8

CAFÉ: Cache-Aware Fair and Efficient Scheduling for CMPs

Richard West

Department of Computer Science, Boston University, Boston, MA, USA

Puneet Zaroo

VMware Inc., Palo Alto, CA, USA

Carl A. Waldspurger

Formerly at VMware Inc., Palo Alto, CA, USA

Xiao Zhang

Google, Inc., Mountain View, CA, USA[1]

CONTENTS

Modern chip-level multiprocessors (CMPs) typically contain multiple processor cores sharing a common last-level cache, memory interconnects, and other

[1] Xiao Zhang was formerly at VMware Inc. for this work.

hardware resources. Workloads running on separate cores compete for these resources, often resulting in highly variable performance. Unfortunately, commodity processors manage shared hardware resources in a manner that is opaque to higher-level schedulers responsible for multiplexing these resources across workloads with varying demands and importance. As a result, it is extremely challenging to optimize for efficient resource utilization or enforce quality-of-service policies.

Effective cache management requires accurate measurement of per-thread cache occupancies and their impact on performance, often summarized by utility functions such as miss-ratio curves (MRCs). We introduce an efficient online technique for generating MRCs and other cache utility curves, requiring only performance counters available on commodity processors. Building on these monitoring and inference techniques, we also introduce novel methods to improve the fairness and efficiency of CMP scheduling decisions. *Vtime compensation* adjusts a thread's scheduling priority to account for cache and memory system interference from co-runners, and *cache divvying* estimates the performance impact of co-runner placements. We demonstrate the effectiveness of our monitoring and scheduling techniques with quantitative experiments, including both simulation results and a prototype implementation in the VMware ESX Server hypervisor.

8.1 Introduction

Advances in processor architecture have led to a proliferation of multi-core 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 (AMD 2009; Intel Corporation 2009). Workloads running on separate cores compete for these shared resources, often resulting in highly variable or unpredictable performance (Fedorova, Seltzer, and Smith 2006; Kim, Chandra, and Solihin 2004).

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 enforce quality-of-service policies.

Many hardware approaches have been proposed to address this problem, introducing low-level architectural mechanisms to support cache occupancy monitoring and/or the ability to partition cache space among multiple work-

loads (Albonesi 1999; Chang and Sohi 2007; Dybdahl, Stenström, and Natvig 2006; Iyer 2004; Kim, Chandra, and Solihin 2004; Liu, Sivasubramaniam, and Kandemir 2004; Rafique, Lim, and Thottethodi 2006; Ranganathan, Adve, and Jouppi 2000; Srikantaiah, Kandemir, and Irwin 2008; Suh, Rudolph, and Devadas 2004). 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 (Berg, Zeffer, and Hagersten 2006; Qureshi and Patt 2006; Suh, Devadas, and Rudolph 2001; Suh, Rudolph, and Devadas 2004; Tam et al. 2009). However, existing techniques for generating MRCs either require custom hardware support, or incur non-trivial software overheads.

Constructing cache utility curves is an important step toward effective cache management. To utilize caches more efficiently and provide differential quality of service for workloads, higher-level resource management policies are needed to leverage them. For example, schedulers can exploit cache performance information to make better co-runner placement decisions (Calandrino and Anderson 2008; Suh, Devadas, and Rudolph 2001; Tam, Azimi, and Stumm 2007), 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 (Cho and Jin 2006; Liedtke, Härtig, and Hohmuth 1997; Lin et al. 2008), such hard partitioning is inflexible and generally prevents efficient cache utilization. Moreover, without special hardware support (Sherwood, Calder, and Emer 1999), 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 offer an alternative for cache-aware fair and efficient scheduling in a system called *CAFÉ*. Unlike most previous approaches, CAFÉ requires no special hardware support, using only basic performance counters found on virtually all modern processors, including commodity x86 CMPs (AMD 2007; Intel Corporation 2009). Several new cache modeling and inference methods are introduced for accurate cache performance monitoring. Building on this basic monitoring capability, we also introduce new techniques for improving the fairness and efficiency of CMP scheduling decisions.

CAFÉ efficiently computes accurate per-workload cache occupancy estimates from per-core cache miss counts. Occupancy estimates are leveraged to support inexpensive construction of general cache utility curves. For example, miss-ratio and miss-rate curves can be generated by incorporating additional performance counter values for instructions retired and elapsed cycles, avoiding the need for special hardware or memory address traces.

We leverage CAFÉ's cache monitoring infrastructure to perform proper charging for resource consumption, accounting for dynamic interference between co-running workloads within a CMP. A new *vtime compensation* technique is introduced to compensate a workload for interference from co-runners. We also present CAFÉ's *cache divvying* policy for predicting approximate

cache allocations during co-runner execution. Using estimated cache utility curves, we are able to determine good co-runner placements to maximize aggregate throughput.

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 8.3 builds on this foundation, explaining our method for online construction of cache utility curves. Using a prototype implementation in the VMware ESX Server hypervisor, we examine its accuracy by comparing CAFÉ's dynamically generated MRCs with MRCs for the same workloads collected via static page coloring. Section 8.4 introduces our cache-aware scheduling policies: vtime compensation for cache-fair scheduling, and our cache-divvying strategy for estimating the performance impact of co-runner placements. Quantitative experiments in the context of ESX Server show that these schemes are able to improve fairness and efficiency. Related work is examined in Section 8.5. Finally, we summarize our conclusions and highlight opportunities for future work in Section 8.6.

8.2 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[2] 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 Standard Performance Evaluation Corporation (SPEC) benchmarks execute concurrently on multiple cores. Since real processors do not expose the contents of hardware caches to software,[3] we measure accuracy using the Intel CMPSched$im simulator (Moses et al. 2009) 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, τ, at time t, given contention for cache lines by multiple threads running on all the cores that share that cache. At time t, thread τ may be descheduled, or it may be actively executing on one core while other threads are active on the remaining cores.

[2]Least Recently Used.

[3]Current processor families do not allow software to inspect cache tags, although the MIPS R4000 (Heinrich 1994) did provide a `cache` instruction with this capability.

8.2.1 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 thread τ 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. Observe that for n-way set-associative caches, a cache set is selected by using a subset of bits in a memory address, and then a victim cache block within the set is typically chosen using an LRU-like algorithm. Our own observations suggest that n-way set-associative caches in modern multicore processors have some element of randomness to their line replacement policies within sets. In many cases, these policies use some form of binary decision tree as well as a degree of random selection to reduce the bitwise logic when approximating algorithms such as LRU. It is reasonable to assume that randomness will have a greater effect as the number of ways in cache sets is increased in future processors.

In this work, we also assume each cache line is allocated to a single thread at any point in time. Furthermore, we do not consider the effects of data sharing across threads, although this is an important topic for future work.

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 τ_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 τ_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, τ_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 τ_l on all cores of a CMP that cause cache lines to be allocated in

response to such misses. We use the notation τ_o to represent the aggregate behavior of all other threads, treating it as if it were a single thread.

Theorem 8.1 *Consider a cache of size C lines, with E cache lines belonging to τ_l and $C-E$ cache lines belonging to τ_o at some time, t. If, in some interval, δt, there are m_l misses corresponding to τ_l and m_o misses corresponding to τ_o, then the expected occupancy of τ_l at time $t+\delta t$ is approximately $E' = E + (1 - \frac{E}{C}) \cdot m_l - \frac{E}{C} \cdot m_o$.*

Proof 8.1 *First, at time t, it is assumed that τ_l and τ_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*

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

This follows from the prior probabilities, at time t:

$$Pr\{i \text{ belonged to } \tau_l\} = \frac{E}{C} \tag{8.1}$$

$$Pr\{i \text{ belonged to } \tau_o\} = 1 - \frac{E}{C} \tag{8.2}$$

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

$$Pr\{\tau_l \text{ replaces } \tau_o \text{ on line } i\} = 1 - \left[1 - \frac{m_l}{C(m_l + m_o)}\right]^{(m_l+m_o)} \tag{8.3}$$

In (8.3), $m_l/[C(m_l+m_o)]$ represents the probability that a miss by τ_l will result in an arbitrary line, i, being populated by contents for τ_l. We know that the probability of a particular line being replaced by a single miss is $1/C$, and the ratio $m_l/(m_l+m_o)$ corresponds to the probability of that miss being caused by one of τ_l's accesses. Note that here we make no assumptions about the order of interleaved memory accesses made by two or more co-running threads. Instead, the ratio $m_l/(m_l + m_o)$ is based on the probability that, among all possible interleaved misses from τ_l and τ_o, τ_l will have the last miss associated with a given cache line.

