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
{suh,devadas,rudolph}@mit.edu

Abstract

We propose a low overhead, on-line memory monitoring scheme utilizing a set of novel hardware counters. The counters act like pressure gauges indicating the marginal gain in the number of hits from increasing the cache size or hit-rates as a function of cache size. Using the counters, we describe a scheme that enables an accurate estimate of the isolated miss-rates of each process as a function of cache size under the standard LRU replacement policy. This information can be used to schedule jobs or to partition the cache to minimize the overall miss-rate. The data collected by the monitors can also be used by an analytical model of cache and memory behavior to produce a more accurate overall miss-rate for the collection of processes sharing a cache in both time and space. This overall miss-rate can be used to improve scheduling and partitioning schemes.

1. Introduction

We present a low-overhead, on-line memory monitoring scheme that is more useful than the simple cache hit counters. The scheme becomes increasingly important as more and more processes and threads share various memory resources, as in the case with SMP [3, 8, 9], Multiprocessor-on-a-chip [4], and SMT [22, 11, 5] computers.

Regardless of whether a single process executes on the machine at a given point in time, or multiple processes execute simultaneously, modern systems are *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. Despite the importance of optimizing memory performance for multi-tasking situations, most published research focuses only on improving the performance of a single process.

Optimizing memory contention of multiple processes is virtually impossible without run-time information. The processes that share resources in the memory hierarchy are only known at run-time, and the memory reference characteristic of each process, 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., [23]) 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?* We may not be able to answer this question by just looking at the footprints since footprints are likely to exceed the cache size for small caches.

Consider the miss-rate curves for three different processes from SPEC CPU2000 [7] shown in Figure 1. These curves provide miss-rates as a function of cache size and give much more information than a single footprint number. This information can be very relevant in scheduling and partitioning 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.

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

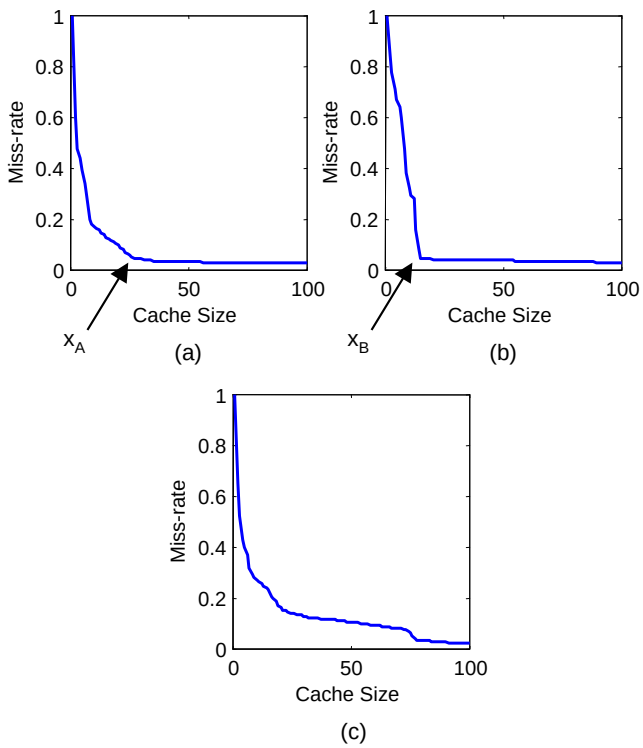

Figure 1. (a) Miss-rate curve for process A (`gcc`). (b) Miss-rate curve for process B (`swim`). (c) Miss-rate curve for process C (`bzip2`).

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.

In our scheduling and partitioning algorithms (Section 3, 4), we use marginal gains rather than miss-rate curves. The marginal gain of process i, namely $g_i(x)$, is defined as the derivative of the miss-rate curve $m_i(x)$ [2] properly weighted by the number of references for a processes (ref_i);

$$g_i(k) = (m_i(k-1) - m_i(k)) \cdot ref_i. \quad (1)$$

Therefore, we directly monitor marginal gains for each process rather than miss-rate curves. Using marginal gains, we can derive schedules and cache allocations for jobs to improve memory performance. If needed, miss-rate curves can be computed recursively from marginal gains.

