Effects of Memory Performance on Parallel Job Scheduling

G. Edward Suh, Larry Rudolph and Srinivas Devadas

MIT Laboratory for Computer Science
Cambridge, MA 02139
{suh,rudolph,devadas}@mit.edu

Abstract. We develop a new metric for job scheduling that includes the effects of memory contention amongst simultaneously-executing jobs that share a given level of memory. Rather than assuming each job or process has a fixed, static memory requirement, we consider a general scenario wherein a process' performance monotonically increases as a function of allocated memory, as defined by a miss-rate versus memory size curve. Given a schedule of jobs in a shared-memory multiprocessor (SMP), and an isolated miss-rate versus memory size curve for each job, we use an analytical memory model to estimate the overall memory miss-rate for the schedule. This, in turn, can be used to estimate overall performance. We develop a heuristic algorithm to find a good schedule of jobs on a SMP that minimizes memory contention, thereby improving memory and overall performance.

1 Introduction

High performance computing is more than just raw FLOPS; it is also about managing the memory among parallel threads so as to keep the operands flowing into the arithmetic units. Hence, some high performance job schedulers are beginning to consider the memory requirements of a job in addition to the traditional CPU requirements. But memory is spread across a hierarchy, it is difficult to know the real requirements of each job, and underallocation of space to one job can adversely affect the performance of other jobs. Allocating a fixed amount of space to a job regardless of the needs of the other concurrently executing jobs can result in suboptimal performance. We argue that a scheduler must compare the marginal utility or marginal gain accrued by a job to the gains accrued by other jobs, when giving more memory to a job.

Shared-memory multiprocessors (SMPs) [2, 8, 9], have become a basic building block for modern high performance computer systems, and in the near future, other layers of the memory hierarchy will be shared as well, with multiple processors (MPC) on a chip [3] and simultaneous multithreading (SMT) systems [13, 10, 4]. So, in nearly all high performance systems, there will be either threads, processes, or jobs that execute simultaneously and share parts of the memory system. But how many jobs should execute simultaneously? There is no magic number, rather it depends on the individual memory requirements of the jobs. Sometimes, it is even beneficial to let some processors remain idle so as to improve the overall performance.

Although most research on job scheduling for high performance parallel processing is concerned only with the allocation of processors in order to maximize processor utilization [6, 5], scheduling with memory considerations is not new. Parsons [11] studied bounds on the achievable system throughput considering memory demand of parallel jobs. Batat [1] improved gang scheduling by imposing admission control based on the memory requirement of a new job and the available memory of a system. The modified gang scheduler estimates the memory requirement for each job, and assigns a job into a time slice only if the memory is large enough for all jobs in the time slice. Although these works have pointed out the importance of considering memory in job scheduling problems, they did not provide a way of scheduling jobs to optimize the memory performance.

Rather than assuming each job or process has a fixed, static memory requirement, this paper considers a general scenario wherein a process' performance monotonically increases as a function of allocated memory. The characteristics of each process' memory usage are given by the miss-rate as a function of memory size when the process is executed in isolation (which can be easily obtained either in an on-line or off-line manner). With this information, an analytical memory model for time-shared systems [12] can be used to estimate the memory miss-rate for each job and the processor idle time for a given schedule. Therefore, our approach provides a good memory performance metric for job scheduling problems.

The new approach based on the miss-rate curves and the analytical model can be used to evaluate a schedule including the effects of memory performance. If multiple processors share the same memory, our method can effectively schedule a given set of processes to minimize memory contention. Finally, the length of time slices can be determined for time-shared systems so as to minimize pollution effects.

The paper is organized as follows. In Section 2, we present a case study of scheduling SPEC CPU2000 benchmarks, which demonstrates the importance and challenges of job scheduling with memory considerations. Section 3 motivates isolated miss-rate curves, and describes how an analytical memory model evaluates the effect of a given schedule on the memory performance. Section 4 discusses new challenges that memory considerations impose on parallel job scheduling, and suggests possible solutions using the miss-rate curves and the model. Finally, Section 5 concludes the paper.

2 Case Study: SPEC CPU2000

This section discusses the results of trace-driven simulations that estimate the miss-rate of main memory when six jobs execute on a shared-memory multiprocessor system with three processors. The results demonstrate the importance of memory-aware scheduling and the problem of naive approaches based on footprint sizes.