It follows from (8.3) that the probability of τ_o replacing τ_l on line i at the end of $m_l + m_o$ misses is

$$Pr\{\tau_o \text{ replaces } \tau_l \text{ on line } i\} = 1 - \left[1 - \frac{m_o}{C(m_l + m_o)}\right]^{(m_l+m_o)} \tag{8.4}$$

Therefore,

$$Pr\{i \text{ belongs to } \tau_l \mid i \text{ belonged to } \tau_l\} = 1 - Pr\{\tau_o \text{ replaces } \tau_l \text{ on line } i\} = \left[1 - \frac{m_o}{C(m_l + m_o)}\right]^{(m_l+m_o)} \quad (8.5)$$

$$Pr\{i \text{ belongs to } \tau_l \mid i \text{ belonged to } \tau_o\} = Pr\{\tau_l \text{ replaces } \tau_o \text{ on line } i\} = 1 - \left[1 - \frac{m_l}{C(m_l + m_o)}\right]^{(m_l+m_o)} \quad (8.6)$$

From (8.1), (8.2), (8.5), *and* (8.6), *we have*

$$Pr\{i \text{ belongs to } \tau_l\} = \frac{E}{C} \cdot \left[1 - \frac{m_o}{C(m_l + m_o)}\right]^{(m_l+m_o)} + \left(1 - \frac{E}{C}\right) \cdot \left[1 - \left[1 - \frac{m_l}{C(m_l + m_o)}\right]^{(m_l+m_o)}\right] \quad (8.7)$$

Ignoring the effects of quadratic and higher-degree terms, the first-degree linear approximation of (8.7) *becomes*

$$Pr\{i \text{ belongs to } \tau_l\} = \frac{E}{C} \cdot (1 - m_o/C) + \left(1 - \frac{E}{C}\right) m_l/C \quad (8.8)$$

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

$$E' = E(1 - m_o/C) + \left(1 - \frac{E}{C}\right) m_l = E + \left(1 - \frac{E}{C}\right) \cdot m_l - \frac{E}{C} \cdot m_o \quad (8.9)$$

This follows from (8.8) *by considering the state of each of the* C *cache lines as independent of all others.*

Observe that the recurrence relation in (8.9) captures the changes in cache occupancy for some thread over a given interval of time, with known local and global misses. The terms $[1 - m_o/(C(m_l + m_o))]^{(m_l+m_o)}$ and $[1 - m_l/(C(m_l + m_o))]^{(m_l+m_o)}$ in (8.7) approximate to $e^{-m_o/C}$ and $e^{-m_l/C}$, respectively. Thus, for situations where $m_l + m_o >> 1$, (8.9) becomes

$$E' = Ee^{-m_o/C} + C(1 - E/C)(1 - e^{-m_l/C}) \quad (8.10)$$

Equation (8.10) is significant in that it shows that the cache occupancy of a thread (here, τ_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 τ_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 (8.9) 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 (Intel Corporation 2009) used for our experiments in Section 8.3, 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.

8.2.2 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 (8.9) to additionally incorporate cache *hit* information, modeling the reduced replacement probability due to LRU effects when lines are reused. Equation (8.9) can be rewritten as

$$E' = E(1 - m_o p_l) + (C - E)m_l p_o \tag{8.11}$$

where p_l is the probability that a miss falls on a line belonging to τ_l, and p_o is the probability that a miss falls on a line belonging to τ_o. Since (8.9) 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

$$r_l = (h_l + m_l)/E \tag{8.12}$$

$$r_o = (h_o + m_o)/(C - E) \tag{8.13}$$

to quantify the frequency of reuse of the cache lines of τ_l and τ_o, respectively. h_l and h_o represent the number of cache hits experienced by τ_l and τ_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 τ_0 and τ_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:

$$p_o/p_l = r_l/r_o \tag{8.14}$$

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

$$p_l E + p_o(C - E) = 1 \tag{8.15}$$

Solving (8.14) and (8.15), we obtain:

$$p_o = r_l/[r_oE + r_l(C - E)] \quad (8.16)$$

$$p_l = r_o/[r_oE + r_l(C - E)] \quad (8.17)$$

The values of p_o and p_l obtained from (8.16) and (8.17) can be substituted in (8.11) to obtain the hit-adjusted occupancy estimation model which handles LRU cache replacement effects.

8.2.3 Experiments

We evaluated the cache estimation models on Intel's CMPSched$im simulator (Moses et al. 2009), 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 8.2.1 as method *Estimate-M*. The extended cache model presented in Section 8.2.2 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 8.1 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.

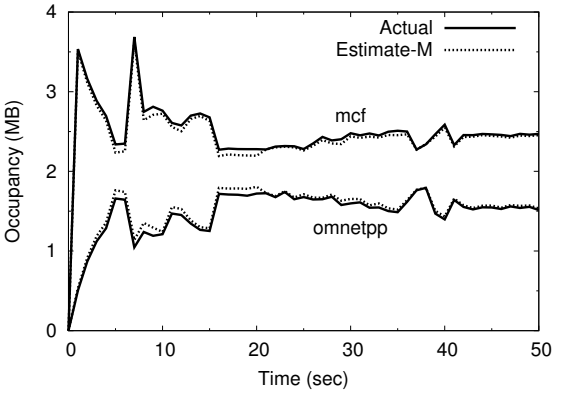


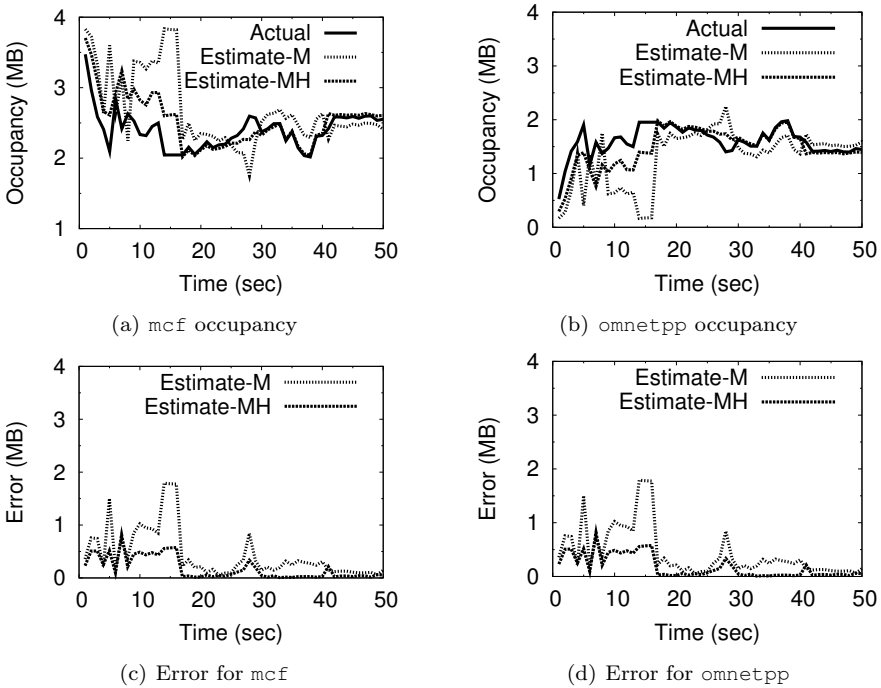
FIGURE 8.1
Accuracy of basic Estimate-M method on a dual-core system with random line replacement policy

Our next experiment evaluates the same workload with the default pseudo-LRU line replacement policy which is used by actual processor hardware. Figure 8.2(a) and (b) plot the estimated and actual cache occupancies over time, for `mcf` and `omnetpp`, respectively, using both the basic Estimate-M and extended Estimate-MH methods. Figure 8.2(c) and (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 8.3 presents the results of two separate experiments with different co-running SPEC CPU2006 benchmarks with a dual-core configuration. Figure 8.3(a) and (b) show `mcf` running with `gcc` on the two cores; `omnetpp` and `perlbmk` are co-runners in Figure 8.3(c) and (d). The estimated occupancies match the actual values very closely.

