



**Job-Speculative Prefetching: Eliminating Page Faults From Context  
Switches in Time-Shared Systems**

Computation Structures Group Memo 442  
June 2001

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

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



# Job-Speculative Prefetching: Eliminating Page Faults From Context Switches in Time-Shared Systems

## Abstract

When multiple applications have to time-share limited physical memory resources, they can incur significant performance degradation at the beginning of their respective time slices due to page faults. We propose a method to significantly improve memory system and overall performance in time-shared computers using job-speculative prefetching. While a given job or jobs are running, the operating system determines which job or jobs are going to run next, and speculatively brings in data corresponding to the next set of jobs. This significantly reduces the cold-start time for each job, ideally to zero. In fact, the goal of our work is to completely eliminate memory overhead associated with context-switching, so users can obtain all the benefits of time-sharing as well as experience the efficiency of batch processing.

Many problems have to be solved to optimize job-speculative prefetching. We have to determine the job that will run next, predict the pages that the job will access, and predict which pages currently resident in physical memory will not be accessed and can be replaced. Finally, we have to decide when, during the time slice of the currently executing job, to begin job-speculative prefetching. We provide solutions to all these problems using theoretical results and an analytical model of cache and memory behavior when multiple time-shared processes share the cache or memory.

We have run preliminary experiments using model-based job-speculative prefetching and we show that memory system performance can be significantly improved using our techniques. We show that the hit-rates achieved by multiple timeshared processes correspond, in many cases, to an effective memory size twice as large as the actual memory when using job-speculative prefetching.

## 1 Introduction

Even with the increasing size of physical memory in modern computing systems, there are many applications that use up a significant fraction, if not all, of the available physical memory. Further, when multiple applications have to time-share limited physical memory resources, they can incur significant performance degradation at the beginning of their respective time slices due to page faults.

We propose a method to significantly improve memory system and overall performance in time-shared computers using job-speculative prefetching. While a given job or jobs are running, the operating system determines which job or jobs are going to run next, and speculatively brings in

data corresponding to the next set of jobs. This significantly reduces the cold-start time for each job, ideally to zero.

In fact, the goal of our work is to completely eliminate memory overhead associated with context-switching, so users can obtain all the benefits of time-sharing as well as experience the efficiency of batch processing.

Many problems have to be solved to optimize job-speculative prefetching. The job that will run next has to be determined, and this information has to be propagated to the prefetch engine in advance of job execution. The pages corresponding to the job that will be accessed first have to be determined. The existing pages in the physical memory that will be replaced by the speculatively prefetched pages have to be chosen. If any of the above decisions is made incorrectly, performance may degrade rather than improve. Finally, we have to decide when, during the time slice of the currently executing job, to begin job-speculative prefetching. Beginning too soon may result in dramatic loss of performance for the currently-executing job, if the prefetched data replaces pages that the job will access later on in the time slice.

We answer the crucial question of when, within a time slice, to begin the job-speculative prefetch in two parts. Under certain assumptions, that include ideal, i.e., optimal replacement within memory, we give a condition that tells us how long we have to wait before beginning the job-speculative prefetch to ensure no degradation of performance. This waiting period depends on whether the currently executing job fills up the entire physical memory with its pages during a time slice or only a part of it. We then derive a general, analytical model that provides information as to when to begin the job-speculative prefetch, what pages to bring in, and what pages to replace. This model assumes the standard LRU replacement policy. In the cases where LRU is equivalent to ideal replacement, the decisions made using the model are guaranteed to not degrade performance.

We have run preliminary experiments using model-based job-speculative prefetching and we show that memory system performance can be significantly improved using our techniques. In addition, the hit-rates achieved by multiple timeshared processes can be shown to correspond to an effective memory size that is significantly larger than the actual memory size.

This paper is organized as follows. We formulate the problem and describe a simple modification of scheduling methods in determining what job(s) will run next in Section 2. We present theoretical results for ideal replacement in memory that shed light on a possible solution technique in Section

3. We present an analytical model for cache behavior under context switching and an algorithm for job-speculative prefetching in Section 4. Experimental results using the model-based algorithm are presented in Section 5. Related work is discussed in Section 6, and Section 7 concludes the paper.

## 2 Problem Definition and Analysis

### 2.1 Problem Statement

We will assume a single processor system, however, the methods described are easily generalized to multiple processors, with distributed or shared memory systems. Consider a processor where  $N$  processes  $P_1, \dots, P_N$  are running. Exactly one of these processes, say  $P_i$ , is executing. A subset  $R$  of processes are ready to run, and the remaining are waiting on some event, e.g., a keyboard input or a page fault.