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

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

Six jobs, which have various footprint sizes, are selected from SPEC CPU2000 benchmark suite [7] (See Table 1). Here, footprint size represents the memory size that a benchmark needs to achieve the minimum possible miss-rate. Benchmarks in the SPEC CPU2000 suite are not parallel jobs, however, the insights obtained from the experiments are also valid for parallel processing of multi-threaded jobs since all threads (or processes) from a job can be considered as one large process from the main memory standpoint.

Concurrent execution of six jobs by three processors requires time-sharing. We assume that there are two time slices long enough to render context switching overhead negligible. In the first time slice, three out of the six jobs execute sharing the main memory and in the second time slice the three remaining jobs execute. Processors are assumed to have 4-way 16-KB L1 instruction and data caches and a 8-way 256-KB L2 cache, and 4-KB pages are assumed for the main memory.

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 and the best schedule. The miss-rate only considers accesses to main memory, not accesses that hit on either L1 or L2 caches. The simulation results are summarized in Table 2 and Figure 1. In the table, a corresponding schedule for each case is also shown. In the 128-MB and 256-MB cases, 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 job: 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 job scheduling can have significant effects 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 a very long page fault penalty, performance can be 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, the smart schedule can still improve the memory performance significantly even for the 128-MB case (over 20% better than the average case, and 40% better than the worst case).

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

Table 2. The miss-rates for various job schedules. A schedule is represented by two sets of letters. Each set represents a time slice, and each letter represents a job: A-`bzip2`, B-`gcc`, C-`gzip`, D-`mcf`, E-`vortex`, F-`vpr`.

Memory traces used in this experiment have footprints smaller than 100 MB. As a result, scheduling of simultaneously executing processes is relevant to the main memory performance only for the memory up to 256 MB. However, many parallel applications have very large footprints often larger than main memory. For these applications, the memory size where scheduling matters should scale up.

An intuitive way of scheduling with memory considerations is to use footprint sizes. Since the footprint size of each job indicates its memory space needs, one can try to balance the total footprint size for each time slice. It also seems to be reasonable to be conservative and keep the total footprint size smaller than available physical memory. The experimental results show that these naive approaches do not work.

Balancing the total footprint size for each time slice may not work for memory smaller than the entire footprint. The footprint size of each benchmark only provides the memory size that the benchmark needs to achieve the best performance, however, it does not say anything about having less memory space. For example, in our experiments, executing `gcc`, `gzip` and `vpr` together and the others in the next time slice seems to be reasonable 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.

If the replacement policy is not ideal, even being conservative and having larger physical memory than the total footprint may not be enough to guarantee the best memory performance. Smart scheduling can still improve the miss-rate by about 10% over the worst case even for 256-MB memory that is larger than

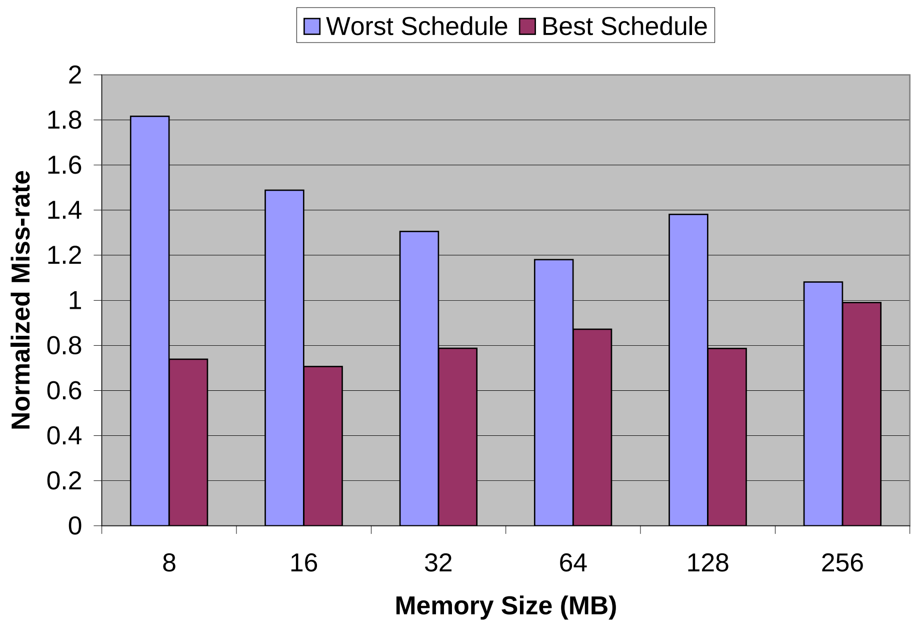

Fig. 1. The comparison of miss-rates for various schedules: the worst case, the best case, and the average of all possible schedules. The miss-rates are normalized to the average miss-rate of all possible schedules for each memory size. Notice that even when the memory is large enough to hold all the footprints of the executing jobs, the set of jobs that execute together has an effect on the miss-rate.