Figure 8.4 shows the cache occupancy over time for four different co-running benchmarks from the SPEC CPU2006 suite in a quad-core configuration. Although not shown, we also conducted similar experiments with other benchmarks from the SPEC CPU2000 and 2006 suites, achieving similar levels of accuracy between estimated and actual values. As with the dual-core results, experiments on a quad-core platform are of similar precision.

We also evaluated the effectiveness of occupancy estimation in an over-



(a) `mcf` occupancy

(b) `omnetpp` occupancy

(c) Error for `mcf`

(d) Error for `omnetpp`

FIGURE 8.2
Occupancy and estimation error for the Estimate-M and Estimate-MH methods

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.

Figures 8.5 and 8.6 show plots of the actual and estimated occupancies over time for an over-committed quad-core system. Together, the two figures show ten software threads running various benchmarks from the SPEC CPU2000 and CPU2006 suites.[4] In the corresponding 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 8.7 zooms in to examine the first three seconds of execution for the `mcf` and `equake00` workloads from Figure 8.5(a) and (c), respectively. The actual

[4]Benchmarks with names ending in `00` are from SPEC CPU2000, while all others are from CPU2006.

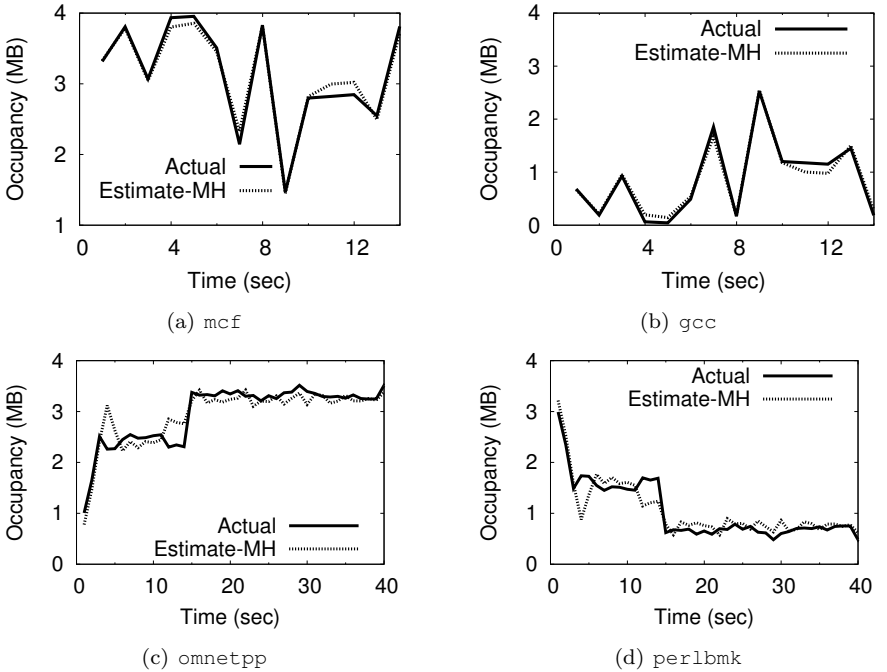
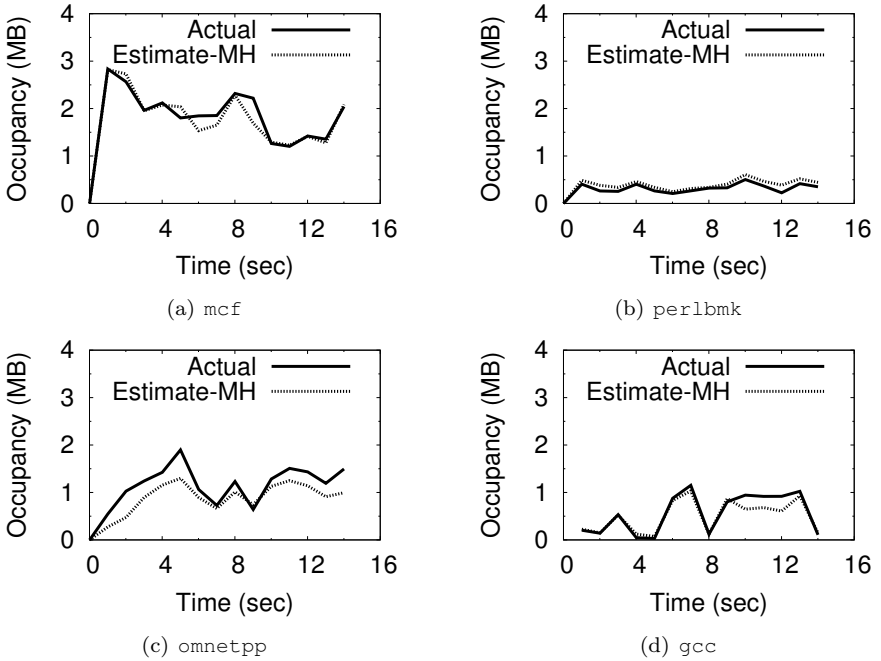


FIGURE 8.3
Two pairs of co-runners in dual-core systems: `mcf` versus `gcc`, and `omnetpp` versus `perlbmk`

and estimated occupancies are plotted every 100 ms. Estimated occupancy tracks actual occupancy very closely, even during periods when a thread is descheduled and its occupancy falls to zero. Although these fine-grained results are reported for only two of the ten workloads from Figures 8.5 and 8.6, we observed similar behavior for the remaining benchmarks.

8.3 Cache Utility Curves

Central to CAFÉ's resource management framework for fair and efficient scheduling is an understanding of workload-specific cache utility curves. These curves are presented with cache occupancy as the independent variable on the x-axis, and a dependent performance metric 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



(a) mcf (b) perlbmk

(c) omnetpp (d) gcc

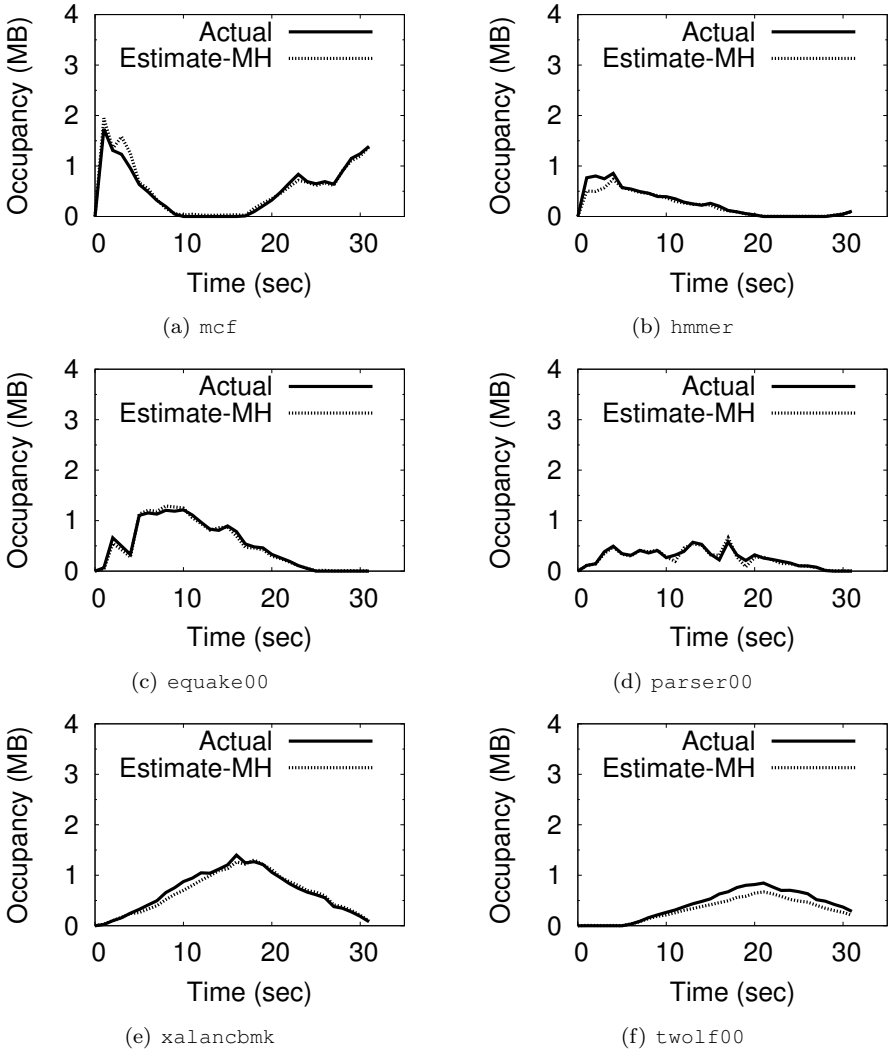
FIGURE 8.4
Cache occupancy over time for four co-runners in a quad-core system

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 CAFÉ 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 4 GB 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 4 MB L2 cache. We implemented our CAFÉ prototype in the VMware ESX Server 4.0 hypervisor (VMware, Inc. 2009). Each benchmark application was deployed in a separate virtual machine, configured with a single CPU and 256 MB RAM, running an unmodified Red Hat Enterprise Linux 5 guest OS (Linux 2.6.18-8.e15 kernel).

