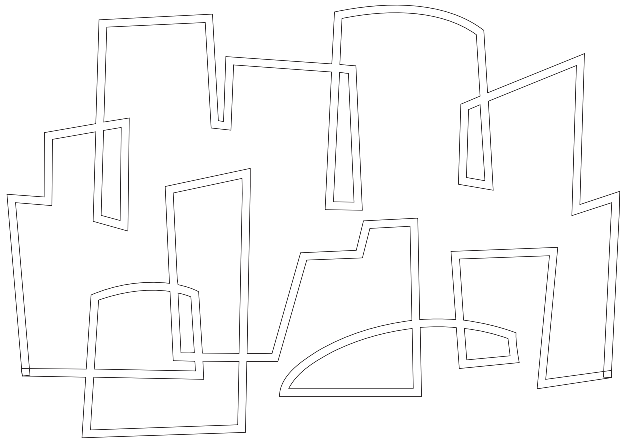
A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning

Edward Suh, Srinivas Devadas, Larry Rudolph

Proceedings of the 8th International Symposium on High-Performance Computer Architecture, February 2002
2001, July

Computation Structures Group
Memo 443



A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning

Computation Structures Group Memo 443
July 2001

G. Edward Suh, Srinivas Devadas and Larry Rudolph
`email: {suh,devadas,rudolph}@mit.edu`

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work is provided in part by the Defense Advanced Research Projects Agency under the Air Force Research Lab contract F30602-99-2-0511.

A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning

G. Edward Suh, Srinivas Devadas, and Larry Rudolph
Laboratory for Computer Science
MIT
Cambridge, MA 02139
email: {suh,devadas,rudolph}@mit.edu

Abstract

The memory hierarchy in modern computing systems is typically time-shared and space-shared amongst multiple processes and threads, some of which execute simultaneously. Memory contention can significantly degrade the performance of running processes. Cache hit counters found in modern microprocessor provide a limited picture as to the memory needs of processes.

We propose a low overhead, on-line memory monitoring scheme, utilizing a set of counters, that enables software to accurately estimate the isolated miss-rates of each process as a function of cache size under the standard LRU replacement policy. The data collected by these monitors serves as input to an analytical model of cache and memory behavior, which produces an accurate overall miss-rate for the collection of processes time-sharing or space-sharing a cache.

We target the minimization of the overall miss-rate in scheduling and partitioning. To the best of our knowledge, our scheduling algorithm is the first to optimize memory performance taking into account memory contention and timesharing effects. Our results show that these effects can be significant, and if ignored by the scheduler, result in suboptimal performance. We also present results that show that on-line monitoring can improve memory and overall performance by driving the optimal partitioning of cache memory across time-shared processes.

1 Introduction

This paper presents a low-overhead, on-line memory monitoring scheme that is more useful than the simple cache hit counters currently available. The scheme becomes increasingly important as more and more processes and threads share memory resources in modern microprocessors and in shared-memory multiprocessors (SMPs) [3, 8, 9], which have become a basic building block for modern high-performance computer systems. In the near future, advanced microprocessors may have multiple processors on a chip sharing a certain level of the memory hierarchy [4]. Or they may consist of simultaneous multithreading (SMT) systems executing multiple threads simultaneously, and therefore effectively have multiple processors sharing all levels of the memory hierarchy from L1 caches to main memory [18, 11, 5]. Regardless of whether a single process executes on the machine at a given point in time, or multiple processes execute simultaneously, modern systems are typically time-shared. Memory in most modern computing systems is thus both *space-shared* and *time-shared*.

Since multiple contexts can interfere in memory or caches, the performance of a process can depend on the actions of other processes. For example, a process experiences additional start-up misses at the beginning of each time slice in time-shared systems. On the other hand, the amount of memory space allocated to each process directly depends on simultaneously-executing processes on other CPUs with which it shares the memory. Considering that bandwidth and latency of memory are major bottlenecks in modern computing systems, it is very important to minimize contention caused by space-sharing and time-sharing.

Despite the importance of optimizing memory performance for multi-tasking situations, most published research focuses only on improving the performance of a single process. Improving the performance of an individual process may also improve the overall performance of multi-tasking systems, however, one can do better. Optimizing memory contention of multiple processes is virtually impossible without run-time information. The processes that share memory are only known at run-time. Moreover, the memory reference characteristic of each process, which significantly affects contention with other processes, heavily depends on inputs to the process and the phase of execution. We believe that on-line memory monitoring schemes are essential to improving memory performance in multi-tasking systems.

Hardware cache monitors in commercial, general-purpose microprocessors (e.g., [19]) count the total number of misses. But these monitors are rarely used in practice, other than for pure performance monitoring. The use of memory monitors that measure the footprint of each process so as to schedule processes considering memory requirements has been proposed [2]. Unfortunately, information available from existing memory monitors is not nearly enough to answer various questions associated with multi-tasking. For example, consider the question of scheduling processes: *How many and which jobs should execute simultaneously?* One can imagine choosing jobs such that the sum of the individual memory footprints is roughly the size of the shared memory. However, memory contention due to simultaneously-running jobs cannot be modeled using footprints. We show, in this paper, that this effect can be significant, and therefore footprints alone are not enough to schedule processes optimally. Moreover, the effects of time-sharing on memory performance cannot be modeled given information from conventional memory monitors.

The memory monitoring scheme presented in this paper requires small modifications to the TLB, L1, and L2 cache controllers and the addition of a set of counters. Despite the simplicity of the hardware, these counters provide isolated miss-rates of each running process as a function of cache size under the standard LRU replacement policy. [1] Moreover, the monitoring information can be used to dynamically reflect changes in process' behavior by properly weighting counters' values.

The information from the memory monitors is analyzed using an analytical framework, which models the effects of memory interference amongst simultaneously-executing processes as well as time-sharing effects. The counter values alone only estimate the effects of reducing

[1] Previous approaches only produce a single number corresponding to one memory size.

cache space for each process. However, when used in conjunction with the analytical model, they can provide an accurate estimate of the overall miss-rate of a set of processes time-sharing and space-sharing a cache. The metric can then be used in software schedulers and partitioners within operating systems, where scheduling and partitioning decisions to optimize cache/memory usage would be based, at least partly, on model outputs.

First, the problem of scheduling processes for a shared-memory multiprocessor system to minimize processor idle time due to either inactive processors or page faults is solved.[2] Trace-driven simulation results show that job schedules can significantly affect the memory performance. Further, we show that our model-based scheduling method finds close to optimal schedules and significantly improves the memory miss-rate and overall performance. To the best of our knowledge, our scheduling methodology is the first to optimize memory performance taking into account memory contention and timesharing effects.

Cache partitioning experiments also demonstrate the usefulness of our memory monitoring scheme and the analytical model. Cache space is explictly allocated to each process to minimize the misses caused by conflicts amongst processes in time-shared systems. We show that cache space can be managed with on-line cache monitoring, and we develop a partitioning scheme for set-associative caches.

The rest of this paper is organized as follows. In Section 2, we motivate miss-rate curves as a foundation for memory performance optimization. In Section 3, we describe the counter scheme and its implementation. Section 4 describes the analytical model which incorporates cache contention effects due to simultaneously-executing processes. Section 5 and 6 validate our approach by targeting memory-aware scheduling and cache partitioning, respectively. We describe an algorithm for each problem, and then provide experimental results. Related work is discussed in Section 7. Finally, Section 8 concludes the paper.

2 Motivating Miss-Rate Curves

We define the isolated miss-rate of process i with cache space x, namely $m_i(x)$, as the miss-rate that process i experiences when the process is isolated (without other competing processes) and uses x cache blocks. Effectively, this curve represents the miss-rate when a process occupies only a part of the cache.

In this section, we show that miss-rate curves provide critical information which allows us to perform memory-aware scheduling and partitioning, thereby motivating a scheme for on-line computation of these curves (cf. Section 3).