the total footprint size of any three jobs from Table 1. This happens because the LRU replacement policy does not allocate the memory properly. (For a certain job, the LRU policy may allocate memory larger than the footprint of the job).

3 New Approach Based on Miss-Rate Curves

The previous section pointed out that the conventional scheduling approaches based on static footprints are very limited. This section proposes a new approach based on the *isolated miss-rate curve*, $m_i(x)$. After defining the isolated miss-rate curve, an analytical model is developed that incorporates the effect of time-sharing and memory contention based on the miss-rate curves. Using these curves and the model, we show how to evaluate a given schedule.

3.1 Miss-rate Curves

The *isolated miss-rate curve* for process i, namely $m_i(x)$, is defined as the miss-rate when process i is isolated without other competing processes using the

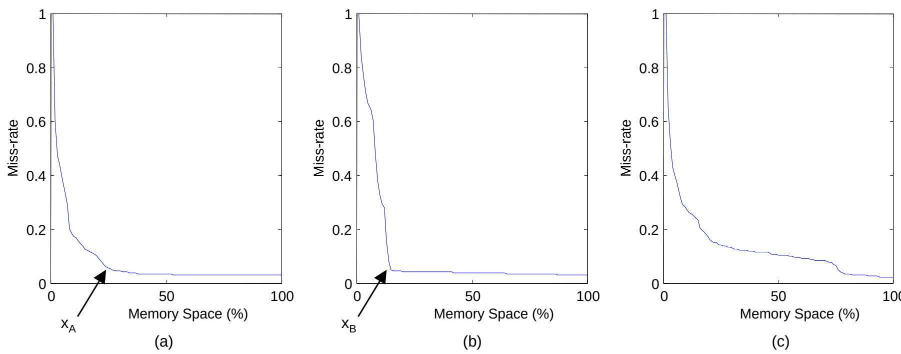

Fig. 2. (a) Miss-rate curve for process P_A (`gcc`). (b) Miss-rate curve for process P_B (`swim`). (c) Miss-rate curve for process P_C (`bzip2`). Clearly, process P_A's miss-rate does not reduce very much after the point marked x_A. Similarly, for process P_B after the point marked x_B. If $x_A + x_B$ is less than the total memory size available, then it is likely that processes P_A and P_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 P_C has a different type of miss-rate curve, and will likely not run well with either P_A or P_B.

The advantage of having a miss-rate curve rather than static footprints is clear for the problem of scheduling processes for shared-memory systems. Consider the case of scheduling three processes, whose miss-rate curves are shown in Figure 2, on a shared-memory system with two processors. *Which two processes should run together?* This question cannot be answered based on the static footprints since the memory is smaller than the individual footprints. However, from the miss-rate curves, it is clear that running both P_A and P_B simultaneously and P_C separately will result in a lower miss-rate than running P_A or P_B with P_C.

3.2 Estimating the Miss-Rate Curves

The miss-rate curves can be obtained either on-line or off-line. Here, an on-line method to estimate a miss-rate curve $m_i(x)$ is described. We use the LRU information of each page and count the number of hits in the k^{th} most recently used page for each process ($counter_i[k]$). For example, $counter_i[1]$ is the number of hits in the most recently used page of process i, and $counter_i[2]$ is the number of hits in the second most recently used page. If we count hits for one time slice, $m_j(x)$ and $counter_j[k]$ have the following relation.

$$counter_i[k] = (m_i(k-1) - m_i(k)) \cdot r_i. \tag{1}$$

where r_i is the number of memory accesses for process i over one time slice. Since $m_j(0) = 1$, we can calculate the miss-rate curve recursively.