8.3.1 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 (Berg, Zeffer,



(a) mcf (b) hmmer

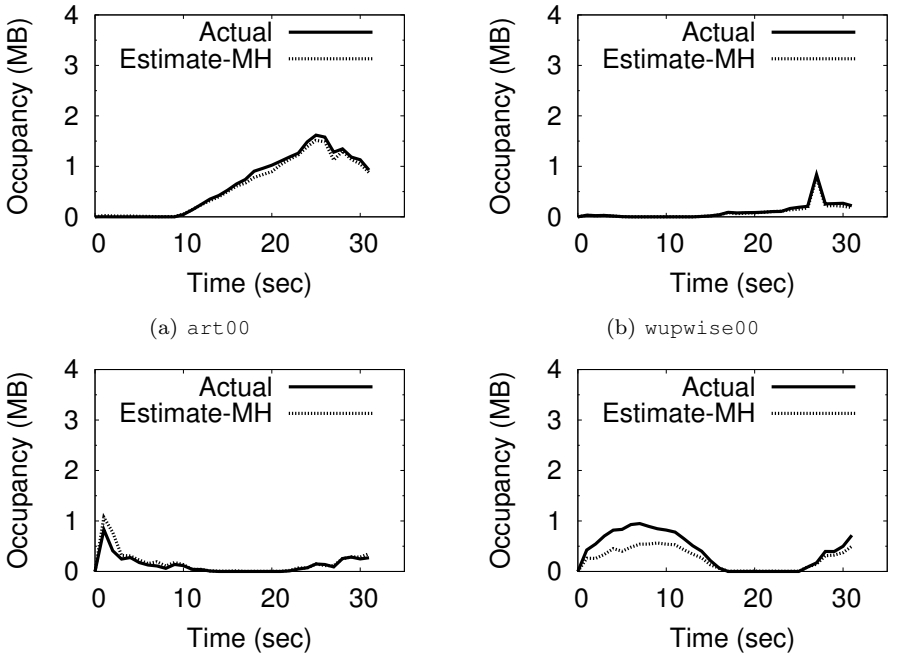
(c) equake00 (d) parser00

(e) xalancbmk (f) twolf00

FIGURE 8.5
Occupancy estimation for an over-committed quad-core system (Part 1)

and Hagersten 2006; Qureshi and Patt 2006; Suh, Devadas, and Rudolph 2001; Suh, Rudolph, and Devadas 2004; Tam et al. 2009). 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.

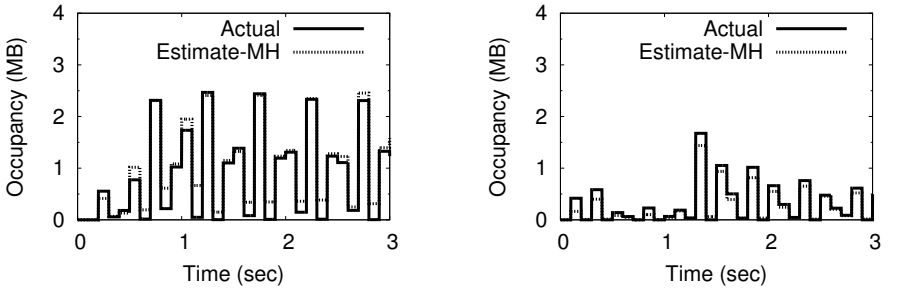
It is also possible to construct *miss-rate curves*, defined in terms of misses



(a) `art00` (b) `wupwise00`

(c) `perlbmk` (d) `omnetpp`

FIGURE 8.6
Occupancy estimation for an over-committed quad-core system (Part 2)



(a) `mcf` (b) `equake00`

FIGURE 8.7
Fine-grained occupancy estimation in an over-committed quad-core system

per kilo-cycle. Such *MPKC* curves are attractive for use with cache-aware scheduling policies, such as those presented in Section 8.4, 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 8.8 illustrates the problem of MPKC sensitivity to memory bandwidth contention using the SPEC2000 `mcf` workload. Miss-rate curves for `mcf` 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 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. Therefore CAFÉ generates both MPKI and CPKI curves and utilizes them to guide its higher-level scheduling policies.

8.3.2 Curve Generation

We implemented CAFÉ's online cache-utility curve generation in ESX Server. Utilizing the occupancy estimation method described in Section 8.2, curve generation consists of two components at different time scales: fine-grained occupancy updates and coarse-grained curve construction.

8.3.2.1 Occupancy Updates

Each core updates the cache occupancy estimate for its currently running thread every two milliseconds, using the linear occupancy model in (8.9). A high-precision timer callback reads hardware performance counters to obtain

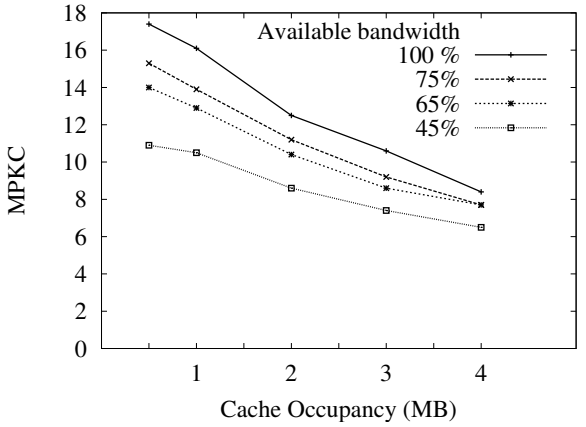


FIGURE 8.8
Effect of memory bandwidth contention on the MPKC miss-rate curve for the SPEC CPU2000 `mcf` workload

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.

Our current implementation tracks cache occupancy in discrete units equal to one-eighth of the total cache size. 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 cache occupancy update for a thread, several performance metrics are associated with its current occupancy level, including accumulated cache misses, instructions retired, and elapsed CPU cycles. 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.

8.3.2.2 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 point, an MPKI value is computed by dividing the accumulated cache misses by the accumulated retired instructions at that occupancy.

MPKI values are expected to be monotonically decreasing with increasing cache occupancy; i.e., more cache leads to fewer misses per instruction. CAFÉ enforces this monotonicity property explicitly by adjusting MPKI values. Preference is given to those occupancy points which 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 CAFÉ does 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 they can remain enabled at all times.

8.3.2.3 Generating Other Curves

The basic CAFÉ framework is extremely flexible. By recording appropriate statistics with each discrete occupancy point, a variety of different cache performance curves can be constructed. By default, CAFÉ collects cache misses, instructions retired, and elapsed cycles, enabling generation of MPKI, MPKC, and CPKI curves.

We could not experiment with generating MPKR curves, due to limitations of our experimental platform. The Intel Core architecture provides only two programmable counters, which were used to obtain core and whole-CMP cache misses, respectively. MPKI, MPKC, and CPKI curves can be generated by CAFÉ, since retired instructions and elapsed cycles are available as additional fixed hardware counters.

8.3.2.4 Obtaining Full Curves

A key challenge with CAFÉ's 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. CAFÉ cannot use frequency and voltage scaling to throttle cores, since in commodity CMPs, all cores must operate at the same frequency (Naveh et al. 2006). However, we did have some success with duty-cycle modulation techniques (Intel Corporation 2009; Zhang, Dwarkadas, and Shen 2009) 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 oc-

cupancy 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.

8.3.3 Experiments

We evaluated CAFÉ's cache curve construction techniques using our ESX Server implementation. We first collected the miss-ratio curves for various SPEC CPU2000 benchmarks (`mcf`, `swim`, `twolf`, `equake`, `gzip`, and `perlbmk`), 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, with CAFÉ generating the miss-ratio curves, configured to construct the curves at benchmark completion time.