2.1 Space-Sharing

Multiple processes or threads are allowed to run on a multi-threaded machine sharing a cache or memory. We can ask the question: *Which processes should run together?* We may not be able to answer this question by just looking at the footprints of the different processes. For small caches, each of them will likely have a footprint larger than the cache size.

Now, consider the miss-rate curves for three different processes from SPEC CPU2000 [7] shown in Figure 1. These curves give much more information than a single footprint

[2]The number of page faults can vary significantly based on what other processes share the memory.

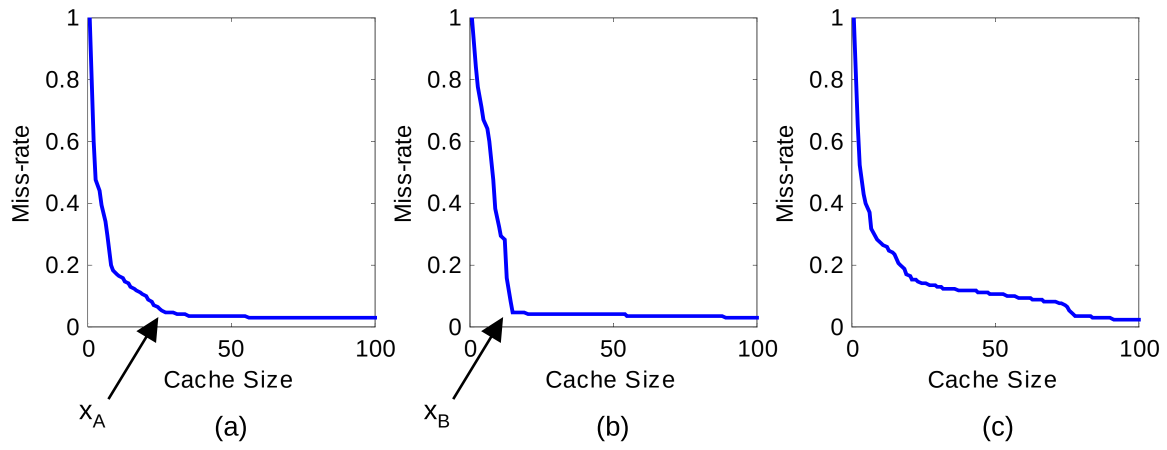


Figure 1: (a) Miss-rate curve for process A (`gcc`). (b) Miss-rate curve for process B (`swim`). (c) Miss-rate curve for process C (`bzip2`). Clearly, process A's miss-rate does not reduce very much after the point marked x_A. Similarly, for process B after the point marked x_B. If $x_A + x_B$ is less than the total cache size available, then it is likely that processes A and B can both be run together, achieving good performance, especially if they are restricted to occupy an appropriate portion of the cache. On the other hand, process C has a different type of miss-rate curve, and will likely not run well with either A or B.

number, and this information can be very relevant in scheduling processes, i.e., deciding which processes will space-share a cache, and with what allocations. In the figure, if $x_A+x_B \leq S$, where S is the size of the cache, then assigning x_A space in the cache to A, and x_B to B, and running both A and B simultaneously, *will* result in a low miss-rate.

However, if the cache is not partitioned explicitly across A and B, then it is possible that each process will try to take over all of the cache space. The overall miss-rate when A and B are run together will depend in a more complex way on the isolated miss-rate curves of A and B. Our analytical model (cf. Section 4) incorporates this effect of contention amongst processes and computes overall miss-rate curves, given isolated miss-rate curves.

2.2 Time-Sharing

Now consider a time-shared system, where the processes A, B and C are run in some sequence. If the footprints of the processes are larger than the cache size, then each process will likely take over the entire cache, *if it runs for a long enough period of time*. However, if the processes run for a short period, they will only use a fraction of the cache.

The isolated miss-rate curves do not directly give information about how much cache a process will use as a function of time, but we can derive this information, by using our analytical model that estimates transient miss-rate curves given the isolated miss-rate curves (cf. Section 4). The transient miss-rate curves tell how much cache is used by the process as a function of time. Finally, multiple transient miss-rate curves can be combined to produce an overall miss-rate when multiple processes sequentially time-share a CPU and cache.

Estimating an overall miss-rate allows us, for instance, to choose a time quantum from within a range, that optimizes cache performance. Using the model, we can also obtain information about whether dedicating cache space to a process is helpful or not. Dedicating cache space to a process helps the process in its startup transient because it is not starting with an empty cache. On the other hand, it restricts the amount of space that other processes have, when they run. Our model quantitatively characterizes this tradeoff.

2.3 On-Line Computation

Miss-rate curves will change with time, since the memory reference characteristics of processes will change with time. Off-line profiling is limited in dealing with dynamic changes in process behavior. Optimal memory-aware scheduling and partitioning requires that we continually compute miss-rate curves on-line, so we can track process behavior.

3 Using Counters To Compute Miss-Rate Curves

To be useful in optimizing cache performance, monitoring schemes should provide information to estimate the performance for different cache configurations or allocations. For example, to improve the performance by explicitly partitioning cache space, we need to know the effects of increasing and decreasing the space for each job. Existing cache monitors are limited in a sense that they only provide one number representing the performance for the current cache configuration.

The first question that arises is: *What information is required to predict performance without actually trying a new configuration?* We answered this question in part in Section 2, where we showed that the isolated miss-rate curves for a process could potentially be used to advantage in scheduling and partitioning decisions.

The next question is: *How can this information be obtained at run-time?* This section answers this question for multi-tasking systems by proposing a new monitoring scheme with multiple counters. Our monitoring scheme counts the number of references for each process, and estimates the miss-rate of each process as a function of cache space. We describe how counters instrumented in fully-associative and set-associative caches can be used to compute the miss-rate curves, described in Section 2, in an on-line fashion. The main advantages of this scheme are its inherent simplicity, and its on-line nature, which allows us to dynamically adapt to changes in process behavior.

The rest of this section discusses the implementation and the cost of the new cache monitoring scheme for different levels of the memory hierarchy assuming that caches are managed by the standard LRU replacement policy. First, the simpler case of fully-associative caches, e.g., main memory, is discussed. Then, this implementation is extended to set-associative caches.

3.1 Fully-Associative Caches

We assume a fully associative cache with a least recently used (LRU) replacement policy. We will compute the miss-rate curve $m_i(x)$ for a process by computing the hits obtained by the x *most recently used* blocks. This is equivalent to the ideal miss-rate curve obtained by running

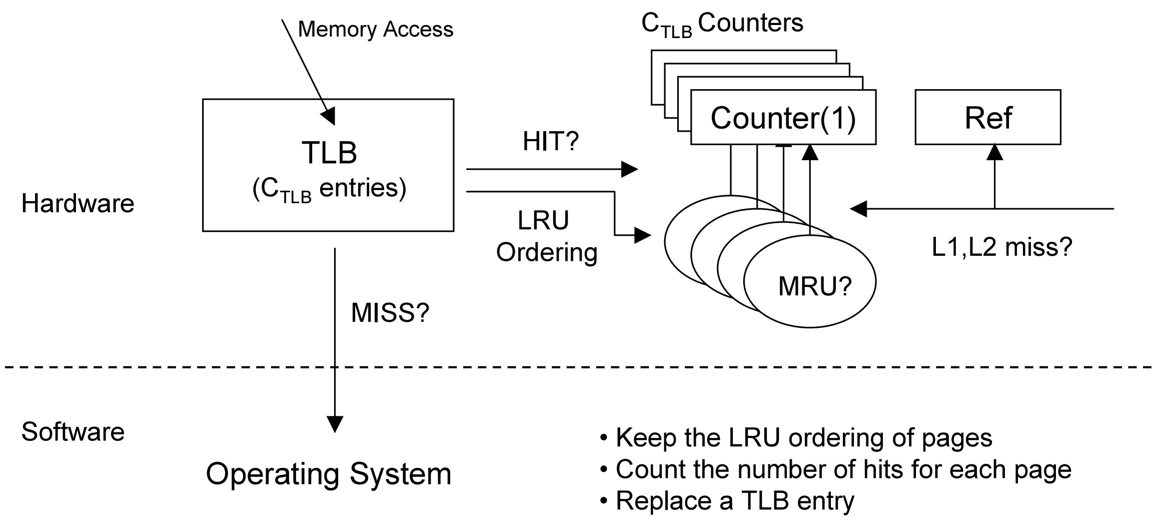


Figure 2: The implementation of memory monitors for main memory.

the process for fully-associative caches of different sizes which all have LRU replacement, and computing the miss-rate. This is because a fully-associative cache of size x will keep the x most recently used blocks of a fully associative cache of size greater than x.