3.3 Modeling Memory Contention

Although isolated miss-rate curves provide much more information than static footprints, the miss-rate curves alone are still not enough to predict the effects of memory contention under a non-ideal replacement policy or under the effects of time-sharing. This subsection explains how a previously developed analytical model can be extended to accurately estimate the overall miss-rate incorporating both space-sharing effects and time-sharing effects. First, the original uniprocessor model of [12] is briefly summarized. Then, we discuss how this original model can be applied to parallel jobs on shared-memory multiprocessor systems.

Uniprocessor Model The cache model from [12] 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 curve for each process ($m_i(x)$). The model assumes that the least recently used (LRU) replacement policy is used, and that there are no shared data structures among processes.

Let us consider a case when N processes execute with a given schedule (sequences of processes) and fixed time quanta for each process (T_i). First, the number of misses for each process' time quantum is estimated. Then, the overall miss-rate is obtained by combining the number of misses for each process.

Define the footprint of process i, $x_i(t)$, as the amount of process i's data in the memory at time t where time t is 0 at the beginning of the process' time quantum. Then, $x_i(t)$ is approximated by the following recursive equation, once $x_i(0)$ is known [1];

$$x_i(t+1) = MIN[x_i(t) + m_i(x_i(t)), C], \tag{2}$$

where C is the size of memory in terms of the number of blocks.

The miss-rate curve, $m_i(x)$, can be considered as the probability to miss when x valid blocks are in the memory. Therefore, the number of misses that process i experiences over one time quantum is estimated from the footprint of the process $x_i(t)$ as follows;

$$\text{miss}_i = \int_0^{T_i} m_i(x_i(t))dt. \tag{3}$$

[1] The estimation of $x_i(0)$ and more accurate $x_i(t)$ can be found in our previous work [12].

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

$$\text{miss-rate}_{\text{overall}} = \frac{\sum_{i=1}^{N} \text{miss}_i}{\sum_{i=1}^{N} T_i} \tag{4}$$

Extension to Multiprocessor Cases The original model assumes only one process executes at a time. Here, we describe how the original model can be applied to multiprocessor systems where multiple processes can execute simultaneously sharing the memory. 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 can be modeled in a similar manner.

No matter how many processes are executing simultaneously sharing the memory, all processes in a time slice can be seen as one big process from the standpoint of memory. Therefore, we take a two-step approach to model shared-memory multiprocessor cases. First, define a conceptual process for each time slice that includes memory accesses from all processes in the time slice, which we call a *combined process*. Then, the miss-rate for the combined process of each time slice is estimated using the original model. Finally, the uniprocessor model is used again to incorporate the effects of time-sharing assuming only the combined process executes for each time slice.

What should be the miss-rate curve for the combined process of a time slice? Since the original model for time-sharing needs *isolated* miss-rate curves, the miss-rate curve of 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 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 quanta. Therefore, the original model is used to obtain the combined miss-rate curves by assuming the time quantum 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 $x_{s,p}(k_{s,p})$:

$$k_{s,p} = \int_{0}^{x_{s,p}(k_{s,p})} \frac{1}{m_{s,p}(x')} dx'. \tag{5}$$

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

$$\sum_{p=1}^{P} x_{s,p}(\alpha(s,p) \cdot K_s) = x. \tag{6}$$

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 $ref_{s,p}/\sum_{i=1}^{P} ref_{s,i}$. In steady-state, the combined miss-rate curve is given by

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

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

Dealing with Shared Memory Space The model described so far assumes that there is no shared memory space among processes. However, processes from the same parallel job often communicate through shared memory space. The analytical model can be modified to be used in the case of parallel jobs synchronizing through shared memory space, as described below.

The accesses to shared memory space can be excluded from the miss-rate curve of each process, and considered as a separate process from the viewpoint of memory. For example, if P processes are simultaneously executing and share some memory space, the multiprocessor model in the previous subsection can be used considering $P + 1$ conceptual processes. The first P miss-rate curves are from the accesses of the original P processes excluding the accesses to the shared memory space, and the $(P + 1)^{th}$ miss-rate curve is from the accesses to the shared memory space. Since the $P + 1$ conceptual processes do not have shared memory space, the original model can be applied.

3.4 Evaluating a Schedule

A poor schedule has lots of idle processors, and a schedule can be better evaluated in terms of a processor idle time rather than a miss-rate. A processor is idle for a time slice if no job 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:

$$
\begin{aligned}
\text{Idle}(\%) = & \{\sum_{s=1}^{S}\sum_{p=1}^{N(s)} miss(p,s) \cdot l \\
& + \sum_{s=1}^{S}(P - N(s) \cdot T(s)\}/\{\sum_{s=1}^{S} T(s)\} \\
= & \{(\text{total misses}) \cdot l \\
& + \sum_{s=1}^{S}(P - N(s)) \cdot T(s)\}/\{\sum_{s=1}^{S} T(s)\}
\end{aligned} \tag{8}
$$

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.