We show how 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 (Section 5). The counter values alone only estimate the effects of reducing cache space for each process. 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 overall miss-rate provided by the model can drive more powerful scheduling and partitioning algorithms.

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

2. Marginal-Gain Counters

To be useful in optimizing cache performance, monitoring schemes should provide information to estimate the performance for different cache configurations or allocations. This section proposes an architectural mechanism using a set of counters to obtain the *marginal-gain* in cache hits for different sizes of the cache for a process or set of processes. Such information is useful for memory-aware scheduling and partitioning.

For fully-associative caches, the counters simply indicate the marginal gains, but for set-associative caches, the counters are mapped to marginal gains for an equivalent sized fully-associative cache. It is much easier to work with fully-associative caches and experimental results show that this works well in practice. For example, the contention between two processes sharing a fully-associative cache is a good approximation to the contention between the two processes sharing a set-associative cache.

2.1. Implementation of Counters

We want to obtain marginal gains of a process for various cache sizes without actually changing the cache configuration. In cache simulations, it has been known that different cache sizes can be simulated in a single pass [16]. We emulate this technique in hardware to obtain multiple marginal gains while executing a process with a fixed cache configuration.

In an ideal situation, where the exact LRU ordering of each cache block is known, computing the marginal gain $g(x)$ is quite simple.

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

[2] the miss-rate of process i using x cache blocks when the process is isolated without competing processes.

Counters for a Fully Associative Cache: There is one counter for each block in the cache; $counter(1)$ records the number of hits in the most recently used block, and $counter(2)$ is the number of hits in the second most recently used block, etc. When there is a reference to the ith most recently used block, then $counter(i)$ is incremented. Note that the item referenced then becomes the most recently used block, so that a subsequent reference to that item is likely to increment a different counter.

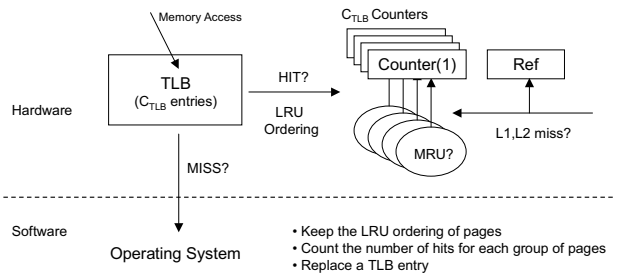

Figure 2. The implementation of memory monitors for main memory.

To compute the marginal gain curve for each process, a set of counters is maintained for each process. In a uniprocessor system, the counters are saved/restored during context switches, and when processes execute in parallel, multiple sets of counters are maintained in hardware. We thus subscript the counters with their associated process id. The marginal gain $g_i(x)$ is obtained directly by counting the number of hits in the xth *most recently used* block ($counter(x)$). The counters plus an additional one, ref_i, that records the total number of cache references for process i, are used to convert marginal gains to miss-rates for analytical models (Section 5).

2.1.1 Main Memory

Main memory can be viewed as a fully-associative cache for which on-line marginal gain counters could be useful. That is, we want to know the marginal gain to a process as a function of physical memory size. For main memory, there are two different types of accesses that must be considered: a TLB hit or a TLB miss. Collecting marginal gain information from activity associated with a TLB hit is important for processes that have small footprints and requires hardware counters in the TLB. Collecting this information when there is a TLB miss is important for processes with larger footprints and requires mostly software support.

Assuming the TLB is a fully-associative cache with LRU replacement, the hardware counters defined above can be used to compute marginal gains for the C_{TLB} most recently used pages, where C_{TLB} is the number of TLB entries, Figure 2. The 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. If the TLB is set-associative we use the technique described in the next subsection.