Main memory is a fully-associative cache that is often managed by operating systems. For the standard LRU replacement policy, either hardware or the operating system keeps the ordering of pages in terms of which page is more recently used (we will refer to this ordering as the LRU ordering).

First, the number of memory references for a process can be counted by increasing a counter (ref_i) for every main memory access. To estimate a miss-rate curve $m_i(x)$, we use the LRU ordering of each page and count the number of hits in the k^{th} most recently used page ($counter_i(k)$). For example, $counter_i(1)$ is the number of hits in the most recently used page, and $counter_i(2)$ is the number of hits in the second most recently used page. Then, $m_i(x)$ and $counter_i(k)$ have the following relation.

$$counter_i(k) = (m_i(k-1) - m_i(k)) \cdot ref_i. \tag{1}$$

Since $m_i(0) = 1$, we can calculate the miss-rate curve recursively.

In practice, accesses to main memory are serviced by both hardware and software as shown in Figure 2. Every memory access is first looked up in a TLB to be converted from a virtual address to physical address. If the accessed page is found in the TLB, the memory access can be serviced by hardware without intervention by the operating system. Assuming the TLB is a fully-associative cache with LRU replacement, the most recently accessed pages will have their translation information in the TLB. Therefore, accesses to C_{TLB} most recently used pages should be counted using hardware counters, where C_{TLB} is the number of TLB entries. The number of accesses (ref_i) is also counted in hardware. On the other hand, the operating system services a memory access if it causes a TLB miss. Therefore, the operating system maintains the rest of counters[3] and the LRU information of the corresponding pages.

[3]The number of counters maintained by the operating system will be the total number of pages minus

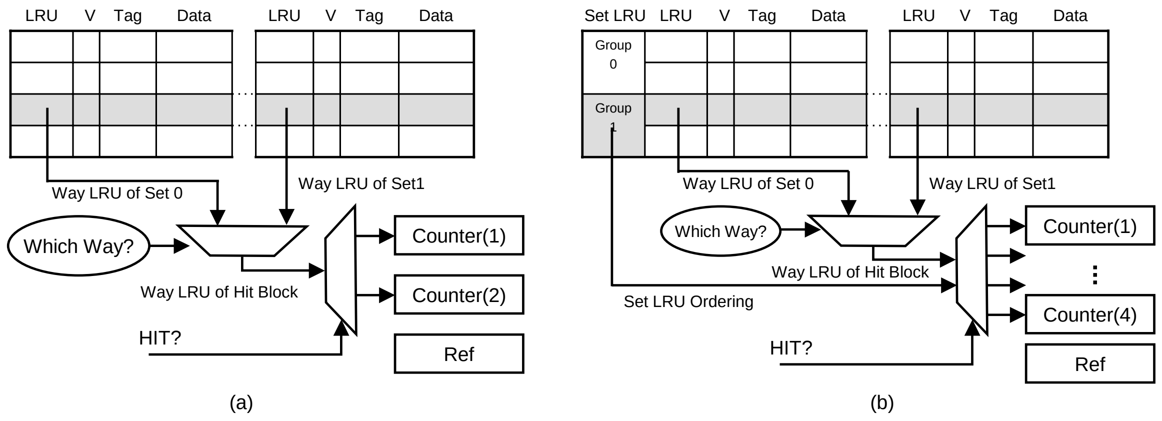


Figure 3: The implementation of memory monitors for 2-way associative caches. On a cache access, the LRU information is read for the accessed set. Then the counter is incremented based on this LRU information if the access hits on the cache. The reference counter is increased on every access. (a) The implementation that only uses the LRU information within a set. (b) The implemenation that uses both the way LRU information and the set LRU information.

We wish to compute isolated miss-rate curves for each process, and therefore, to exclude the interference with other processes, the operating system maintains the LRU ordering for each process separately. Also, the operating system memorizes the values of the hardware counters for the process context-switching out, and reinitializes the counters with the values for the starting process. multiple blocks are the most recently used within In a multiprocessor system, each processor will have its own set of hardware counters and will execute only one process at a time, and so there is no inter-process interference.

Hardware/software costs associated with implementing a new monitoring scheme for main memory is negligible. In terms of hardware, we only added a set of counters, one for each TLB entry. Also, these counters are only increased if a memory access misses on both L1 and L2 caches. Therefore, counting accesses to main memory does not introduce additional delay on any critical path. In terms of software, we only need tens of bytes for each page whose size is of the order of 4-KB. If even this overhead is too much, we can always have a counter per group of pages. We have found that having a counter per tens of pages still provides enough information for memory optimization.

3.2 Set-Associative Caches

Estimating the isolated miss-rate curves is a little more complicated for set-associative caches than fully-associative caches. Ideally, the LRU ordering of all cache blocks in the set-associative cache should be known so that the same counter scheme as fully-associative caches can be used. Unfortunately, in set-associative caches, LRU ordering is kept only within each set (We call this LRU ordering within a set as *way LRU ordering*).

If the cache has reasonably high associativity, e.g., 8-way, it is often good enough to

C_{TLB}.

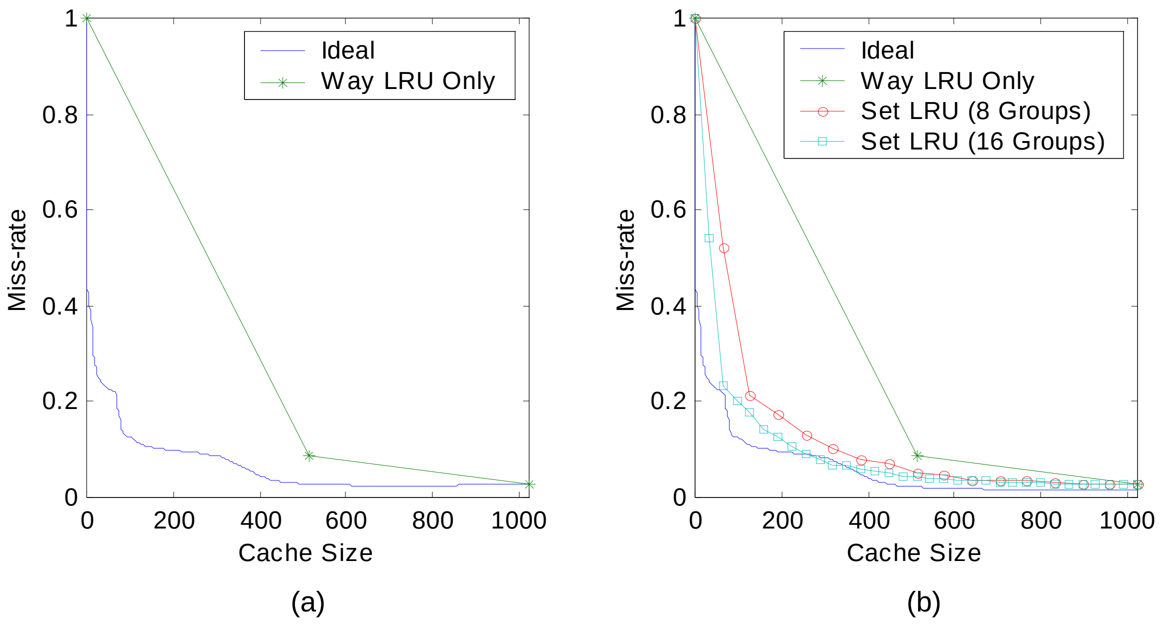


Figure 4: The estimated miss-rate curves using the set-associative cache monitor. The cache is 32-KB 2-way associative, and the benchmark is `vpr` from SPEC CPU2000. The ideal curve represents the case when you know the LRU ordering of all cache blocks. (a) Approximation only using the way LRU information. (b) Approximation using both the way LRU information and the set LRU information.

approximate the miss-rate curve from the LRU ordering within a set. If the cache is D-way associative, $D + 1$ counters are used for the entire cache. For every cache access, one counter (ref_i) counts the total number of accesses for process i. The remaining D counters ($counter_{APPROX,i}(k)$) approximate the miss-rate curve of the process. For a hit on k^{th} recently used cache block within a set, the corresponding k^{th} counter is incremented. Then, the miss-rate curve is approximated by

$$counter_{APPROX,i}(k) = (m_i(Q \cdot (k-1)) - m_i(Q \cdot k)) \cdot ref_i. \quad (2)$$

where Q is the number of sets. The Figure 3 (a) illustrates the implementation of this hardware counters when we only use the way LRU information.

