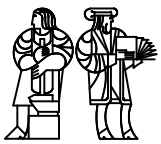
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Effects of Memory Performance on Parallel Job Scheduling

Computation Structures Group Memo 441
June 2001

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

In Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing, SIGMETRICS 2001, Cambridge, MA, USA, June 2001.

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.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Effects of Memory Performance on Parallel Job Scheduling

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

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. In otherwords, in shared-memory multiprocessors (SMPs) [2, 8, 9], which have become a basic building block for modern high performance computer systems, it is important to schedule jobs to minimize their memory contention. In the near future, some advanced microprocessors may have multiple processors (MPC) on a chip sharing a certain level of memory hierarchy [3] and others may be simultaneous multithreading (SMT) systems executing multiple threads simultaneously, and therefore effectively have multiple processors sharing all levels of memory hierarchy from L1 caches to main memory [13, 10, 4].

On current large-scale SMPs and future SMP/SMT/MPC shared memory systems, multiple high performance jobs will execute simultaneously. But how many jobs should execute simultaneously? There is no magic number, rather it depends on the individual memory requirements of the jobs. Fortunately, many high-performance applications have only coarse-grained response requirements and so the scheduler has a lot of flexibility.

Although job scheduling for high performance parallel processing has been the subject of much research, most is concerned only with the allocation of processors in order to maximize processor utilization [6, 5]. A few efforts have been made in job scheduling with memory considerations. 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. Moreover, the proper length of a time slice for gang scheduling is still a question.

Rather than assuming each job or process has a fixed, static memory requirement, this paper assumes that a process performance monotonically increases as a function of allocated memory. In particular, this paper extends an analytical memory model of time-shared systems [12] to parallel processing so as to estimate the effect of both space and time sharing on memory performance. The characteristics for each process are given by the frequency of synchronization and the miss-rate as a function of memory size when the process is executed in isolation (which can be easily obtained either on-line or off-line manner). With this information, our model accurately estimates the memory miss-rate for each job and the processor idle time for a given schedule. The model can be used for both evaluating a scheduling strategy including memory performance and developing a scheduling algorithm with memory considerations.

The rest of this paper is organized as follows. In Section 2, we discusses a case study of scheduling SPEC CPU2000 benchmarks, which demonstrate the

importance and challenges of job scheduling with memory considerations. Section 3 proposes analytical models to evaluate the effect of a given schedule on the memory performance. Based on the models and simulation results, Section 4 discusses the effect of memory considerations on parallel job scheduling. Finally, Section 5 concludes the paper.

2 Case Study: SPEC CPU2000

Simulation is a good way to understand the quantitative effects of job scheduling. This section discusses the results of trace-driven simulations that estimate the miss-rate of main memory when six jobs executing on a shared-memory multiprocessor system with three processors. The results demonstrate the importance of memory-aware scheduling and the problems of naive approaches based on footprint sizes.

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 SPEC CPU2000 suite are not parallel jobs, and each benchmark uses only one processor. Although the benchmarks are not parallel jobs, the results from these simulations can be generalized to parallel processing since multiple processes from a parallel job can be considered as one large process from the main memory standpoint.

Because there are six jobs and three processors, we assume that there are two time slices, which are long enough to ignore the context switching costs. In the first time slice, three out of the six jobs execute sharing the main memory. Then, the three remaining jobs execute in the second time slice. 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 schedule, and the best schedule. 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 worse case).

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 parellel 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. Unfortunately, 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, does not tell anything about having smaller amount of 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

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.

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`.