On a TLB miss, a memory access is serviced by either a hardware or software TLB miss handler. Ideally, we want to maintain the LRU ordering for each page and count hits per page. However, the overhead of per-page counting is too high and experimentation shows that only dozens of data points are needed for performance optimization such as scheduling and partitioning. Therefore, the entire physical memory space can be divided into a couple of dozen groups and we count the marginal gain per group. It is easy for software to maintain the LRU information. All of a process' pages in physical memory form a linked list in LRU ordering. When the page is accessed, its group counter is updated, its position on the linked list is moved to the front, and all the pages on group boundaries update their group. Machines that handle TLB misses in hardware need only insert the referenced page number into a buffer and software can do the necessary updates to the linked list on subsequent context switches. The overhead is minor requiring only several bytes for each page whose size is of the order of 4-KB, and tens of counters to compute marginal gains.

2.1.2 Set-Associative Caches

In set-associative caches, LRU ordering is kept only within each set. (We call this LRU ordering within a set as *way LRU ordering*.) Although we can only estimate marginal gains of having each *way*, not each cache block, it turns out to often be good enough for scheduling and partitioning if the cache has reasonably high associativity.

Way-Counters for Set-Associative Cache: There is one counter for each way of the cache. A hit in the cache to the MRU block of some set updates $counter(1)$. A hit in the cache to the LRU block of some set updates $counter(D)$, assuming D-way associativity. There is an additional counter, ref, recording all the hits to the cache.

It is also possible to have counters associated with each set of a cache.

> **Set-Counters for Set-Associative Cache:** There is one counter for each set of the cache. LRU information for all sets is maintained. A hit to any block within the MRU set updates $counter(1)$. A hit to any block within the LRU set updates $counter(S)$, assuming S sets in the cache. There is an additional counter, ref, recording all the hits to the cache.

To obtain the most detailed information, we can combine both *way-counters* and *set-counters*. There are $D \cdot S$ counters, one for each cache block. A hit to a block within the ith MRU set and the jth MRU way updates $counter(i, j)$.

2.2. Computing fully-associative marginal gain from set-associative counters

The marginal gain for a fully-associative cache can be approximated from the way-counters as follows:

$$counter_i(k) = \sum_{x=(k-1)\cdot S+1}^{k\cdot S} g_i(x) \qquad (2)$$

where S is the number of sets. The Figure 3 (a) illustrates the implementation of this hardware counters for 2-way associative caches.

With a minimum monitoring granularity of a *way*, high-associativity is essential for obtaining enough information for performance optimization; our experiments show that 8-way associative caches can provide enough information for partitioning. Content-addressable-memory (CAM) tags are attractive for low-power processors [24] and they have higher associativity; the SA-1100 StrongARM processor [10] contains a 32-way associative cache.

If the cache has low associativity, the information from the way LRU ordering alone is often not enough for good performance optimization. For example, consider a 2-way associative cache shown in Figure 4 (a). For cache partitioning, the algorithm would conclude that the process needs a half of the cache to achieve a low miss-rate from two given points, even though the process only needs one tenth of the cache space.

To obtain finer-grained information, we use the set LRU ordering mechanism for low-associative caches. Additional hardware keeps track of which cache set is more recently accessed. For example, the 2-way 32-KB cache used in Figure 4 has 512 cache sets.