Figure 8.9 compares the miss-ratio curves of the benchmarks obtained by CAFÉ with those obtained by page coloring. In most cases, the MRC shapes and absolute MPKI values match reasonably well. However, in Figure 8.9(a), the MRC generated by CAFÉ for `mcf` 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 `mcf` 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 monotonocity enforcement.

To analyze this further, Figure 8.10 shows separate MRCs generated by CAFÉ for `mcf` with different co-runners, `swim` and `gzip`. The MRC generated when `mcf` is running with `gzip` is flat because `mcf` only has updates at the highest occupancy point. The miss ratio of `mcf` at the highest occupancy point is a factor of sixty more than the miss ratio of `gzip`, 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 `swim` matches the MRC obtained by page coloring closely.

8.3.4 Discussion

Our online technique for MRC construction builds upon our cache occupancy estimation model. While the MRCs generated for a working system in Section 8.3.3 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,

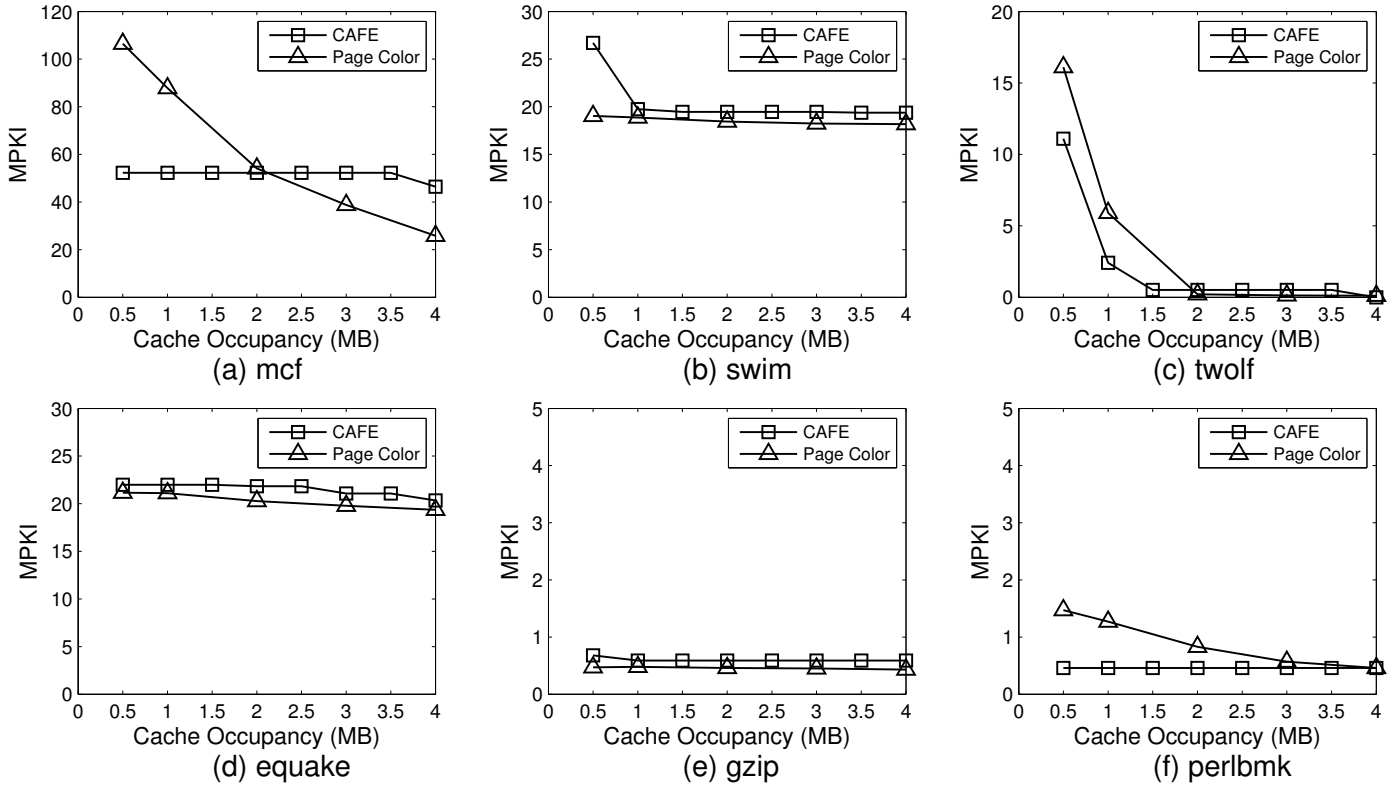


FIGURE 8.9
Miss-ratio curves (MRCs) for various SPEC CPU workloads, obtained online by CAFÉ versus offline by page coloring

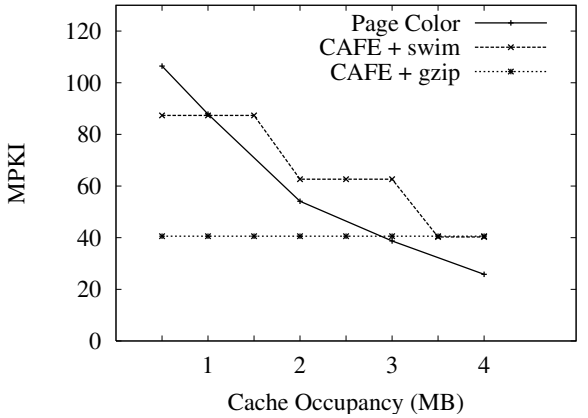


FIGURE 8.10
MRC for `mcf` with different co-runners

as observed in Figure 8.10. 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 8.3.3 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.

8.4 Cache-Aware Scheduling

In this section, we present higher-level scheduling policies that leverage CAFÉ's low-level methods for estimating cache occupancies and generating cache utility curves. We first examine the issue of fairness in CMPs, and present a new *vtime compensation* technique for improving CMP fairness in proportional-share schedulers. Next, we show how to use cache utility functions for estimating the impact of co-runner placements via a novel *cache divvying* approach. The scheduler considers new co-runner placements periodically, in order to maximize aggregate throughput. Unless otherwise stated,

all scheduling experiments in this section were conducted using the same system configuration as in Section 8.3.

8.4.1 Fair Scheduling

Operating systems and hypervisors are designed to multiplex hardware resources across multiple workloads with varying demands and importance. Administrators and users influence resource allocation policies by specifying settings such as priorities, reservations, or proportional-share weights. Such controls are commonly used to provide differential quality of service, or to enforce guaranteed service rates.

When all workloads are assigned equal allocations, *fairness* implies that each workload should receive equal service. More generally, a scheduler is considered fair if it accurately delivers resources to each workload consistent with specified allocation parameters.

Fair scheduling requires accurate accounting of resource consumption, although few systems implement this properly (Zhang and West 2006). For example, if a hardware interrupt occurs in the context of one workload, but performs work on behalf of a different workload, then the interrupt processing cost must be subtracted from the interrupted context and added to the workload that benefited. The VMware ESX Server scheduler (VMware, Inc. 2009), used for our experiments, implements proper accounting for interrupts, bottom halves, and other system processing; we extended this with cache-miss accounting for CAFÉ.

8.4.1.1 Proportional-Share Scheduling

In this work, we focus on *proportional-share* scheduling. Resource allocations are specified by numeric *shares* (or, equivalently, *weights*), which are assigned to threads that consume processor resources.[5] A thread is entitled to consume resources proportional to its share allocation, which specifies its importance relative to other threads.

Most proportional-share scheduling algorithms (Bennett and Zhang 1996; Parekh 1992; Stoica et al. 1996; Waldspurger and Weihl 1995; Zhang and Keshav 1991; Goyal, Vin, and Cheng 1996) use a notion of *virtual time* to represent per-thread progress. Each thread τ_i has an associated virtual time v_i, which advances at a rate that is directly proportional to its resource consumption q_i, and inversely proportional to its share allocation w_i:

$$v_i' = v_i + q_i/w_i \tag{8.18}$$

The scheduler chooses the thread with the minimum virtual time to execute next. For example, consider threads τ_i and τ_j with share allocations $w_i = 2$

[5] Although we use the term *thread* to be concrete, the same proportional-share framework can accommodate other abstractions of resource consumers, such as processes, applications, or VMs.

and $w_j = 1$. Thread τ_i is entitled to execute twice as quickly as τ_j; this 2:1 ratio is implemented by advancing v_i at half the rate of v_j for the same execution quantum q.