In Equation 8, the first term represents the processor idle time due to page faults and the second term represents the idle time due to processors with no job scheduled on. 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, which can be estimated from the model in the previous subsection.

4 The Effects of Memory Performance on Scheduling

This section discusses new considerations that memory performance imposes on parallel job scheduling and their solutions based on the miss-rate curves and the analytical model. First, we discuss scheduling problems to optimize memory performance for the space-shared systems. Then, scheduling considerations for time-sharing the memory are studied.

4.1 Processes to Space-Share Memory

In shared-memory multiprocessor systems, processes in the same time slice space-share the memory since they access the memory simultaneously. In this case, the amount of memory space allocated to each process is determined by the other processes that are scheduled in the same time slice. Therefore, the performance (execution time) of each process can be significantly affected by which processes are scheduled to space-share the memory (see Section 2). The main consideration of memory-aware schedulers in space-shared systems is to group jobs in a time slice properly so as to minimize the performance degradation caused by the memory contention.

A schedule can be evaluated using the isolated miss-rate curves and the analytical model. Effectively, the model provides a new cost function of memory performance, and any scheduler can be modified to incorporate memory considerations by adding this new cost function from the model. As an example, here we show how a simple gang scheduler can be modified to consider the memory performance. The modification of more complicated schedulers is left for future studies.

Consider the problem of scheduling J jobs on a P_{tot} processor system, which consists of SMPs with P_{node} processors. Gang scheduling is assumed, i.e., all processes from one job are scheduled in the same time slice, and context switch at the end of the time slice. All P_{tot} processors are assumed to context switch at the same time. A processor does not context switch even on a page fault, but only when the time slice expires. The problem is to determine the number of time slices S to schedule all jobs, and assign each job to a time slice so that the processor idle time is minimized. Also, each process should be mapped to a SMP node considering memory contention.

The most obvious way of scheduling with memory consideration is to use the analytical model detailed in Section 3. If the isolated miss-rate curves are obtained either on-line or off-line, the model can easily compare different schedules. The problem is to search for the optimal schedule with the given evaluation method. For a small number of jobs, an exhaustive search can be performed to find the best schedule. As the number of jobs increases, however, the number of possible schedules increases exponentially, which makes exhaustive search impractical. Unfortunately, there appears to be no polynomial-time algorithm that guarantees an optimal solution.

A number of search algorithms can be developed to find a sub-optimal schedule in polynomial time using the analytical model directly. Alternately, we can just utilize the miss-rate curves and incorporate better memory considerations into existing schedulers. Although the analytical model is essential to accurately compare different schedules and to find the best schedule, we found that a heuristic algorithm based only on the miss-rate curves is often good enough for optimizing memory performance for space-sharing cases. The following subsection presents the heuristic search algorithm.

A Heuristic Algorithm For most applications, the miss rate curve as a function of memory size has one prominent knee (See Figure 2). That is, the miss rate quickly drops and then levels off. As a rough approximation, this knee is considered as a *relative footprint* of the process. Then, processes are scheduled to balance the total size of relative footprints for each node. Although this algorithm cannot consider the complicated effects of memory contention, it is much cheaper than computing the model and often results in a reasonable schedule.

The algorithm works in three steps; First, the *relative footprints* are determined considering the number of processes and the size of memory. At the same time, we decide the number of time slices S. Then, jobs are assigned to a time slice to balance the total relative footprints for each time slice. Finally, processes are assigned to a node to balance the relative footprints for each node.

In the explanation of the algorithm, we make use of the following notations:

- P_{tot}: the total number of processors in the entire system.
- P_{node}: the number of processors in a node.
- J: the total number of jobs to be scheduled.

- $Q(j)$: the number of processors that job j requires.
- m_j: the miss-rate curve for job j.
- r_j: the number of memory references of job j for one time slice.
- S: the number of time slices to schedule all jobs.

The relative footprint for job j, $fp(j)$ is defined as the number of memory blocks allocated to the job when the memory with $C \cdot S \cdot P/P_{node}$ blocks is partitioned among all jobs so that the marginal gain for all jobs is the same. Effectively, the relative footprint of a job represents the optimal amount of memory space for that job when all jobs execute simultaneously sharing the entire memory resource over S time slices.