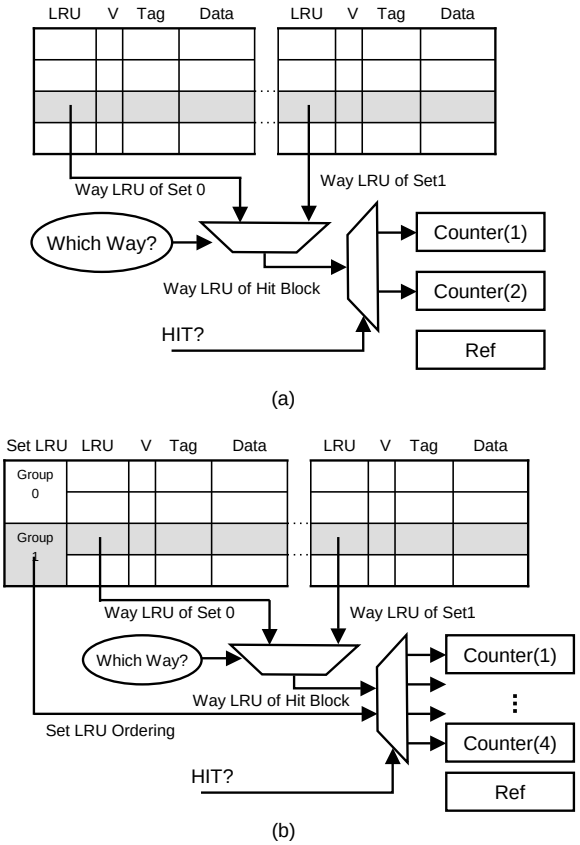

Figure 3. The implementation of memory monitors for 2-way associative caches. On a cache access, the LRU information is read for the accessed set. Then the counter is incremented based on this LRU information if the access hits on the cache. The reference counter is increased on every access. (a) The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

In practice, we do not need to maintain LRU ordering on a per cache set basis. Since there could be thousands of cache sets, the sets are divided into several groups and the LRU ordering is maintained for the groups. Figure 4 (b) demonstrates the usefulness of this set LRU information. As shown in the figure, we can obtain much more detailed information if we keep the set LRU ordering for 8 or 16 groups. The implementation of set LRU counters is shown in Figure 3 (b).

There are several strategies for converting the way-counters and set-counters into full-associative marginal gain information. In Figure 4 (b), we used *sorting* as a conversion method. First, the number of hits is counted using counters combining both way-counters and set-counters. Then, the counters are sorted in decreasing order and as-

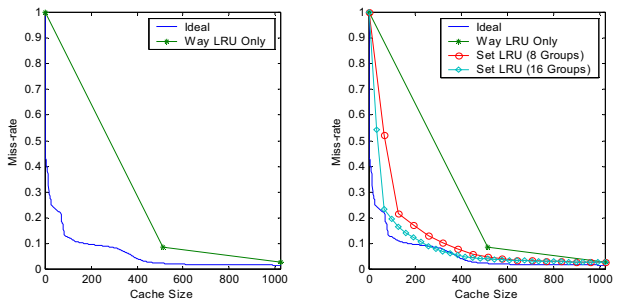

Figure 4. The estimated miss-rate curves using the set-associative cache monitor. The cache is 32-KB 2-way associative, and the benchmark is `vpr` from SPEC CPU2000. The ideal curve represents the case when you know the LRU ordering of all cache blocks. (a) Approximation only using the way LRU information.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

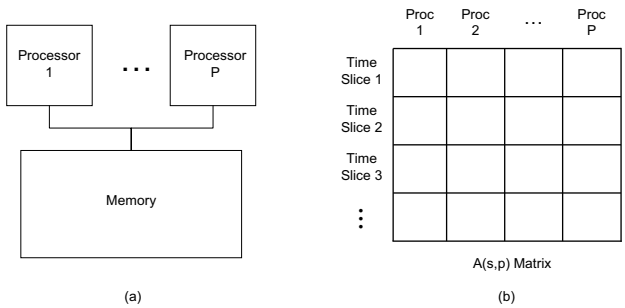

Figure 5. (a) A shared memory multiprocessor system with P processors. (b) Space-sharing and Time-sharing in multiprocessor system.

signed to marginal gains $g_i(x)$. $g_i(1)$ gets the maximum counter value and $g_i(C)$ gets the minimum counter value, where C is the number of blocks in the cache. This conversion is based on the assumption that the marginal gain is monotonically decreasing function of cache size. We are also investigating other conversion methods; column-major conversion, binomial probability conversion, etc.

Since characteristics of processes change dynamically, the estimation of $g_i(x)$ 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.

3. Memory-Aware Scheduling

When a scheduler has the freedom to select which processes execute in parallel, knowing the memory requirements of each process can help produce a better schedule. In particular, this section demonstrates how the marginal gain counters can be used to produce a memory-aware schedule. First, we begin with the problem definition and assumptions. Then, a scheduling algorithm based on marginal gains of each process is briefly explained. Finally, we validate our approach by simulations for main memory.

3.1. Scheduling Problem

We consider a system where P identical processors share the memory and N processes are ready to execute, see Figure 5 (a). The system can be a shared-memory multiprocessor system where multiple processors share the main memory, or it can be a chip multiprocessor system where processors on a single chip share the L2 cache.