Since the minimum monitoring granularity in our approximation method is a *way*, high-associativity is essential for accurate miss-rate curves. Although high-associativity can create a high implementation overhead, our partitioning experiments demonstrate that 8-way associative caches can provide reasonable information for partitioning. Moreover, content-addressable-memory (CAM) tags, which enable higher associative caches, becoming more attractive for low-power processors [20]. The SA-1100 StrongARM processor [10] contains a 32-way associative caches implemented with CAMs.

If the cache is low associative, the above approximation method often results in a miss-rate curve that is not accurate enough to be used for performance optimizations as shown in Figure 4 (a). In this case, we found that adding the LRU ordering for *sets* can significantly improve the granularity of cache monitoring, and the accuracy of the resulting miss-rate curve. Sets are devided into several groups and the LRU ordering is maintained for the groups (we call this LRU ordering as *set LRU ordering*). Using the set LRU information, we can tell which cache block is more recently used even within the same most recently used blocks within a set. Figure 4 (b) demonstrates the usefullness of this set LRU information. As shown in the figure, we can obtaion very accurate miss-rate curves if we keep the set LRU ordering for 8 or 16 groups. The implementation with the set LRU is shown in Figure 3 (b).

The same mechanism as fully-associative caches is used to obtain *isolated* miss-rate curves; That is, the operating system memorizes and reinitializes counter values at a context switch. Also, multiple sets of counters are maintained in the case of multiple simultaneous processes.

The counters for set-associative caches are all implemented in hardware. However, the overhead associated with this is minimal. Since only one counter is used for each *way*, only a few counters are added for the entire cache. Although the scheme needs multiple set of counters when multiple processes can access the cache simultaneously, the number of simultaneous processes is often limited to less than 8. Therefore, the total number of counters is of the order of one hundred.

3.3 Adapting to Dynamic Changes of Process Behavior

Since characteristics of processes change dynamically, the estimation of $m_i(x)$ should reflect the changes. But we also wish to maintain some history of the memory reference characteristics of a process, so we can use it to make decisions. We can achieve both objectives, by giving more weight to the counter value measured in more recent time periods.

When a process begins running for the first time, all its counter values are initialized to zero. At the beginning of each time quantum that process i runs, the operating system multiplies $counter_i(k)$ for all k and ref_i by $\delta = 0.5$. As a result, the effect of hits in the previous time slice exponentially decays, but we maintain some history.

4 Analytical Models

This section presents analytical methods that can model the effects of memory contention amongst simultaneously-running processes, as well as the effects of time-sharing, using the miss-rate curves from the memory monitoring scheme. The output miss-rate curve of the memory monitor alone has limited use, however, it can be used for any problem related to the effects of multi-tasking with the help of the model. First, a uni-processor cache model for time-shared systems is briefly summarized. Then, the model is extended to include the effects of memory contention amongst simultaneously-running processes.

4.1 Model for Time-Sharing

The time-sharing model from [15] estimates the overall miss-rate for a fully-associative cache when multiple processes time-share the same cache (memory) on a uni-processor system. There are three inputs to the model: (1) the memory size (C) in terms of the number of memory blocks (pages), (2) job sequences with the length of each process' time slice (T_i) in terms of the number of memory references, and (3) the miss-rate of each process as a function of cache space ($m_i(x)$). The model assumes that the least recently used (LRU) replacement policy is used, and there are no shared data structures among processes.

Let us consider a simple case when N processes execute in a round-robin fashion with fixed time quanta (T_i). The overall miss-rate for one round-robin cycle is estimated assuming that the miss-rate curves for the cycle is given. For the next set of time slices, the overall miss-rate can be re-estimated with a new set of miss-rate curves. First, the number of misses for each process' time-slice is estimated. Then, the overall miss-rate is obtained by combining the number of misses.

Define the footprint of process i ($x_i(t)$) as the amount of process i's data in the cache at time t where time t is 0 at the beginning of the process' time quantum. Then, $x_i(t)$ is estimated by the following recursive equation, once $x_i(0)$ is known;

$$x_i(t+1) = MIN[x_i(t) + m_i(x_i(t)), c_i], \tag{3}$$

where c_i is the maximum cache space that process i can consume. Without explicit partitioning, c_i is usually C for all processes.

Since the input miss-rate curve ($m_i(x)$) provides the probability to miss when x blocks are in the cache, the number of misses that process i experiences over one time quantum is given by

$$\text{miss}_i = \int_0^{T_i} m_i(x_i(t))dt. \tag{4}$$

Once the number of misses for each process is estimated, the overall miss-rate is straightforwardly calculated from those numbers.

$$\text{miss-rate}_{\text{overall}} = \frac{\sum_{i=1}^{N} \text{miss}_i}{\sum_{i=1}^{N} T_i} \tag{5}$$

From the above equations, the overall miss-rate of time-sharing systems can be obtained when the amount of data for each process at the beginning of its time quantum ($x_i(0)$) is known. Cache partitioning explicitly manages the amount ($x_i(0)$), therefore the model can be directly used for cache partitioning. $x_i(0)$ can also be modeled for more general cases.

4.2 Extension to Space-Sharing

The original model assumes only one process executes at a time. In this subsection, we describe how the original model can be applied to multiprocessor systems where multiple processes can execute simultaneously sharing the memory (cache). Although the model can be applied to more general cases, we consider the situation where all processors context switch at the same time. More complicated cases where each processor can context switch at a different time can be modeled in a similar manner.

Let us say that there are P processors that simultaneously access the memory. Since we assume that all P processors context switch at the same time, all processes in a time slice can be seen as one big process from the standpoint of memory. Therefore, we use two step approach to model both time-sharing and space-sharing. First, we use the original model to obtain the miss-rate curve for each time-slice considering all processes in the time slice as one big process. Then, the estimated miss-rate curves are processed by the model again to incorporate the effects of time-sharing.

What should be the miss-rate curve for each time slice? Since the model for time-sharing needs *isolated* miss-rate curves, the miss-rate curve each time-slice s is defined as the overall miss-rate of all processes in time slice s when they execute together without context switching using on the memory of size x. We call this miss-rate curve for a time slice as a combined miss-rate curve $m_{combined,s}(x)$. Next we explain how to obtain the combined miss-rate curves.

The simultaneously executing processes within a time slice can be modeled as time-shared processes with very short time slices. Therefore, the original model is used to obtain the combined miss-rate curves by assuming the time slice is $ref_{s,p}/\sum_{i=1}^{P} ref_{s,i}$ for processor p in time-slice s. $ref_{s,p}$ is the number of memory accesses that processor p makes over time slice s. The following paragraphs summarize this derivation of the combined miss-rate curves. Here, we use $m_{s,p}$ to represent the isolated miss-rate curve for the process that executes on processor p in time slice s.