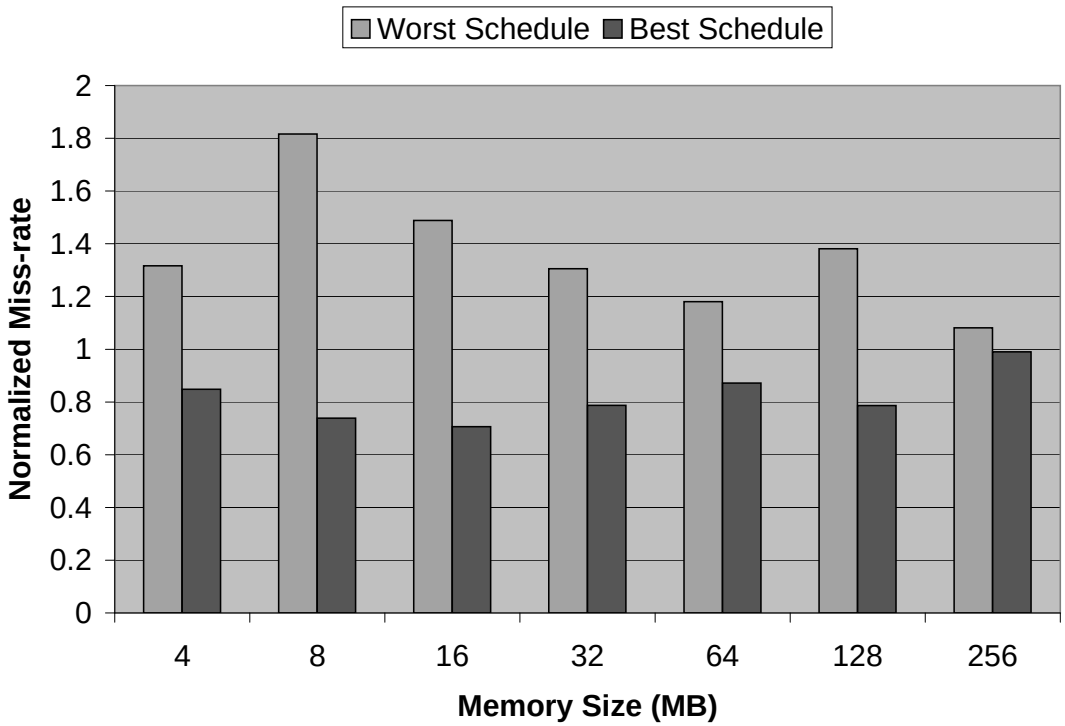


Figure 1: The comparison of miss-rates for various schedules: the worst case, the best case, and the average of all possible schedules. The miss-rates are normalized to the average miss-rate for each memory size.

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 the total footprint size of any three jobs from Table 1. This happens because the LRU policy does not allocate the memory properly.

3 Analytical Models

This section presents analytical methods that can be used to evaluate a given schedule and develop better scheduler algorithms. First, a cache model for time-shared systems is extended to parallel processing on SMPs to estimate the memory miss-rate. Then, we discuss how to estimate processor idle time from the miss-rate considering synchronization issues in parallel processing.

3.1 Estimation of Miss-rate

This subsection explains how to estimate the memory miss-rate for parallel jobs when the memory is both time-shared and space-shared. First, an uniprocessor cache model for time-shared systems de-

veloped by Suh, Devadas and Rudolph [12] is briefly summarized. Then, the model is extended to SMP cases where multiple processes space-share the memory. Finally, handling shared memory space for parallel processing is discussed.

3.1.1 Uni-Processor Model

The analytical cache model [12] 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 time slice ($R(s)$) in terms of the number of memory references, and (3) the miss-rate of each process as a function of memory size ($m_i(x)$). The model assumes that the least recently used (LRU) replacement policy is used, and there is no shared memory space among processes.

3.1.2 Extension to Multi-Processor Cases

Since the original model assumes only one process executes at a time, it should be modified to be used for shared-memory multiprocessor cases. Although the model can be extended to more general cases, consider the gang scheduling situation where all processors context switches at the same time. More complicated cases where each processor can context switch at a different time can be modeled in a similar manner. Moreover, parallel job schedulers tend to context switch all processors at once.

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, if we can obtain the combined miss-rate curve for each time slice ($m_{combined,s}(x)$) that includes memory references from all processors in the time slice, the original uni-processor model can be used for multiprocessor cases.

The input miss-rate curve ($m_i(x)$) to the original model is the miss-rate as a function of memory size assuming that only one process is executing for a long time. Therefore, the combined miss-rate curve for time slice s ($m_{combined,s}(x)$) is the miss-rate when the processes in time slice s are executing on the memory of size x for a long enough time to ignore all other time slices. This miss-rate can be obtained from the uni-processor model.

The case when multiple processes execute simultaneously and access the memory can be seen as a time-shared uni-processor with very short time slices. If we represent the miss-rate curve of the process that is assigned to processor p for time slice s as $m_{s,p}(x)$, and the number of memory reference of the process in the time slice as $r_{s,p}$. The combined miss-rate curve ($m_{combined,s}(x)$) is given by the uni-processor model assuming P processes with the time slices $r_{s,p}/\sum_{i=1}^{P} r_{s,i}$ for process p. The following paragraphs summarize this derivation of the combined miss-rate curves.

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 $x_{s,p}(k_{s,p})$:

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

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

$$\sum_{p=1}^{P} x_{s,p}(\alpha(s,p) \cdot K_s) = x \tag{2}$$

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

$$m_{combined,s}(x) = \sum_{p=1}^{P} \alpha(s,p) \cdot m_{s,p}(x_p(\alpha(s,p) \cdot K_s)). \tag{3}$$

3.1.3 Dealing with Shared Memory Space

Our analytical model 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 for parallel jobs synchronizing through shared memory space.

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 share memory space, and the $(P+1)^{th}$ miss-rate curve is from the accesses to the shared memory space.