Some proportional-share schedulers differ significantly in their treatment of virtual time for threads blocked waiting on I/O or synchronization objects. For example, some algorithms partially credit a thread for time when it was blocked, while others do not. Here, we focus on CPU-bound threads, so these differences are not important; time spent blocking will be addressed in future work.

8.4.1.2 Fair Scheduling for CMPs

How should fairness be defined in the context of a CMP, where multiple processor cores may share last-level cache space, memory bandwidth, and other hardware resources? Accounting based solely on the amount of real time a thread has executed is clearly inadequate, since the amount of useful computation performed by a thread varies significantly with resource contention from co-runners.

One option is to define *cache-fair* as equal sharing of CMP cache space among co-running threads (Fedorova, Seltzer, and Smith 2006). However, this definition does not reflect the marginal utility of additional cache space, which typically differs across threads. For efficiency, we want to allocate more cache space to those threads which can utilize it most productively. Moreover, this definition of cache-fair does not facilitate our goal of proportional-share fairness, where different threads may be entitled to unequal amounts of shared resources.

We instead assume that a thread is entitled to consume *all* shared CMP resources while it is executing, including the entire last-level cache, in the absence of competition from co-running threads. At runtime, we dynamically estimate the actual performance degradation experienced by a thread due to co-runner interference and compensate it appropriately. Since most threads are negatively impacted to some degree by co-runners, this means that most threads will receive at least some compensation.

To quantify fairness, we first define the *weighted slowdown* for each thread to be the ratio of its actual execution time (in the presence of co-running threads) to its ideal execution time when running alone without co-runners, scaled by the thread's relative share allocation. The relative share allocation is, itself, the ratio of the local thread's weight to total weights of all competing threads. We then use the coefficient of variation of these per-thread weighted slowdowns as an unfairness metric; with perfect fairness, all weighted slowdowns are identical.

8.4.1.3 Virtual-Time Compensation

In a proportional-share scheduler, a convenient way to compensate threads for co-runner interference is to adjust the virtual time update in (8.18). In

particular, when a thread τ_i is charged for consuming its timeslice, we reduce its consumption q_i to account for the time it was stalled due to contention for shared resources. We call this virtual-time adjustment technique *vtime compensation*.[6] We present two different vtime compensation methods – an initial approach that compensates for conflict misses and an improved method that compensates for negative impacts on cycles per instruction (CPI).

Compensating for Conflict Misses

Our initial attempt at vtime compensation was designed to compensate a thread for conflict misses that it incurred while executing with co-runners and while on a ready queue waiting to be dispatched. We first estimate the cache occupancy that a thread τ_i would achieve without interference from other threads. Starting with (8.9), this reduces to

$$E_{i,NI} = E_i + \left(1 - \frac{E_i}{C}\right) m_i \tag{8.19}$$

where $E_{i,NI}$ represents the expected occupancy of thread τ_i with *no interference* from other threads.

We then use the *miss-rate* curve for τ_i to obtain two values: $M(E_i)$, the miss rate at E_i, and $M(E_{i,NI})$, the miss rate at $E_{i,NI}$, according to (8.9) and (8.19), respectively. Given our monotonicity enforcement for miss-rate curves, it must be the case that $M(E_i) \geq M(E_{i,NI})$.

Taking the difference between these two miss rates over τ_i's most recent timeslice, q_i, provides a measure of the *conflict misses* experienced by the thread. In practice, the latency of a cache miss is not constant, depending on several factors, including prefetching and contention for memory bandwidth. However, if we assume the average latency of a single last-level cache (LLC) miss is L, then we can approximate the stall cycles due to conflict misses, denoted by S_i, as

$$S_i = (M(E_i) - M(E_{i,NI})) \cdot L \tag{8.20}$$

Given this measure of the conflict stall cycles experienced by a thread, we modify the virtual time update from (8.18) accordingly:

$$v_i' = v_i + (q_i - S_i)/w_i \tag{8.21}$$

In (8.21), the updated virtual timestamp, v_i' factors in the amount of time τ_i stalls during its use of a CPU due to conflict misses with other threads. The number of conflict misses considers both the time during which τ_i executes *and* the time it waits for the CPU, since during this time its cache state may be evicted by other threads. This method of virtual time compensation attempts

[6]Similar compensation approaches could be used in proportional-share schedulers that are not based on virtual time. For example, in probabilistic lottery scheduling (Waldspurger and Weihl 1994), the concept of 'compensation tickets' introduced to support non-uniform quanta could be extended to reflect co-runner interference.

to benefit those threads that are affected by cache interference by reducing their effective resource consumption, which increases their scheduling priority.

Unfortunately, this approach requires miss-rate curves, which, as explained in Section 8.3, are difficult to derive accurately in the presence of co-runners competing for limited memory bandwidth. A related problem is modeling the average cache miss latency L, which may vary due to contention for memory bandwidth.

Compensating for Increased CPI

To address these issues, we revised our vtime compensation strategy to simply determine the *actual* cycles per instruction, CPI_{actual}, at the current occupancy, as well as the *ideal* cycles per instruction CPI_{ideal} if the thread were to experience no resource contention from other threads. We obtain CPI_{ideal} from the value at full occupancy in the CPKI curve. This is more robust than simply measuring the minimum observed CPI value, because the CPKI curve captures the average value over an interval, reducing sensitivity to phase transitions. Hence, our revised virtual time adjustment for τ_i becomes

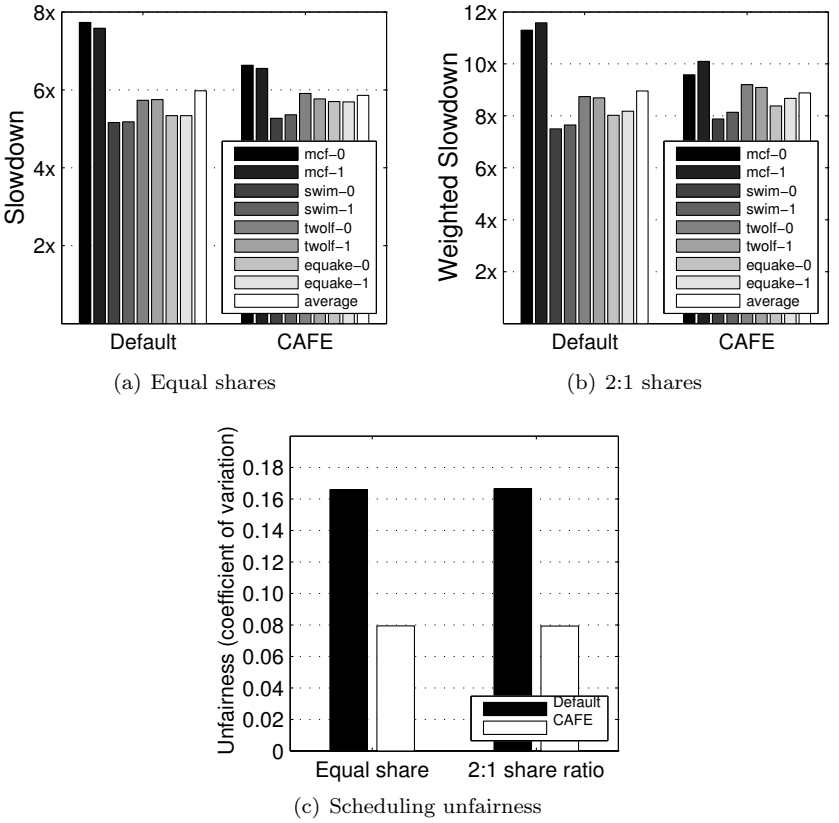
$$v_i' = v_i + \frac{CPI_{\text{ideal}}}{CPI_{\text{actual}}} \cdot q_i / w_i \tag{8.22}$$

This approach effectively replaces the use of miss-ratio curves with cache performance curves that provide CPI values at different cache occupancies (i.e., CPKI instead of MPKI). As a result, it reflects contention for *all* shared CMP resources, including memory interconnect bandwidth. Thus, compensating for negative impacts on CPI is simpler and more accurate than compensating only for cache conflict misses.