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. We will assume processes are single-threaded, and 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. There may or may not be constraints in scheduling the ready processes. Constraints will merely affect the search for feasible schedules.

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, see Figure 5 (b). A matrix with S non-empty rows indicates that S time slices are needed to schedule all N processes. In our problem, $S = \lceil \frac{N}{P} \rceil$.

Our problem is to find the optimal scheduling that minimizes processor idle time due to memory misses. The number of memory misses depends on both contention amongst processes in the same time slice and contention amongst different time slices. In this section, we only consider the contention within the time slice. Considering contention amongst time slices is briefly discussed in Section 5. For a more general memory-aware scheduling strategy, see [19].

3.2. Scheduling Algorithm

For many applications, the miss rate curve as a function of memory size has a knee (See Figure 1). That is, the miss

rate quickly drops and then levels off. To minimize the number of misses, we want to schedule processes so that each process can use more cache space than the ordinate of its knee.

The relative footprint for process i is defined as the number of memory blocks allocated to the process when the memory with $C \cdot S$ blocks is partitioned among all processes such that the marginal gain for all processes is the same. C represents the number of blocks in the memory, and $C \cdot S$ represents the amount of available memory in S time slices. Effectively, the relative footprint of a process represents the optimal amount of memory space for that process when all processes execute simultaneously sharing the total memory resource over S time slices [3]. Intuitively, relative footprints corresponds to a knee of the miss-rate curve for a process.

We use a simple $C \cdot S$ step greedy algorithm to compute relative footprints. First, no memory block is allocated to any process. Then, for each block, we allocate the block to the process that obtains the maximum marginal gain for an additional block. After allocating all $C \cdot S$ blocks to processes, the allocation for each process is the relative footprint of the process. We limit the number of blocks assigned to each process to be less than or equal to C.

Once the relative footprints are computed, assigning processes to time slices is straightforward. In a greedy manner, the unscheduled process with the largest relative footprint is assigned to a time slice with the smallest total relative footprint at the time. We limit the number of processes for each time slice to be P.

3.3. Experimental Results

A trace-driven simulator demonstrates the importance of memory-aware scheduling and the effectiveness of our memory monitoring scheme. Consider scheduling six processes, randomly selected from SPEC CPU2000 benchmark suite [7] on the system with three processors sharing the main memory. The benchmark processes have various footprint sizes (See Table 1), that is, the memory size that a benchmark requires 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 6. 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 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. How-

[3] Stone, Turek, and Wolf [15] proved the algorithm results in the optimal partition assuming that marginal gains monotonically decrease as allocated memory increases.

Name	Description	FP (MB)
`bzip2`	Compression	6.2
`gcc`	C Compiler	22.3
`gzip`	Compression	76.2
`mcf`	Combinatorial Optimization	9.9
`vortex`	Object-oriented Database	83.0
`vpr`	FPGA Placement and Routing	1.6

Table 1. The descriptions and Footprints of benchmarks used for the simulations. All benchmarks are from SPEC CPU2000 [7].

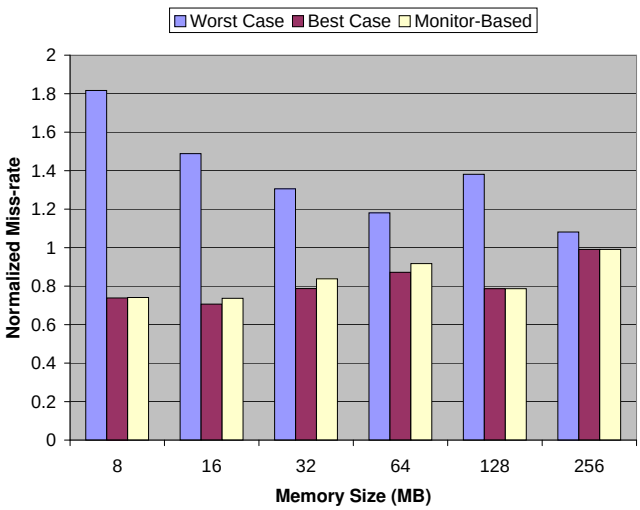

Figure 6. The comparison of miss-rates for various schedules: the worst case, the best case, and the schedule decided by the algorithm. The miss-rates are normalized to the average miss-rate of all possible schedules for each memory size.