To compute the relative footprints, the number of time slices S should also be decided. First, make an initial, optimistic guess; $S = \lceil \sum_{j=1}^{J} Q(j)/P \rceil$. Then, compute the relative footprints for that S and approximate the processor idle time using Equation 8 assuming that each job experiences $m_j(fp(j)) \cdot r_j$ misses over a time slice. Finally, increase the number of time slices and try again until the resultant idle time increases. For a given S, the following greedy algorithm determines the relative footprints.

1. Compute the marginal gain $g_j(x) = (m_j(x-1) - m_j(x)) \cdot r_j$. This function represents the number of additional hits for the job j, when the allocated memory blocks increases from $x-1$ to x.
2. Initialize $fp(1) = fp(2) = ... = fp(J) = 0$.
3. Assign a memory block to the job that has the maximum marginal gain. For each job, compare the marginal gain $g_j(fp(j)+1)$ and find the job that has the maximum marginal gain j_{max}. Increase the allocation for the job $fp_{j_{max}}$ by one.
4. Repeat step 3 until all memory blocks are assigned.

Once the relative footprints are computed, assigning jobs to time slices is straightforward. In a greedy manner, the unscheduled job with the largest relative footprint is assigned to a time slice with the smallest total footprint at the time. After assigning jobs to time slices, we assume that each process from job j has the relative footprint of $fp(j)/Q(j)$. Then, assign processes to nodes in the same manner.

Notice that the analytic model is not used by this algorithm. However, the model is needed to validate the heuristic. For jobs that have significantly different miss-rate curves, new heuristics are needed and the model will be required to validate those as well.

Experimental Validation The model-based algorithm and the heuristic algorithm are applied to solve a scheduling problem in Section 2. The problem is to schedule six SPEC CPU2000 benchmarks using three processors and two time slices. Figure 3 compares the miss-rates of the model-based algorithm and the heuristic algorithm with miss-rates of the best schedule and the worst schedule, which are already shown in Section 2. The best schedule and the worst schedule

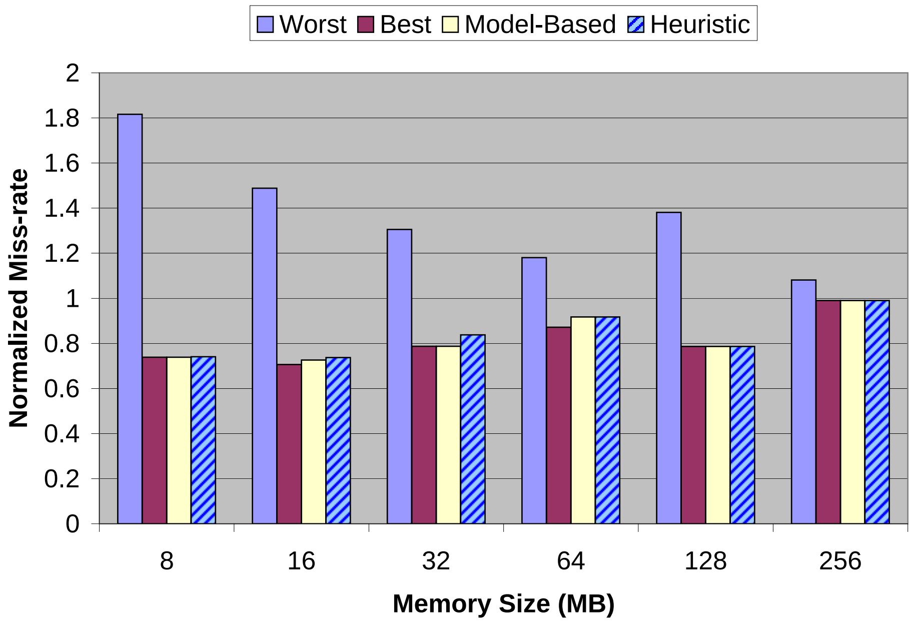

Fig. 3. The performance of the model-based scheduling algorithm and the heuristic scheduling algorithm. The miss-rates are normalized to the average miss-rate of all possible schedules for each memory size.

are found by simulating all possible schedules and comparing their miss-rates. For the model-based algorithm, the average isolated miss-rate curves over the entire execution are obtained by trace-driven simulations. Then, the schedule is found by an exhaustive search based on the analytical model. The heuristic algorithm uses the same average isolated miss-rate curves, but decides the schedule using the algorithm in the previous subsection. Once the schedules are decided by either the model-based algorithm or the heuristic algorithm, the actual miss-rates for those schedules are obtained by trace-driven simulations.