Let $x_{s,p}(k_{s,p})$ be the number of memory blocks that processor p brings into memory after $k_{s,p}$ memory references in time slice s. The following equation estimates the value of $x_{s,p}(k_{s,p})$:

$$k_{s,p} = \int_0^{x_{s,p}(k_{s,p})} \frac{1}{m_{s,p}(x')} dx'. \tag{6}$$

Considering all P processors, the system reaches the steady-state after K_s memory references where K_s satisfies the following equation.

$$\sum_{p=1}^{P} x_{s,p}(\alpha(s,p) \cdot K_s) = x. \tag{7}$$

In the above equation, x is the number of memory blocks, and $\alpha(s,p)$ is the length of a time slice for processor p, which is equal to $ref_{s,p}/\sum_{i=1}^{P} ref_{s,i}$. From the steady-state, the combined miss-rate curve is given by

$$m_{combined,s}(x) = \sum_{p=1}^{P} \alpha(s,p) \cdot m_{s,p}(x_p(\alpha(s,p) \cdot K_s)). \tag{8}$$

Now we have the combined miss-rate curve for each time-slice. The overall miss-rate is estimated by using the original model assuming that only one process executes for a time slice whose miss-rate curve is $m_{combined,s}(x)$.

5 Memory-Aware Scheduling

This section describes the memory-aware scheduling algorithm beginning with the problem definition, assumptions, and some notation. Since the algorithm searches the space of potential schedules, we first present a way of specifying and evaluating a schedule.

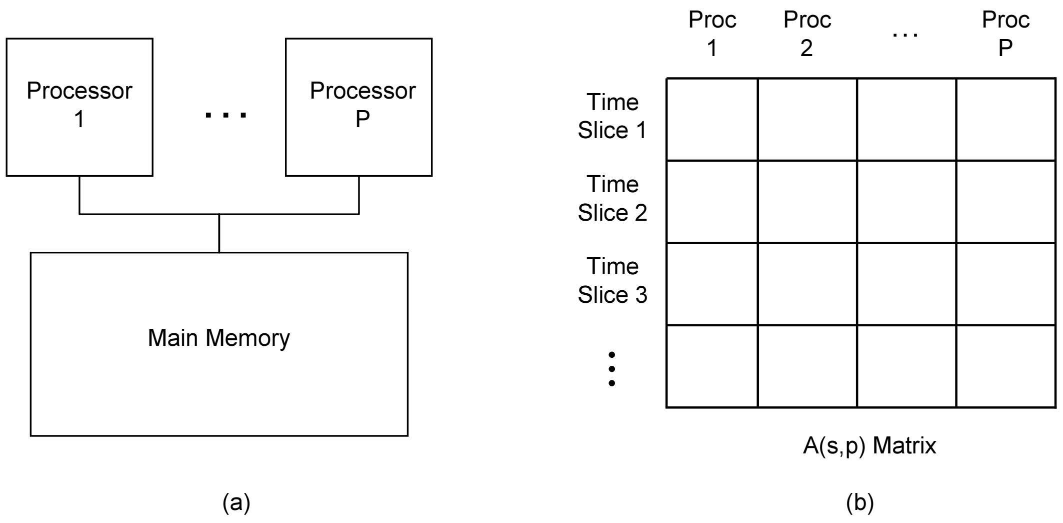


Figure 5: (a) A shared memory multiprocessor system with P processors. (b) Space-sharing and Time-sharing in multiprocessor system.

5.1 Scheduling Problem

We assume there are N processes ready to execute on a shared-memory system with P processors. Processors are identical and share the main memory, see Figure 5 (a). There may or may not be constraints in scheduling the ready processes. Constraints will merely affect the search for feasible schedules.

We will assume single-threaded processes, and a time-shared system, where processes are run for specified time quanta. We will further assume that all P processors context switch at the same time as would be done in gang scheduling [6]. These assumptions are not central to our approach, rather for the sake of brevity, we have focused on a basic scheduling scenario.

Since there are P processors, a maximum of P processes can execute at the same time. To schedule more than P processes, the system is time-shared. If the processes have large memory requirements, we consider coarse-grained time-sharing with large time quanta that amortize the context switching time. Alternatively, it is possible to consider a batch system where the processes execute to completion. Since this will require new scheduling assignments as each process finishes, we assume a time-shared system to simplify the exposition.

Our problem is to find the optimal scheduling that minimizes processor idle time. A matrix, e.g., Figure 5 (b), can be used to illustrate both space-sharing and time-sharing of the system. Each column of the matrix represents a processor and each row of the matrix represents a time slice.

A schedule is a mapping of processes to matrix elements, where element $A(s, p)$ represents the process scheduled on processor p for time slice s. A matrix with S non-empty rows indicates that S time slices are needed to schedule all N processes. We also use the notation $N(s)$ for the number of processes scheduled for time slice s.

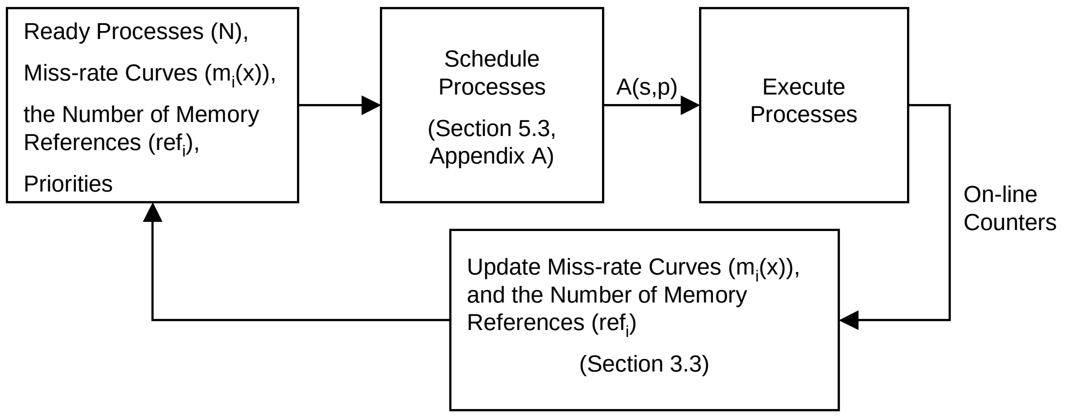

Ready Processes (N),
Miss-rate Curves (m_i(x)),
the Number of Memory References (ref_i),
Priorities
Schedule Processes
(Section 5.3, Appendix A)
A(s,p)
Execute Processes
On-line Counters
Update Miss-rate Curves (m_i(x)), and the Number of Memory References (ref_i)
(Section 3.3)

Figure 6: Overview of the Scheduler. This is only one of the schemes that can be used. Here, given a pool of ready processes, the scheduler determines S, the number of time slices in the schedule, and the mapping of processes to time slices and processors. $S \geq \lceil \frac{N}{P} \rceil$.

5.2 Scheduler Overview

Given a set of N ready processes and a P processor machine, the scheduling algorithm must determine $S \geq \lceil \frac{N}{P} \rceil$ and the mapping of processes to matrix elements. Our algorithm incorporates the analytical model of Section 4 which requires the isolated miss-rate curves for each process to determine which processes will run well together, and searches over a number of potential schedules choosing the best. After the table is complete, the S time slices can be executed. Then, the set of ready processes and the miss-rate curves are updated and the process continues. Figure 6 illustrates the procedure.

We next show how to evaluate a given schedule.

5.3 Evaluating a Given Schedule

A poor schedule has lots of idle processors. A processor is idle for a time slice if no process is assigned to it for that time slice or it is idle if it is waiting for the data to be brought into the memory due to a "miss" or page fault. Although modern superscalar processors can tolerate some cache misses, it is reasonable to assume that a processor stalls and therefore idles on every page fault.

Let the total processor idle time for a schedule be as follows:

$$\begin{aligned} \text{idle time} &= \sum_{s=1}^{S} \sum_{p=1}^{N(s)} miss(p,s) \cdot l + \sum_{s=1}^{S} (P - N(s)) \cdot T(s) \\ &= (\text{total misses}) \cdot l + \sum_{s=1}^{S} (P - N(s)) \cdot T(s) \end{aligned} \tag{9}$$

where $miss(p,s)$ is the number of misses on processor p for time slice s, l is the memory latency, $T(s)$ is the length of time slice s, and $N(s)$ is the number of processes scheduled in time slice s.