Memory Size (MB)		Average of All Cases	Worst Case	Best Case	Algorithm
8	Miss-Rate(%)	1.379	2.506	1.019	1.022
	Schedule		(ADE,BCF)	(ACD,BEF)	(ACE,BDF)
16	Miss-Rate(%)	0.471	0.701	0.333	0.347
	Schedule		(ADE,BCF)	(ADF,BCE)	(ACD,BEF)
32	Miss-Rate(%)	0.187	0.245	0.148	0.157
	Schedule		(ADE,BCF)	(ACD,BEF)	(ABD,CEF)
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)	(ACD,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.

ever, 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 good schedule, which results in a low miss-rate. In fact, the algorithm found the optimal schedule when the memory is larger than 64-MB. Even for small memory, the schedule found by the algorithm shows a miss-rate very close to the optimal case.

Finally, the results demonstrate the advantage of having marginal gain information 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 actually 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.

4. Cache Partitioning

Just like knowing memory requirements can help a scheduler, it can also be used to decide the best way to dynamically partition the cache among simultaneous processes. A partitioned cache explicitly allocates cache space to particular processes. In a partitioned cache, if space is allocated to one process, it cannot be used to satisfy cache misses by other processes. Using trace-driven simulations, we compare partitioning with normal LRU for set-associative caches.

4.1. The Partitioning Scheme

The 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 multiple processes run simultaneously and share the cache as in simultaneous multithreading and chip multiprocessor systems, 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 this problem 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 a certain amount of cache space allocated explicitly. Then, the replacement unit decides which block within a set will be evicted based on how many blocks a process has in the cache and how many blocks are allocated to the process.

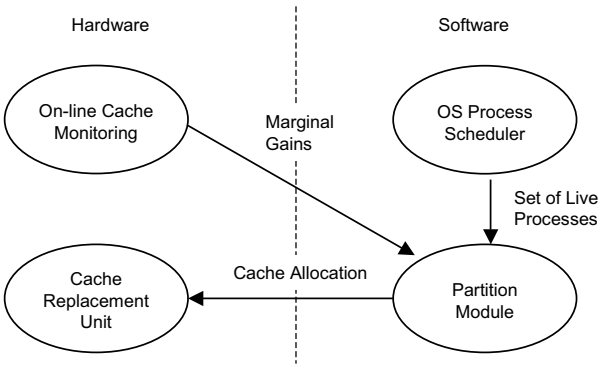

Figure 7. The implementation of on-line cache partitioning.

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 7). The scheduler provides the partition module with the set of executing processes that shares the cache at the same time. Then, the partition module uses this scheduling information and the marginal gain information from the on-line cache monitor to decide a cache partition; the module uses a greedy algorithm to allocate each cache block to a process that obtains the maximum marginal gain by having one additional block. 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 [18].

4.2. Experimental Results

This section presents quantitative results using our cache allocation scheme. The simulations concentrate on chip multiprocessor systems where processors (either 2 or 4) share the same L2 cache. The shared L2 cache is 8-way set-associative, whose size varies over a range of 256 KB to 4 MB. Each processor is assumed to have its own L1 instruction and data caches, which are 4-way 16 KB. Due to large space and long latency to main memory, 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 if L1 caches are also shared.

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

Name	Process	Description
Mix-1	`art`	Image Recognition/Neural Network
	`mcf`	Combinatorial Optimization
Mix-2	`vpr`	FPGA Circuit Placement and Routing
	`bzip2`	Compression
	`iu`	Image Understanding
Mix-3	`art1`	Image Recognition/Neural Network
	`art2`	
	`mcf1`	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].