3.2 Estimation of Processor Idle Time

A poor schedule has lots of idle processors, and a schedule can better be 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:

$$\begin{aligned}\text{idle time} &= \sum_{s=1}^{S}\sum_{p=1}^{N(s)} miss(p,s)\cdot l + \sum_{s=1}^{S}(P-N(s))\cdot T(s) \\ &= (\text{total misses})\cdot l + \sum_{s=1}^{S}(P-N(s))\cdot T(s)\end{aligned} \tag{4}$$

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 jobs scheduled in time slice s.

In Equation 4, 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.

The effects of the miss-rate on execution time also affects the behavior of barrier synchronization actions. It is possible to approximate the effects assuming the time between barrier synchronizations is known. It is then possible to approximate the variance in execution rates from the miss-rate curves.

4 The Effects of Memory Performance on Scheduling

This section discusses new considerations that memory performance imposes on parallel job scheduling. First, we discuss scheduling problems to optimize memory performance for the space-shared cases. Then, scheduling considerations for time-sharing the memory are studied.

4.1 Space Sharing

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 performance (execution time) of each process depends on the other processes that are scheduled in the same time slice with the process. In this case, the main consideration of memory-aware schedulers is to group jobs in a time slice properly so as to minimize the performance degradation caused by the memory contention.

The case study in Section 2 pointed out that conventional scheduling approaches based on footprint size are very limited because they cannot estimate either the effect of having less memory space or the effect of non-ideal replacement policy. The analytical model explained Section 3 can accurately estimate the effect of any given schedule with the LRU replacement policy. Therefore, we can evaluate any given schedule considering memory performance using the model.

Now, the problem is to search the optimal schedule with a given evaluation method. For a small num-

ber 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 near-optimal schedule in a polynomial time using the anlyatical model directly. Otherwise, we can just utilize the intuitions from the model and incorporate better memory considerations into existing schedulers. In the following subsections, we will briefly discuss two scheduling algorithms for optimizing space-shared memory performance: a greedy algorithm based on the model and an intuitive approach based on miss-rate curves.

4.1.1 A Greedy Algorithm

A greedy algorithm solves problems by making the best choice one at a time. Although this approach does not guarantee the optimal solution, it is a very simple yet results in a reasonably good solution for many cases. Here, we propose a simple greedy search algorithm to find a schedule based on the anlyatical model. The algorithm is evaluated by simulations.

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

- P: the total number of processor available in the system.
- J: the total number of jobs to be scheduled.
- $Q(j)$: the number of processors that job j requires.
- S: the number of time slices to schedule all jobs.
- $A(s)$: the number of processors available in time slice s.

The algorithm works as follows: First, make an initial, optimistic guess for the number of time slices; initially $S = \lceil \sum_{j=1}^{J} Q(J)/P \rceil$. Then, assign each job to a time slice one at a time in a greedy manner. Finally, increase the number of time slices and try again until the resultant schedule worsens.

For a fixed number of time slices, the following steps find a near-optimal schedule.

1. Initialize $A(s) = P$.
2. Calculate the gain of assigning job j in time slice s, for unassigned job j and $1 \leq s \leq S$. The gain, $g(j, s)$, is defined as the difference in the processor idle time between the current schedule and the schedule after assigning job j to time slice s. This is computed using the method described in Section 3.
3. Assign job j to time slice s, where j and s are chosen so that $g(j, s)$ is the maximum and $A(s) \geq Q(j)$. Decrease $A(s)$ by $Q(j)$.
4. Repeat steps 2 and 3 J times.

The model-based algorithm is 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. Table 3 compares the simulation results of the model-based algorithm with results of other schedules already shown in Section 2. The results demonstrate 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.

4.1.2 An Intuitive Approach

For most applications, the miss rate curve as a function of memory size has one prominent knee. That is, the miss rate quickly drops and then levels off. As a rough approximation, this knee marks the best amount of memory to allocate to the process. Less memory would result in too high of a miss-rate and more memory would be squandered and could be better used to reduce the miss rate of some other process.

4.2 Time Sharing

When available processors are not enough to execute all jobs in parallel, processors should be time-shared among jobs. In conventional batch processing, each

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)	(ACD,BEF)	(ACD,BEF)

Table 3: The performance of the model-based scheduling algorithm. 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`. For some cases multiple schedules result in the same miss-rate. Note that for 16 and 64 MB, the Algorithm yields a slightly worse schedule.

job runs to the completion before giving up the processors. However, this approach may block short jobs from executing and significantly degrade the response time. 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 not too long to improve the response time.

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 given time slices, it can be used to determine the proper length of time slices.