Name	Benchmark Suite	Description	Footprint (MB)
`bzip2`	SPEC CPU2000	Compression	6.2
`gcc`	SPEC CPU2000	C Programming Language Compiler	22.3
`gzip`	SPEC CPU2000	Compression	76.2
`mcf`	SPEC CPU2000	Image Combinatorial Optimization	9.9
`vortex`	SPEC CPU2000	Object-oriented Database	83.0
`vpr`	SPEC CPU2000	FPGA Circuit Placement and Routing	1.6

Table 1: The descriptions and Footprints of benchmarks used for the simulations.

In Equation 9, the first term represents the processor idle time due to page faults and the second term represents the idle time due to processors which have no process scheduled. Since the number of idle processors is given with a schedule, we can evaluate a given schedule once we know the total number of misses. The number of misses depends on memory contention amongst processes that simultaneously execute within the same time slice and the effects of time-sharing. The number of misses is determined from Equation 8 based on the information from the memory monitoring scheme ($m_i(x)$, ref_i).

Having a cost function, we now require a search algorithm to find the best schedule. The actual search algorithm used is detailed in Appendix A.

5.4 Experimental Results

This section presents the results of trace-driven simulations that demonstrate the importance of memory-aware scheduling and the effectiveness of our memory monitoring scheme with the analytical model. Although our algorithm can optimize the total processor idle time, simulations only focus on the miss-rate. In the case when there are no idle processors, i.e., processes are always scheduled on all processors in each time slice, the miss-rate can be directly converted into the processor idle time in Equation 9.

The case of scheduling six processes on the system with three processors sharing the main memory is simulated. Six processes are randomly selected from SPEC CPU2000 benchmark suite [7], and have various footprint sizes (See Table 1). Here, by footprint size we mean the memory size that a benchmark needs to achieve the minimum miss-rate. Processors are assumed to have 4-way 16-KB L1 instruction and data caches and a 8-way 256-KB L2 cache. The simulations concentrate on the main memory varying over a range of 8 MB to 256 MB with 4-KB pages.

All possible schedules are simulated. For various memory sizes, we compare the average miss-rate of all possible schedules with the miss-rates of the worst schedule, the best schedule, and the schedule by our algorithm. The simulation results are summarized in Table 2 and Figure 7. In the table, a corresponding schedule for each case is also shown except for the 128-MB and 256-MB cases where many schedules result in the same miss-rate. A schedule is represented by two sets of letters. Each set represents a time slice, and each letter represents a process: A-`bzip2`, B-`gcc`, C-`gzip`, D-`mcf`, E-`vortex`, F-`vpr`. In the figure, the miss-rates are normalized to the average miss-rate.

The results demonstrate that process scheduling can have a significant effect on the

Memory Size (MB)		Average of All Cases	Worst Case	Best Case	Algorithm
8	Miss-Rate(%)	1.379	2.506	1.019	1.019
	Schedule		(ADE,BCF)	(ACD,BEF)	(ACD,BEF)
16	Miss-Rate(%)	0.471	0.701	0.333	0.342
	Schedule		(ADE,BCF)	(ADF,BCE)	(ABD,CEF)
32	Miss-Rate(%)	0.187	0.245	0.148	0.148
	Schedule		(ADE,BCF)	(ACD,BEF)	(ACD,BEF)
64	Miss-Rate(%)	0.072	0.085	0.063	0.066
	Schedule		(ABF,CDE)	(ACD,BEF)	(ACF,BDE)
128	Miss-Rate(%)	0.037	0.052	0.029	0.029
	Schedule		(ABF,CDE)	(ACD,BEF)	(ACD,BEF)
256	Miss-Rate(%)	0.030	0.032	0.029	0.029
	Schedule		(ABF,CDE)	(ACE,BEF)	(ACD,BEF)

Table 2: The performance of the memory-aware scheduling algorithm. A schedule is represented by two sets of letters. Each set represents a time slice, and each letter represents a process: A-`bzip2`, B-`gcc`, C-`gzip`, D-`mcf`, E-`vortex`, F-`vpr`. For some cases multiple schedules result in the same miss-rate.

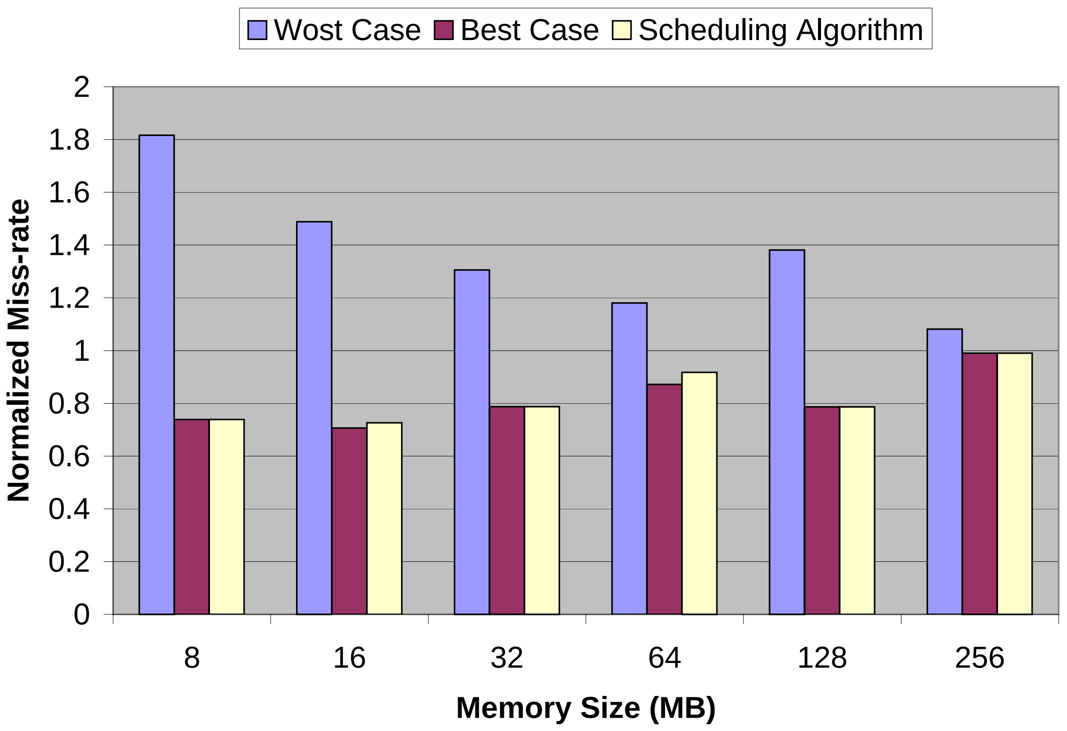


Figure 7: The comparison of miss-rates for various schedules: the worst case, the best case, and the schedule decided by the algorithm. The miss-rates are normalized to the average miss-rate of all possible schedules for each memory size.

memory performance, and thus the overall system performance. For 16-MB memory, the best case miss-rate is about 30% better than the average case, and about 53% better than the worst case. Given the very large penalty for a page fault, performance is significantly improved due to this large reduction in miss-rate. As the memory size increases, scheduling becomes less important since the entire workload fits into the memory. However, note that smart scheduling can still improve the miss-rate by about 10% over the worst case even for 256-MB memory that is larger than the total footprint size from Table 1. This happens because the LRU policy does not allocate the memory properly.

The results also illustrate that our scheduling algorithm can effectively find a near-optimal schedule. In fact, the algorithm found the optimal schedule except for the 16-MB and 64-MB cases. Even for these cases, the schedule found by the algorithm shows a miss-rate very close to the optimal case.

Finally, the results demonstrate the advantage of having a miss-rate curve for each process rather than one value of footprint size. If we schedule processes based on the footprint size, executing `gcc`, `gzip` and `vpr` together and the others in the next time slice seems to be natural since it balances the total footprint size for each time slice. However, this schedule is actully the *worst* schedule for memory smaller than 128-MB, and results in a miss-rate that is over 50% worse than the optimal schedule.