Size (MB)	L1 %Miss	L2 %Miss	Part. L2 %Miss	Abs. %Imprv.	Rel. %Imprv.
`art` + `mcf`					
0.2		84.4	84.7	-0.3	-0.4
0.5		82.8	83.6	-0.8	-0.9
1	28.1	73.8	63.1	10.7	14.5
2		50.0	48.9	1.1	2.2
4		23.3	25.0	-1.7	-7.3
`vpr` + `bzip2` + `iu`					
0.2		73.1	77.9	-0.8	-1.1
0.5		72.5	71.8	0.7	1.0
1	4.6	66.5	64.2	2.3	3.5
2		40.4	33.7	6.7	16.6
4		18.7	18.5	0.2	1.1
`art1` + `mcf1` + `art2` + `mcf2`					
0.2		88.0	87.4	0.6	0.7
0.5		85.8	85.7	0.1	0.1
1	28.5	83.1	81.0	2.1	2.5
2		73.4	65.1	8.3	11.3
4		49.5	48.7	0.8	1.6

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

The simulations compare the overall L2 miss-rate of a standard LRU replacement policy and the overall L2 miss-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 2.2). 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 miss-rate significantly: for cache sizes between 1 MB to 2 MB, partitioning improved the miss-rate up to 14% relative to the miss-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.

5. Analytical Models

Although the straightforward use of the marginal gain counters can improve performance, it is important to know its limitation. This section discusses analytical methods that can model the effects of memory contention amongst simultaneously-running processes, as well as the effects of time-sharing, using the information from the memory monitoring scheme. The model estimates the overall miss-rate when multiple processes execute simultaneously and concurrently. Estimating an overall miss-rate gives a better evaluation of a schedule or partition. First, a uniprocessor 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. Finally, a few examples of using the model with the monitoring scheme are shown.

5.1. Model for Time-Sharing

The time-sharing model from elsewhere [17] estimates the overall miss-rate for a fully-associative cache when multiple processes time-share the same cache (memory) on a uniprocessor 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.

5.2. 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). We consider the situation where all processors context switch at the same time. More general cases where each processor can context switch at a different time can be modeled in a similar manner.

To model both time-sharing and space-sharing, we apply the original model twice. First, the model is applied to processes in the same time slice and generates a miss-rate curve for a 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 for 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 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.

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 $m_{combined,s}(x)$.

5.3. Model-Based Optimization

The analytical model can estimate the effects of both time-sharing and space-sharing using the information from our memory monitors. Therefore, our monitoring scheme with the model can be used for any optimization related to multi-tasking. For example, more accurate schedulers, which consider both time-sharing and space-sharing can be developed. Using the model, we can also partition the cache among concurrent processes or choose proper time quanta

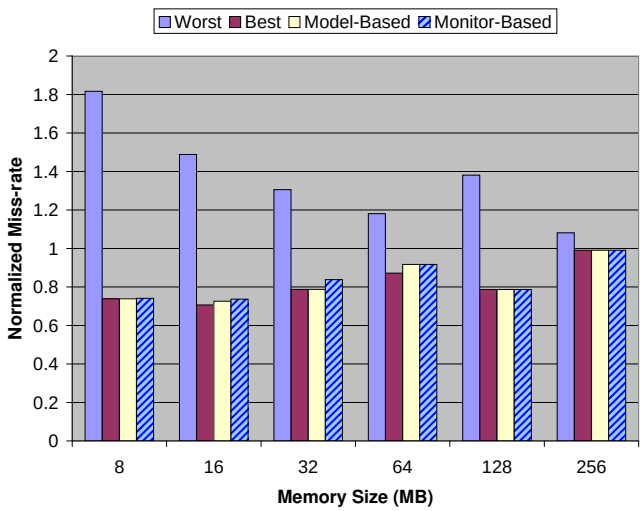

Figure 8. The comparison of miss-rates for various schedules: the worst case, the best case, the schedule based on the model, and the schedule decided by the algorithm in Section 3.

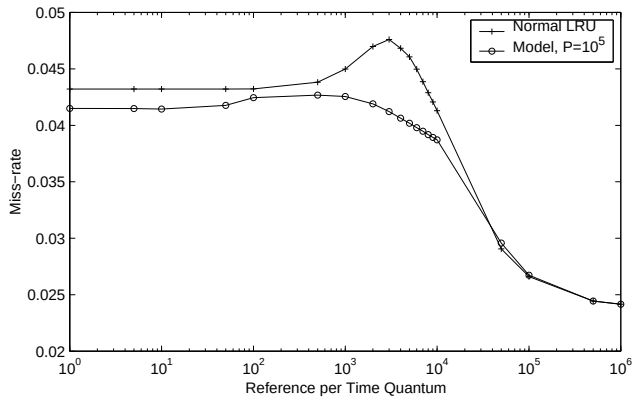

Figure 9. The results of cache partitioning among concurrent processes.

for them. In this subsection, we provide some preliminary examples of these applications.