Figure 2 shows the overall miss-rate as a function of the length of time slices when four SPEC CPU2000 benchmarks, `gcc`, `bzip2`, `vpr` and `vortex`, are concurrently executing. The solid line represents the miss-rate estimated by the model, and the dashed line represents the simulation results. The figure shows a very interesting fact that a certain range of time slices can be very problematic for memory performance. Conventual wisdom assumes that the miss-rate to monotonically decrease as the length of time slices increase. However, the miss-rate may increase for some cases since more data of next processes 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 longer enough to avoid the problematic bump. Fortunately, the analytical model can estimate the miss-rate very close to the simulation results. Therefore, we can

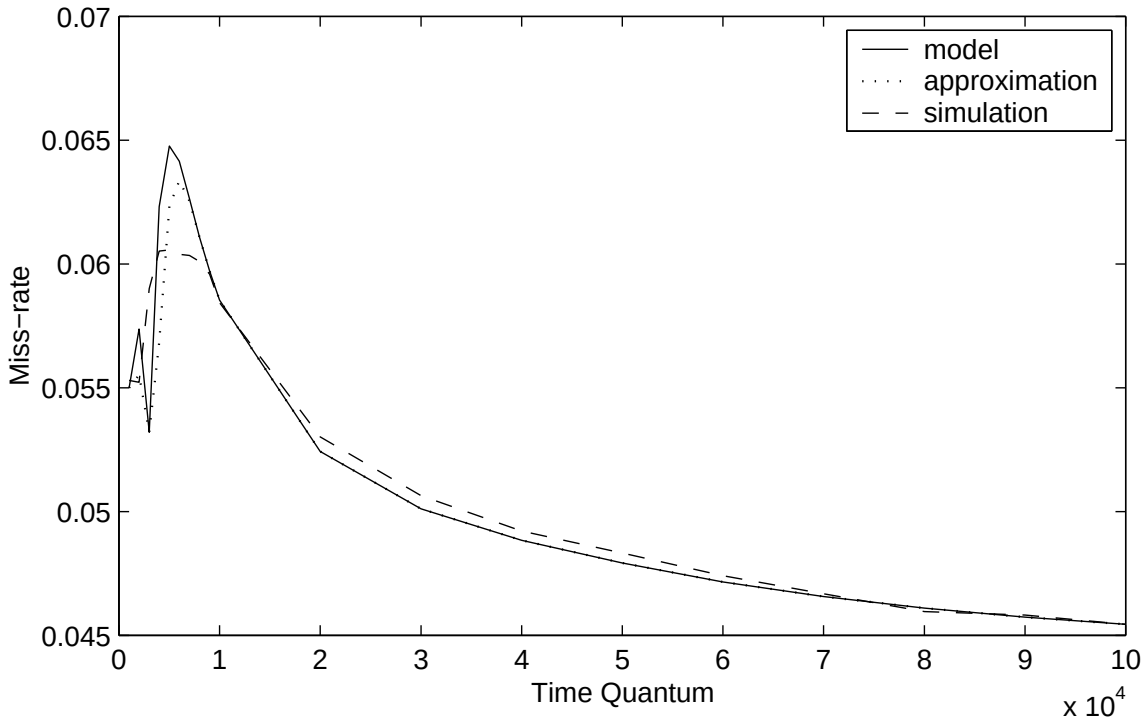


Figure 2: The overall miss-rate when four processes (`vpr`, `vortex`, `gcc`, `bzip2`) are sharing a cache (32 KB, fully-associative).

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 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 forego virtual memory and sharing of memory resources. We believe such extreme measures are not needed. Memory-aware scheduling can solve the problem.

Memory-aware scheduling has been studied as one method to reduce memory interference amongst simultaneously executing jobs. Our algorithm schedules jobs to minimize the processor idle time based on the miss-rate characteristics of each job. To apply theory to practice, some problems have been solved. First, we have proposed a mechanism to estimate the miss-rate characteristics at run-time. Second, the search algorithm has been developed to find a near-optimal solution in polynomial time. Since the miss-rate curves are approximate it is not clear that an optimal schedule using this information is indeed optimal in terms of the total number of misses. There-

fore, we believe that anything more than an inexpensive heuristic is overkill. Further investigation is required to see how inexpensive we can make the search algorithm while maintaining near-optimal results for practical situations.

Memory-aware scheduling is especially helpful when the available memory is smaller than the total footprint. However, under the LRU replacement policy, the scheduling improved the miss-rate even for the memory that is larger than the total footprint. Finally, we have found that having a miss-rate curve has an advantage over having only footprint size information.

Our scheduling algorithm has focused only on the interference amongst jobs that execute simultaneously. This is enough if a time slice is long enough. However, for shorter time slices, jobs assigned in different time slices can interfere. To have complete memory-aware scheduling, the algorithm remains to be extended to shorter time slices.

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.

[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.

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

[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.