Memory traces used in this experiment have footprints smaller than 100 MB. As a result, the scheduling algorithm could not improve the miss-rate for memory which is larger than 256 MB. However, many applications have very large footprints, often larger than main memory. For these applications, the memory size where scheduling matters should scale up. We will present results for larger footprints and memory sizes in the final paper.

6 Cache Partitioning

This section shows how the cache monitoring scheme and the analytical cache model can be used to dynamically partition the cache. A partitioned cache explicitly allocates cache space to particular processes. If space is dedicated to the process, this space cannot be used to satisfy cache misses by other processes. Using trace-driven simulations, we compare partitioning with normal LRU for set-associative caches.

6.1 The Partitioning Scheme

Ths standard LRU replacement policy treats all cache blocks in the same way. For multi-tasking situations, this can often result in poor allocation of cache space among processes. When a processor and caches are time-shared, the LRU policy allocates too much to an active process even when the active process does not have enough to to reuse data. Even for cases of short time quanta, the LRU policy blindly allocates more cache space to processes that generate more misses even though other processes may benefit more from increased cache space.

We solve these problems by explicitly allocating cache space to each process. The standard LRU policy still manages cache space within a process, but not among processes. Each process gets some cache space as a dedicated area to the process. This area can only be

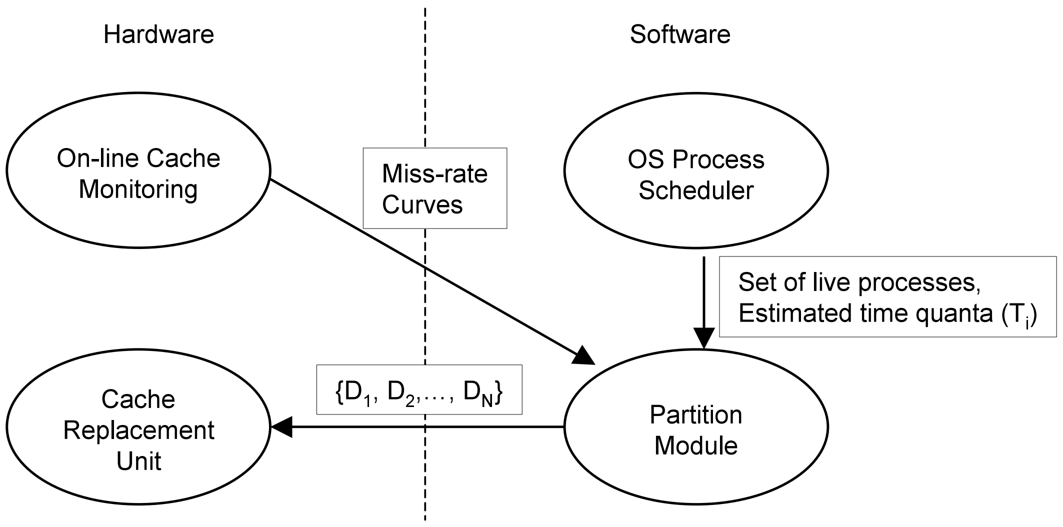


Figure 8: The implementation of on-line cache partitioning.

replaced by the process, and therefore the cache always keeps some of the process' data even when other processes are active. The rest of the cache space is a shared area that can be used by any process while it is active.

The overall flow of the partitioning scheme can be viewed as a set of four modules: on-line cache monitor, OS processor scheduler, partition module, and cache replacement unit(Figure 8). The scheduler provides the partition module with the set of executing processes and the lengths of their time slices. Then, the partition module uses this scheduling information and the miss-rate information from the on-line cache monitor to decide a cache partition. Finally, the replacement unit maps these partitions to the appropriate parts of the cache. Since the characteristics of processes change dynamically, the partition is re-evaluated after every time slice. For details on the partitioning algorithm, see Appendix B.

6.2 Experimental Results

This section presents quantitative resulting using our cache allocation scheme. The simulations concentrate on systems with very short time slices, and an 8-way set-associative L2 cache with 32-Byte blocks. We vary the size of the cache over a range of 256 KB to 4 MB. Short time slices for L2 caches represent next-generation architectures such as multi-threaded machines with fast context switch, chip multiprocessor systems, and simultaneous multithreading systems. Due to large space and long latency, our scheme is more likely to be useful for an L2 cache, and so that is the focus of our simulations. We note in passing, that we believe our approach will work on L1 caches as well.

Three different sets of benchmarks are simulated, see Table 3. The first set (Mix-1) has two threads, `art` and `mcf` both from SPEC CPU2000. The second set (Mix-2) has three threads, `vpr`, `bzip2` and `iu`. Finally, the third set (Mix-3) has four threads, two copies of `art` and two copies of `mcf`, each with a different phase of the benchmark.

The simulations compare the overall hit-rate of a standard LRU replacement policy and

Name	Thread	Benchmark Suite	Description
Mix-1	`art`	SPEC CPU2000	Image Recognition/Neural Network
	`mcf`	SPEC CPU2000	Combinatorial Optimization
Mix-2	`vpr`	SPEC CPU2000	FPGA Circuit Placement and Routing
	`bzip2`	SPEC CPU2000	Compression
	`iu`	DIS Benchmark Suite	Image Understanding
Mix-3	`art1`	SPEC CPU2000	Image Recognition/Neural Network
	`art2`		
	`mcf1`	SPEC CPU2000	Combinatorial Optimization
	`mcf2`		

Table 3: The benchmark sets simulated. All but the Image Understanding benchmark are from SPEC CPU2000 [7]. The Image Understanding is from DIS benchmark suite [13].

Cache Size (MB)	LRU L1 Hit-Rate(%)	LRU L2 Hit-Rate(%)	Partition L2 Hit-Rate(%)	Absolute Improvement(%)	Relative Improvement(%)
`art` + `mcf`					
0.2		15.6	15.3	-0.2	-1.5
0.5		17.2	16.4	-0.8	-4.6
1	71.9	26.2	36.9	10.6	40.4
2		50.0	51.1	1.1	2.2
4		76.7	75.0	-1.6	-2.2
`vpr` + `bzip2` + `iu`					
0.2		22.9	22.1	-0.8	-3.6
0.5		27.5	28.2	0.6	2.5
1	95.4	33.5	35.8	2.3	7.0
2		59.6	66.3	6.6	11.2
4		81.3	81.5	0.2	0.2
`art1` + `mcf1` + `art2` + `mcf2`					
0.2		12.0	12.6	0.6	5.3
0.5		14.2	14.3	0.1	0.7
1	71.5	16.9	19.0	2.1	12.5
2		26.6	34.9	8.2	31.0
4		50.5	51.3	0.7	1.5

Table 4: Hit-rate Comparison between the standard LRU and the partitioned LRU.

the overall hit-rate of a cache managed by our partitioning algorithm. The partition is updated every two hundred thousand memory references ($T = 200000$), and the counters are multiplied by $\delta = 0.5$ (cf. Section 3.3). Carefully selecting values of T and δ is likely to give better results. The hit-rates are averaged over fifty million memory references and shown for various cache sizes (see Table 4).

The simulation results show that the partitioning can improve the L2 cache hit-rate significantly: for cache sizes between 1 MB to 2 MB, partitioning improved the hit-rate up to 40% relative to the hit-rate from the standard LRU replacement policy. For small caches, such as 256-KB and 512-KB caches, partitioning does not seem to help. We conjecture that the size of the total workloads is too large compared to the cache size. At the other extreme, partitioning cannot improve the cache performance if the cache is large enough to hold all the workloads.

The results demonstrate that on-line cache monitoring can be very useful for cache partitioning. Although the cache monitoring scheme is very simple and has a low implementation overhead, it can significantly improve the performance for some cases.

7 Related Work

Several early investigations of the effects of context switches use analytical models. Thiébaut and Stone [16] modeled the amount of additional misses caused by context switches for set-associative caches. Agarwal, Horowitz and Hennessy [1] also included the effect of conflicts between processes in their analytical cache model and showed that inter-process conflicts are noticeable for a mid-range of cache sizes that are large enough to have a considerable number of conflicts but not large enough to hold all the working sets. However, these models work only for long enough time quanta, and require information that is hard to collect on-line.