The results demonstrate that our scheduling algorithms can effectively find a good schedule. In fact, the model-based algorithm found the best schedule except for the 16-MB and 64-MB cases. Even for these cases, the model-based schedule found by the algorithm shows a miss-rate very close to the best case.

The heuristic algorithm also results in good schedules in most cases with significantly less computation than the model-based algorithm. However, the heuristic algorithm shows worse performance than the model-based algorithm because it cannot accurately estimate the effects of the LRU replacement policy.

4.2 The Length of Time Slices

When available processors are not enough to execute all jobs in parallel, processors should be time-shared amongst jobs. In conventional batch processing, each job runs to completion before giving up the processor(s). However, this approach may block short jobs from executing and significantly degrade the response time. Batch processing may also cause significant processor fragmentation. Therefore, many modern job scheduling methods such as gang scheduling use time slices shorter than the entire execution time to share processors.

Unfortunately, shorter time slices often degrade the memory performance since each job should reload the evicted data every time it restarts the execution. To amortize this context switching cost and achieve reasonable performance in time-shared systems, schedulers should ensure that time slices are long enough to reload data and reuse them. Time slices should be long to reduce the context switch overhead, but short to improve response time and processor fragmentation.

The proper length of time slices still remains as a question. Conventionally, the length of time slices are determined empirically. However, the proper length of time slices depends on the characteristics of concurrent jobs and changes as jobs and/or memory configuration vary. For example, a certain length of time slice may be long enough for jobs with a small working set, but not long enough for larger jobs. Since the proposed analytical model can predict the miss-rate for a given length of time slices, it can be used to determine the proper length once another cost function such as response time or fragmentation is given.

Figure 4 shows the overall miss-rate as a function of the length of time slices when three SPEC CPU2000 benchmarks, `gzip`, `vortex`, and `vpr`, are concurrently executing with a round-robin schedule. The solid line represents the simulation results, and the dashed line represents the miss-rates estimated by the model. The figure shows a very interesting fact that a certain range of time slices can be very problematic for memory performance. Conventional wisdom assumes that the miss-rate will monotonically decrease as the length of time slices increase. However, the miss-rate may increase for some cases since more data of processes that will run next are evicted as the length of time slices increase. The problem occurs when a time slice is long enough to pollute the memory but not long enough to compensate for the misses caused by context switches.

It is clear that time slices should always be long enough to avoid the problematic bump. Fortunately, the analytical model can estimate the miss-rate very close to the simulation results. Therefore, we can easily evaluate time slices and choose ones that are long enough.

5 Conclusion

Modern multiprocessor systems commonly share the same physical memory at some levels of memory hierarchy. Sharing memory provides fast synchronization

