, R. Illikkal, D. Newell, and S. Makineni. CMPSched$im: Evaluating OS/CMP interaction on shared cache management. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*, pages 113–122, April 2009.

[20] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2):109–122, 2006.

[21] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, 2006.

[22] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *Parallel Architectures and Compilation Techniques (PACT '06)*, pages 2–12, September 2006.

[23] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *the 27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[24] T. Sherwood, B. Calder, and J. S. Emer. Reducing cache misses using hardware and software page placement. In *International Conference on Supercomputing (ICS '99)*, June 1999.

[25] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: Managing shared caches in CMPs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, March 2008.

[26] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *International Conference on Supercomputing (ICS '01)*, pages 1–12, June 2001.

[27] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, April 2004.

[28] D. Tam, R. Azimi, L. Soares, and M. Stumm. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, March 2009.

[29] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of EuroSys 2007*, pages 47–58, March 2007.

[30] VMware, Inc. *vSphere Resource Management Guide: ESX 4.0, ESXi 4.0, vCenter Server 4.0*, 2009.

[31] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *Proceedings of the USENIX Annual Technical Conference*, June 2009.

[32] L. Zhao, R. Iyer, R. Illikkal, J. Moses, D. Newell, and S. Makineni. CacheScouts: Fine-grain monitoring of shared caches in CMP platforms. In *Parallel Architectures and Compilation Techniques (PACT '07)*, September 2007.

[33] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 129–141, March 2010.