We applied the model to the same scheduling problem solved in Section 3. In this case, however, the model evaluates each schedule based on miss-rate curves from the monitor and decides the best schedule. Figure 8 illustrates the results. Although the improvement is small, the model-based scheduler finds better schedules then the monitor-based scheme for small memories.

The model is also applied to partition the cache space among concurrent processes. Some part of the cache is dedicated to each process and the rest is shared by all. Figure 9 shows the partitioning results when 8 processes (`bzip2`, `gcc`, `swim`, `mesa`, `vortex`, `vpr`, `twolf`, `iu`) are sharing the cache (32 KB, fully associative). The partition is updated every 10^5 cache references. The figure demonstrates that time-sharing can degrade cache performance for some mid-range time quanta. Partitioning can eliminate the problem.

6. Related Work

Several early investigations of the effects of context switches use analytical models. Thiébaut and Stone [20] 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 time-sharing 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.

Snavely and Tullsen [14] proposed a scheduling algorithm considering various resource contention for simultaneous multithreading systems. Their algorithm runs samples of possible schedules to identify good schedules. While this approach is shown to be effective for a small number of jobs, random sampling is unlikely to find a good schedule for a large number of jobs. The number of possible schedules increase exponentially as the number of jobs increases. Our monitoring scheme with cache models can estimate cache miss-rates for all possible schedules without running all schedules. Therefore, our mechanism enables a scheduler to identify good schedules without a long sampling phase when the major resource contention is in caches. When there are many resources shared by processes (threads), the cache monitor can help the sampling scheme by suggesting good candidates with low cache contention.

Stone, Turek and Wolf [15] 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 [15] to improve disk cache hit-ratios [21]. 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.

An analytical model for time-sharing effects in fully-associative caches was presented in [17] (cf. Section 5.1). Partitioning methods based on off-line profiling were presented. Here, we have focused 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 5.2). We have also addressed the memory interference issue in scheduling problems, and presented a memory-aware scheduling algorithm. An earlier version of this scheduling work has presented at the Job Scheduling Workshop for Parallel Processing [19].

7. Conclusion

The effects of memory contention are quite complex and vary with time. Current cache-hit counters and other profiling tools are geared to single job performance when executed in isolation. We have developed a methodology to solve certain scheduling and partitioning problems that optimize memory usage and overall performance, for both time-shared and space-shared systems.

Marginal gain information is collected for each process separately using simple on-line hardware counters. Rather than simply counting the number of hits to the cache, we propose to count the number of hits to the most recently, second most recently, etc., items in a fully-associative cache. For set-associative caches, a similar set of counters are used to approximate the values of counters had the cache been fully associative. A small amount of hardware instrumentation enables main memory to be similarly monitored.

This information can be used to either schedule jobs or partition the cache or memory. The key insight is that by knowing the marginal gains of all the jobs, it is then possible to predict the performance of a subset of the jobs executing in parallel or the performance from certain cache partitioning schemes.

The isolated miss-rate curves for each job can be computed from the marginal gain information and the 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, we can apply search algorithms that repeatedly compute the overall miss-rate for different sets of processes or cache sizes to determine which configuration is best. In some cases, the model-based approach outperforms the monitor-based approach.

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.

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] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

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

[16] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 1995.

[17] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with application to cache partitioning. In *the* 15th *international conference on Supercomputing*, 2001.

[18] G. E. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. In *Thirteenth IASTED International Conference on Parallel and Distributed Computing System*, 2001.

[19] G. E. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In *7th International Workshop on Job Scheduling Strategies for Parallel Processing (in LNCS 2221)*, pages 116–132, 2001.

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

[21] 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.

[22] 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.

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

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