8.4.1.4 Vtime Compensation Experiments

We implemented vtime compensation in the VMware ESX Server hypervisor. ESX Server implements a proportional-share scheduler that employs a virtual-time algorithm similar to those described in Section 8.4.1.1. Our experiments ran two instances each of four different SPEC2000 benchmark applications: `mcf`, `swim`, `twolf`, and `equake`. In this case, we restricted all software threads to run on one package of the Dell PowerEdge machine, as described in Section 8.3. This meant that four cores were overcommitted with eight threads that were scheduled by the ESX Server hypervisor. The hypervisor was responsible for the assignment of threads to cores.

In Figure 8.11(a), all benchmark instances had equal share allocations, while in Figure 8.11(b), a 2:1 share ratio was specified for the two instances of each application. To evaluate the efficacy of vtime compensation, we measured per-application weighted slowdown, as defined in Section 8.4.1.2. The overall slowdown was calculated as the arithmetic mean of the weighted slowdowns of all the applications. Although CAFÉ only slightly reduces the average slowdown, it significantly reduces the variation in slowdowns experienced by all



(a) Equal shares

(b) 2:1 shares

(c) Scheduling unfairness

FIGURE 8.11
Vtime compensation

workloads. For both Figures 8.11(a) and (b), the slowdown experienced by `mcf` is much less when using CAFÉ compared to the default ESX Server scheduler.

Figure 8.11(c) plots the unfairness measured for the equal-share and 2:1-share ratio experiments. The unfairness metric is the coefficient of variation of the per-application weighted slowdowns, and vtime compensation improves it by approximately 50%. Overall, vtime compensation provides a slight increase in performance while reducing unfairness significantly.

8.4.2 Efficient Scheduling

Now we describe how CAFÉ's cache monitoring infrastructure can be leveraged to improve the performance of co-running workloads. We start by introducing the concept of *cache pressure*, which represents how aggressively a

thread competes for additional cache space. We then present a *cache divvying* algorithm, based on cache pressure, for approximating the steady-state cache occupancies of co-running threads. Using cache divvying to determine the performance impact of various co-runner placements, we demonstrate simple scheduler modifications for selecting good co-runner placements to maximize aggregate system throughput.

8.4.2.1 Cache Pressure

To understand cache pressure, recall that CAFÉ estimates the cache occupancy for a single thread using (8.9), which defines a recurrence relation between its previous and current occupancies. Since $(1 - E/C) \cdot m_l$ specifies the increase in occupancy, we define the cache pressure P_i exerted by thread τ_i as

$$P_i = (1 - E_i/C) \cdot M(E_i) \tag{8.23}$$

where C is the total number of cache lines in the shared cache, and $M(E_i)$ is the miss rate of τ_i at its current occupancy, E_i. In short, cache pressure reflects how aggressively a thread tends to increase its cache occupancy.

A key insight is that at equilibrium occupancies, the cache pressures exerted by co-running threads are either equal or zero. If the cache pressures are not equal, then the thread with the highest cache pressure increases its cache occupancy. We have observed that in most cases, co-running threads do not converge at equilibrium occupancies, but instead cycle through a series of occupancies with oscillating cache pressures.

Calculating a thread's cache pressure requires $M(E_i)$, which is obtained from its miss-rate curve. As explained earlier, since miss-rate curves are sensitive to contention for memory bandwidth and other dynamic interference, we instead construct miss-ratio curves, despite our desire to examine time-varying behavior. To translate MRCs that track MPKI values into misses per cycle, we normalize each point on the discrete curve by the ideal CPI for the corresponding thread. While this is not completely accurate, it nonetheless provides a practical way to generate approximate miss-rate curves that are not sensitive to interference from co-runners.

8.4.2.2 Cache Divvying

Using the insight above that cache pressures of co-running threads should match at equilibrium occupancies, we are able to estimate their average occupancies, enabling us to predict how the cache will be divided among them. Our *cache divvying* technique does not control how cache lines are actually allocated to threads, but rather serves to predict how cache lines would be allocated given their current occupancy and working-set demands. It also captures the average occupancies of co-running threads that cycle through a series of occupancy values at equilibrium.

Algorithm 8.1 summarizes the cache divvying strategy, assuming the cache is initially empty. In reality, each thread, τ_i, will have a potentially non-zero

Algorithm 8.1: Cache Divvying

```
// initialize surplus cache lines S
S = C;
foreach τ_i do
    E_i = 0; // initial occupancy
end
repeat
    // reset max pressure
    P_max = 0; foreach τ_i do
        // pressure at current occupancy
        P_i = (1 − E_i/C) · M(E_i);
        if P_i > P_max then
            // record thread with max pressure
            P_max = P_i;
            max = i;
        end
    end
    // greedily assume chunk of size B
    // allocated to thread with max pressure
    E_max = E_max + B;
    S = S − B;
until S = 0 or ∀P_i = 0;
```

current occupancy, E_i. The algorithm compares the pressures of each thread at their initial occupancies, by using miss-rate information obtained from MRC data. The thread with the highest pressure is assumed to be granted a chunk of cache. The allocatable chunk size, B, is configurable, but serves to limit the number of iterations of the algorithm required to predict steady-state occupancies for the competing threads. In practice we have found that setting B to one-eighth or one-sixteenth of the total cache size works well with our MRCs, which are also quantized using discrete cache occupancy values.

During each iteration, the thread with the highest pressure increases its hypothetical cache occupancy. This in turn affects its current miss rate, $M(E_i)$, and hence its current pressure, P_i, for its new occupancy. As the pressure from a thread subsides, its competition for additional cache lines diminishes. When the entire cache is divvied, or when all pressures reach zero, the algorithm terminates, yielding a prediction of cache occupancies for each co-runner.

Figure 8.12 shows Algorithm 8.1 used with our simulator for a dual-core system as described in Section 8.2.2. Cache divvying is used to predict the occupancies for six pairs of co-runners, separated by vertical dashed lines in the figure. In each case, the chunk size, B, is set to one-sixteenth of the cache size (i.e., 256 KB). Each co-runner generates 10 million interleaved cache references from a Valgrind trace. While this is insufficient to lead to full cache occupancy in all cases, results show that predicted and actual occupancies are almost always within one chunk size of the actual occupancy. This suggests

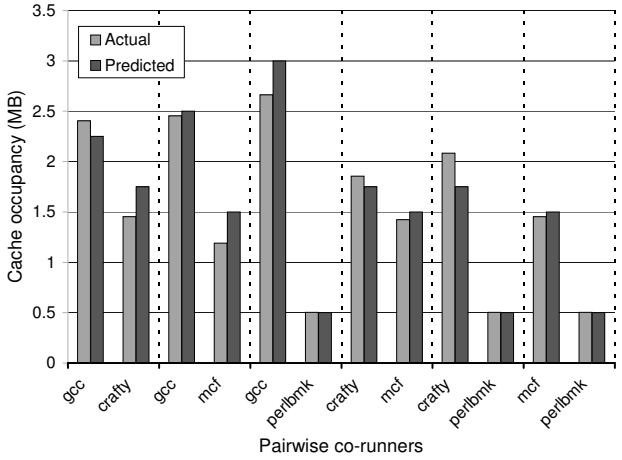


FIGURE 8.12
Cache divvying occupancy prediction

cache divvying is an accurate method of determining cache shares among co-runners. We are investigating its accuracy on architectures with higher core counts.

8.4.2.3 Co-Runner Selection

Cache divvying provides the ability to predict the equilibrium occupancies achieved by workloads co-running on a shared cache. This information can be used in CPU scheduling decisions to enhance overall system throughput.

We extended the VMware ESX Server scheduler with a simple heuristic. A user-level thread periodically snapshots the miss-ratio curves generated by CAFÉ and evaluates various co-runner pairings using *cache divvying* to predict their associated equilibrium occupancies. Based on a workload's estimated occupancy, we predict its miss ratio by consulting the workload's miss-ratio curve. We employ a simple approximation to convert the predicted miss ratio into a time-based miss rate, multiplying the workload's miss ratio by $1/CPI_{\text{ideal}}$, its instructions-per-cycle metric at full occupancy. The pairing which achieves the smallest aggregate *conflict miss rate* is chosen and communicated to the scheduler, which migrates threads to implement the improved placements.

The conflict miss rate is the miss rate in excess of what a thread experiences at full cache occupancy. By selecting pairings which reduce aggregate conflict misses, CAFÉ tries to improve performance as well as fairness. While we have demonstrated one practical heuristic incorporating cache divvying predictions, many other scheduler optimizations could benefit from this information.