Mogul and Borg [12] studied the effect of context switches through trace-driven simulations. Using a timesharing system simulator, their research shows that system calls, page faults, and a scheduler are the main sources of context switches. They also evaluated the effect of context switches on cycles per instruction (CPI) as well as the cache miss-rate. Depending on cache parameters, the cost of a context switch appears to be in the thousands of cycles, or tens to hundreds of microseconds in their simulations.

Stone, Turek and Wolf [14] investigated the optimal allocation of cache memory between two competing processes that minimizes the overall miss-rate of a cache. Their study focuses on the partitioning of instruction and data streams, which can be thought of as multitasking with a very short time quantum. Their model for this case shows that the optimal allocation occurs at a point where the miss-rate derivatives of the competing processes are equal. The LRU replacement policy appears to produce cache allocations very close to optimal for their examples. They also describe a new replacement policy for longer time quanta that only increases cache allocation based on time remaining in the current time quantum and the marginal reduction in miss-rate due to an increase in cache allocation. However, their policy simply assumes the probability for a evicted block to be accessed in the next time quantum as a constant, which is neither validated nor is it described how this probability is obtained.

Thiébaut, Stone and Wolf applied their partitioning work [14] to improve disk cache hit-ratios [17]. The model for tightly interleaved streams is extended to be applicable for more than two processes. They also describe the problems in applying the model in practice, such as approximating the miss-rate derivative, non-monotonic miss-rate derivatives, and updating the partition. Trace-driven simulations for 32-MB disk caches show that the partitioning improves the relative hit-ratios in the range of 1% to 2% over the LRU policy.

Analytical model for time-sharing effects in fully-associative caches was presented in [15] (cf. Section 4.1). Partitioning methods based on off-line profiling were presented. Here, we have focussed on on-line monitors to drive a partitioning scheme that better adapts to changes of behavior in processes. Further, we have extended the model to include the effects of memory contention amongst simultaneously-executing processes (Section 4.2). We have also addressed the memory interference issue for the first time in scheduling problems, and presented memory-aware scheduling algorithm.

8 Conclusion

The purpose of this paper was to optimize memory performance taking into account memory contention and time-sharing effects. These effects are quite complex and they also vary dynamically, however, we have developed a three-step methodology to solve certain scheduling and partitioning problems while optimizing memory usage and overall performance.

First, we collect hit-rate information for each process separately using simple on-line counters instrumented in caches or main memory. We showed that the information obtained from the counters can be used to construct a miss-rate versus cache size curve. Second, the isolated miss-rate versus cache size curves are fed to an analytical model which combines the running processes' miss-rates to obtain an overall miss-rate curve for the entire set of running processes. The model includes the effects of space-sharing and time-sharing in producing the overall miss-rate, which is the quantity that we wish to minimize. Therefore, as a third step, we have developed search algorithms that repeatedly compute the overall miss-rate for different sets of processes or cache sizes to determine which configuration is best. We have described the above methodology for two different problems, one being scheduling and the other being cache partitioning.

The overhead associated with our methodology is quite low. We require hardware counters in a number that grows with the associativity of hardware caches, L1 or the TLB. Other counters are implemented in software. Our model is quite easy to compute, and is computed in schedulers or partitioners within an operating system. Alternately, in multi-threaded applications, *schedulers* can be modified to incorporate the model.

Our results justify collecting additional information from on-line monitoring beyond the conventional total hit and miss counts. Our framework will apply to other problems in memory optimization, including prefetching, selection of time quanta, etc.

A Heuristic Search Algorithm for Memory-Aware Scheduling

For a small number of processes, exhaustive search can be performed to find the best schedule. As the number of processes increases, however, the number of possible schedules increases exponentially, which make exhaustive search impractical. Unfortunately, there appears to be no polynomial-time algorithm that guarantees an optimal solution. In this section, we propose a polynomial-time greedy algorithm that finds a locally-optimal schedule.

First, make an initial, optimistic guess for the number of time slices; initially $S = \lceil N/P \rceil$. Then, assign each process to a time slice one at a time in a greedy manner. Finally, increase the number of time slices until the schedule worsens.

For a fixed number of time slices, the following steps find a locally-optimal schedule. Although the algorithm cannot guarantee global optimality, in practice it often results in a schedule that is close to optimal.

1. Initialize $N(1) = N$, and $N(2) = N(3) \ \ldots \ N(S) = 0$. We assume that all N processes are in time slice 1.

2. Calculate the gain of moving process j in time slice 1 to time slice s, for $2 \le s \le S$. The gain, $g(j, s)$, is defined as the difference in the processor idle time between the current schedule and the schedule after moving process j to time slice s. This is computed using the method described in Section 5.3.

3. Move process j to time slice s, where j and s are chosen so that $g(j, s)$ is the maximum and $N(s) < P$. Increase $N(s)$ by one and decrease $N(1)$ by one. Place process j to $A(s, N(s))$.

4. Repeat steps 2 and 3 until $N(1) \le P$ and no movement has positive gain.

5. Assign processes remaining in time slice 1 to the matrix $A(1, i)$.

B Cache Partitioning Algorithm

The partition module decides the number of cache blocks that should be dedicated to a process (D_i). The D_i most recently used cache blocks of Process i are kept in the cache over other process' time slices, and Process i starts its time slice with those cache blocks in the cache. During its own time slice, Process i can use all cache blocks that are not reserved for other processes ($S = C - \sum_{j=1, j \neq i}^{N} D_j$).

The best partition is determined based on the time-shared model described in Section 4.1. The amount of data that the process has in the cache at the beginning of its time slice ($x_i(0)$)is the size of a dedicated area (D_i), and the maximum cache space that each process can use is $S = C - \sum_{j=1, j \neq i}^{N} D_j$. Therefore, the overall miss-rate can be estimated from Equation 5 for all possible partitions. The optimal partition is the set of D_i that minimizes the overall miss-rate.

In addition to LRU information, our replacement decision depends on the number of cache blocks that currently belong to each process (X_i), that is, the number of cache blocks in the cache that currently contain memory of that process. An active process (i) replaces its own LRU block if its desired allocation is smaller than its current use ($D_i + S \le X_i$). Otherwise, the LRU cache block of a dormant overallocated process is replaced. For set-associative caches, there may be no cache block of the desired process in the set. In this case, the LRU cache block of the set is replaced.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), May 1989.

[2] A. Batat and D. G. Feitelson. Gang scheduling with memory considerations. In *14th International Parallel and Distributed Processing Symposium*, 2000.

[3] Compaq. Compaq AlphaServer series. http://www.compaq.com.

[4] W. J. Dally, S. Keckler, N. Carter, A. Chang, M. Filo, and W. S. Lee. M-Machine architecture v1.0. Technical Report Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, 1994.

[5] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5), 1997.

[6] D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 1996.

[7] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.

[8] HP. HP 9000 superdome specifications. http://www.hp.com.

[9] IBM. RS/6000 enterprise server model S80. http://www.ibm.com.

[10] Intel. *Intel StrongARM SA-1100 Microprocessor*, April 1999.

[11] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15, 1997.

[12] J. C. Mogul and A. Borg. The effect of context switches on cache performance. In *the fourth international conference on Architectural support for programming languages and operating systems*, 1991.

[13] J. Munoz. *Data-Intensive Systems Benchmark Suite Analysis and Specification.* http://www.aaec.com/projectweb/dis, June 1999.

[14] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), Sept. 1992.

[15] G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Application to Cache Partitioning. In *the* 15^{th} *international conference on Supercomputing*, 2001.

[16] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), Nov. 1987.

[17] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6), June 1992.

[18] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, 1995.

[19] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R1000 performance counters. In *Supercomputing'96*, 1996.

[20] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Kool Chips Workshop in 33rd International Symposium on Microarchitecture*, 2000.