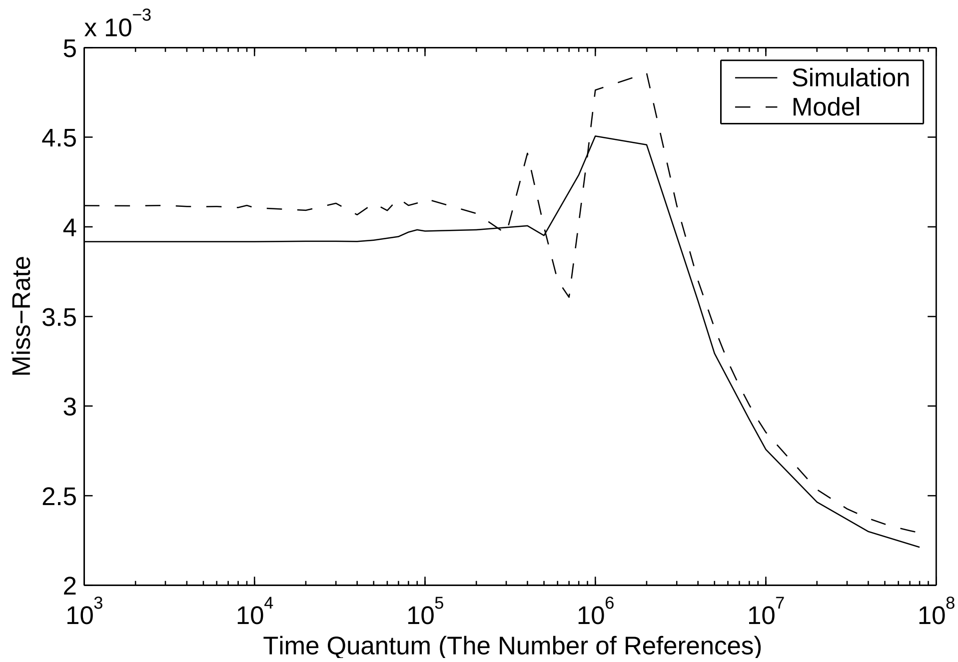

Fig. 4. The overall miss-rate when three processes (`gzip`, `vortex`, `vpr`) are sharing the memory (64 MB). The solid line represents the simulation results, and the dashed line represents the miss-rates estimated by the analytical model. The length of a time quantum is assumed to be the same for all three processes.

and communication amongst processors. Sharing memory also enables flexible management of the memory. However, it is clear that sharing memory can exacerbate the memory latency problem due to conflicts amongst processors. Currently, users of high performance computing systems prefer to "throw out the baby with the bathwater" and fore-go virtual memory and sharing of memory resources. We believe such extreme measures are not needed. Memory-aware scheduling can solve the problem.

This paper has studied the effects of the memory contention amongst processors that share the same memory on job scheduling. The case study of SPEC CPU2000 benchmarks has shown that sharing the memory can significantly degrade the performance unless the memory is large enough to hold the entire working set of all processes. Further, memory performance is heavily dependent on job scheduling. We have shown that the best schedule that minimizes memory contention cannot be found based on conventional footprints.

Miss-rate curves and an analytical model has been proposed as a new method to incorporate the effects of memory contention in job scheduling. The analytical model accurately estimates the overall miss-rate including both space-sharing effects and time-sharing effects from the miss-rate curves. Therefore, they provide a new cost function of memory performance, and any scheduler can be modified to incorporate memory considerations by adding this new cost function.

As an example, a simple gang scheduler is modified to optimize the memory performance. Applying theory to practice is not straightforward: First, some mechanism is needed to estimate the miss-rate characteristics at run-time since it is unreasonable to expect the user to provide an accurate function. Second, a heuristic algorithm is required to find a solution in polynomial time. Simulation results have validated our approach that can effectively find a good schedule that results in low miss-rates. Both a model-based algorithm and a heuristic algorithm were simulated and evaluated. Although the exhaustive search algorithm based on the model showed slightly better performance than the heuristic algorithm, the difference is minimal. Therefore, we believe that anything more than an inexpensive heuristic is overkill.

The paper is mainly focused on optimizing the performance for simultaneously executing processes. However, the approach based on the miss-rate curves and the analytical model is also applicable to scheduling problems related to time-sharing. In time-shared systems, there is a tradeoff in the length of time slices. Our model provides the metric of memory performance for this tradeoff. Especially, it is shown that a certain range of time slices can be very harmful for memory performance and this range can be avoided using the model.

The development of more realistic memory-aware schedulers is left for future studies. Practical schedulers have many considerations other than memory performance, thus it is more complicated to incorporate memory considerations into these schedulers as compared to a simple gang scheduler. However, we believe that the miss-rate curves and the analytical model provide a good metric for memory performance and existing schedulers can be modified to optimize the memory performance utilizing the given degrees of freedom.

References

1. A. Batat and D. G. Feitelson. Gang scheduling with memory considerations. In *14th International Parallel and Distributed Processing Symposium*, 2000.
2. Compaq. Compaq AlphaServer series. http://www.compaq.com/alphaserver/platforms.html.
3. 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.
4. 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.
5. D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 1996.
6. D. G. Feitelson and A. M. Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *12th International Parallel Processing Symposium*, 1998.

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/products1/unixservers/highend/superdome/specifications.html.
9. IBM. RS/6000 enterprise server model S80. http://www-1.ibm.com/servers/eserver/pseries/hardware/enterprise/s80.html.
10. 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.
11. E. W. Parsons and K. C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *the ACM SIGMETRICS conference on Measurement & modeling of computer systems*, 1996.
12. G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *15th ACM International Conference on Supercomputing*, 2001.
13. 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.