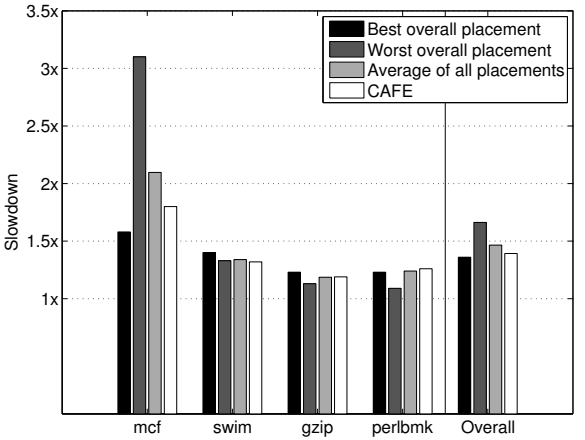


FIGURE 8.13
Co-runner placement

8.4.2.4 Co-Runner Selection Experiments

To evaluate our implementation of the co-runner placement heuristic in the ESX Server scheduler, we used the SPEC2000 benchmarks `mcf`, `swim`, `gzip`, and `perlbmk`, each running on a separate core. To focus on the effectiveness of CAFÉ at finding good co-runner placements, we restricted the workloads to execute on a single package containing two dual-core CMPs, each with its own last-level cache.

As before, we use the average of the per-application slowdowns as the metric for overall efficiency and their coefficient of variation as the metric for unfairness. At the start of the experiment, the co-runner pairings were manually selected to be the pairing that was determined to result in the worst overall performance (`mcf` paired with `swim` and `perlbmk` paired with `gzip`).

Note that in Figure 8.13, the 'Worst overall placement' column for each separate workload shows the slowdown of that benchmark when running in the worst overall configuration. As can be seen, some benchmarks do not suffer as much as others in this worst-case configuration, but `mcf` was the one that incurred significant slowdown. Notwithstanding, the rightmost 'Overall' column shows that when `mcf` experiences its worst slowdown that is when we have the worst overall slowdown across all workloads.

As Figure 8.13 shows, CAFÉ was able to achieve performance close to the best overall placement by adjusting the workload assignments to better cores. CAFÉ co-runner placement reduces unfairness by 24% and improves performance by 5% compared to the average of all placements. Compared to the worst overall placement, CAFÉ reduces unfairness by 64% and improves performance by 16%.

8.5 Related Work

The focus of this chapter encompasses several areas of related work, from shared-cache resource management to co-scheduling of threads on parallel or multi-core architectures. In the area of shared-cache resource management, there is a significant literature on cache partitioning, using either hardware or software techniques (Albonesi 1999; Chang and Sohi 2007; Dybdahl, Stenström, and Natvig 2006; Iyer 2004; Kim, Chandra, and Solihin 2004; Liu, Sivasubramaniam, and Kandemir 2004; Rafique, Lim, and Thottethodi 2006; Ranganathan, Adve, and Jouppi 2000; Srikantaiah, Kandemir, and Irwin 2008; Suh, Rudolph, and Devadas 2004). 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, Kim, Chandra, and Solihin (2004) showed significant variation in execution times of SPEC benchmarks, depending on co-runners competing for shared resources.

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 (Suh, Devadas, and Rudolph 2001; Suh, Rudolph, and Devadas 2004). 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 (Cho and Jin 2006; Liedtke, Härtig, and Hohmuth 1997; Lin et al. 2008).

A significant challenge with cache partitioning is deriving the optimal allocation size for a workload. One way to tackle this problem is to construct cache utility functions, or performance curves, that associate workload benefits (e.g., in terms of miss ratios, miss rates, or CPI) with different cache sizes. In particular, methods to construct miss-ratio curves (MRCs) have been proposed that capture workload performance impacts at different cache occupancies, but either require special hardware (Qureshi and Patt 2006; Suh, Devadas, and Rudolph 2001; Suh, Rudolph, and Devadas 2004) or incur high overhead (Berg, Zeffer, and Hagersten 2006; Tam et al. 2009).

The Mattson Stack Algorithm (Mattson et al. 1970) can derive MRCs by maintaining an LRU-ordered stack of memory addresses. RapidMRC uses this algorithm as the basis for its online MRC construction (Tam et al. 2009). This requires hardware support in the form of a *Sampled Data Address Register* (SDAR) in the IBM POWER5 performance monitoring unit to obtain a stream of memory addresses that match a pre-specified selection criterion. The total cost of online MRC construction is several hundred milliseconds, with more than 80 ms 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 overcommitted environments.

In contrast, we deploy an online method to construct MRCs and other cache-performance curves efficiently, 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, CAFÉ does not require an offline reference point to account for vertical shifts in the online curves due to phase transitions 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.

Other researchers have inferred cache usage and utility of different cache sizes. In CacheScouts (Zhao et al. 2007), for example, hardware support for monitoring IDs and set sampling are used to associate cache lines with different workloads, enabling cache occupancy measurements. However, the use of special IDs differs from our occupancy estimation approach, which only requires currently available performance monitoring events common to modern CMPs.

Given cache utility curves, we attempt to perform fair and efficient scheduling of workloads on multiple cores. Fedorova, Seltzer, and Smith (2006) devised a cache-fair thread scheduler that redistributes CPU time to threads to account for unequal cache sharing. This work assumes that different workloads competing for shared resources should receive equal cache shares to be fair, regardless of different memory demands from workloads. A two-phase procedure is employed, first computing the fair cache miss rate of each thread, followed by adjustments to CPU allocations. Computing fair cache miss rates requires sampling a subset of co-runners followed by a linear regression and is potentially expensive. In contrast, we derive a workload's current and fair CPI values inexpensively and then perform vtime compensation to improve fairness.

8.6 Conclusions and Future Work

This chapter introduces several novel techniques for chip-level multiprocessor resource management. In particular, we focus on the management of shared last-level caches and their impact on fair and efficient scheduling of workloads. Toward this end, our first contribution is the online estimation of cache oc-

cupancies for different threads, using only performance counters commonly available on commodity processors. Simulation results verify the accuracy of our mathematical model for cache occupancy estimation.

Building on occupancy estimation, we demonstrate how to dynamically generate cache performance curves, such as MRCs, that capture the utility of cache space on workload performance. Empirical results using the VMware ESX Server hypervisor show that we are able to construct per-thread MRCs online with low overhead, in the presence of interference from co-runners. We show how duty cycle modulation can be used to help a thread increase its cache occupancy by reducing interference from co-runners. This approach facilitates obtaining a wide range of occupancy data points for MRCs.

Our fast online MRC construction technique is used as part of a cache divvying heuristic to predict the average occupancies of a set of co-running workloads. Simulation results show this to be an effective method of using MRCs to estimate the expected occupancies if two or more workloads were to co-execute and compete for cache space. Cache divvying forms the basis of our co-runner selection strategy, which partitions threads across separate CMPs. By carefully partitioning threads, we avoid potentially bad groupings of co-runners that could negatively impact the shared last-level cache on the same CMP. Experiments show that for a group of SPEC CPU workloads, we are able to reduce slowdown by as much as 5% in the average case and 16% in the best case.

Finally, we attempt to improve fairness by compensating a workload for the resource conflicts it experiences when co-running with other workloads. Our vtime compensation technique accounts for the time a thread is stalled contending for resources, including the stall cycles caused by last-level cache conflict misses and memory bus access. Estimates of performance degradation experienced by a thread due to co-runner interference are calculated online. Results show as much as 50% improvement in fairness using vtime compensation.

While we have presented several new online techniques for CMP resource management, a variety of interesting research opportunities remain. We are exploring various approaches for improving CAFÉ's ability to generate accurate cache performance curves at all occupancy points. We continue to investigate new scheduling heuristics that leverage our cache monitoring capabilities, and we are examining applications of vtime compensation to other problems, such as NUMA locality management. We also plan to extend our modeling techniques to address the impact of threads that block waiting for events such as I/O completion, and to incorporate the effects of data sharing and constructive interference between threads. Finally, we are actively exploring ways to extend and integrate our software techniques with future hardware, such as architectural support for cache quality of service (QoS) monitoring and enforcement, and large-scale CMPs containing tens to hundreds of cores.