We will assume a multilevel memory beginning with the  $L_1$  cache and ending with disk. We will refer to these memories as  $L_j$ ,  $1 \leq j \leq D$ . Thus,  $L_D$  corresponds to disk. In the sequel, we will speak generally of the current level of memory as being  $L_k$  with  $k < D$ , and we will be prefetching from the next level, in this case  $L_{k+1}$ . The techniques we propose will work for any level,  $1 \leq k < D$ .

Consider a sequence of processes  $P_{i_1}, P_{i_2}, \dots, P_{i_t}$  that execute on the processor during some time duration. Note that processes may be repeated in the sequence. When  $P_{i_{j+1}}$  begins running, there may be data in  $L_k$  corresponding to  $P_{i_1}$  through  $P_{i_j}$ . The amount of data corresponding to these processes depends on the footprint and memory reference characteristics, and the time quantum assigned of these processes. Assume that  $P_{i_1}$  is the same as  $P_{i_{j+1}}$ . Then, it is possible that some amount of data that was accessed (and might again be accessed) by this process is still in  $L_k$ . However, this typically is not the case. Therefore, the process  $P_{i_{j+1}}$  will experience cold misses at the beginning of its time quantum, and data will have to be brought into  $L_k$  from  $L_{k+1}$  through  $L_D$ .

Our goal is to minimize performance degradation due to cold misses in the  $L_k$  memory and ideally eliminate all cold misses experienced due to context switching. We will do this by prefetching

$P_{i_{j+1}}$ 's data into  $L_k$  while  $P_{i_j}$  is executing. To do this, we need to perform the following steps.

1. Know or predict  $P_{i_{j+1}}$ .
2. Predict the data that  $P_{i_{j+1}}$  will access at the beginning of its time quantum.
3. When  $P_{i_j}$  is executing, estimate how space  $P_{i_j}$  is using in  $L_k$  as a function of time (within this time quantum).
4. Bring in the appropriate amount of  $P_{i_{j+1}}$  data (determined in Step 2) at the appropriate time during  $P_{i_j}$ 's time quantum.

We describe how we handle Step 1 in Section 2.2. In Section 3, we provide analysis that helps with the solution of problems posed by Steps 2-4. In Section 4, we present a model-based algorithm for Steps 2-4.

Note that while we have focussed on a single processor in this section, the main change when there are  $p > 1$  processors is that there will be  $p$  jobs running at any given point of time, and we have to determine the next  $p$  jobs that will run.

## 2.2 Process Schedulers

When multiple processes or jobs can run, the scheduler in the operating system typically decides which one to run, and for how long. Operating systems implement a variety of schedulers that target fairness, efficiency, response time for interactive jobs, job turnaround and throughput. Some examples of scheduling algorithms are round-robin scheduling, priority scheduling, shortest job first scheduling, and two-level scheduling [14].

If we treat the scheduler as a black box, then it is hard to guess what is going to happen next, unless there is only one job in the pool of ready jobs. However, there is no reason to treat the scheduler as a black box. The strategy we follow is to modify the scheduler such that, rather than determining the very next job  $P_i$  to run, it determines the next *two* (or more) jobs, say  $P_i$  and  $P_j$ , that will run. When the first of these jobs  $P_i$  begins execution, the prefetch engine is told to assume that the next job will be  $P_j$ . We do not require that this decision of running  $P_j$  after  $P_i$  be cast in stone, but obviously, we would like it to be case, since the prefetch engine will begin

prefetching  $P_j$ 's data. Therefore, while the scheduler is allowed to choose any pair of jobs that are to its liking, a strong hint about what the first of these jobs should be is provided. Once  $P_i$  has finished executing, the scheduler is given a hint to select  $P_j$  and some other job, in that order, in this scheduling step.

We now discuss how we can modify a scheduler to perform next-two-job scheduling. Clearly, for a round robin scheduler this is trivial. Consider a priority scheduler with priority classes. If there are runnable processes in the highest class, they are run for one time quantum in the least recently run order. If there are no processes in the highest class, the runnable processes in the next highest class are run for two time quanta, and so on. Whenever a process uses up all the time quanta allocated to it, it is moved down one class. However, when a process does not finish its assigned time quanta due to I/O interrupts it is moved to the highest class, under the assumption that it has become interactive. When the process does not finish its assigned time quanta due to a page fault, it may be allowed to stay at the same class, or be moved up one class.

To decide the next two processes to run, the scheduler would first look at the highest class that has at least one process in it, and choose it, call it  $P_i$ , to run next. Assuming that  $P_i$  would finish its assigned time quanta, it would predict which process it would choose next, call it  $P_j$ . If  $P_j$  is in the highest class, then we are guaranteed that the scheduler will choose  $P_j$  to run next. If  $P_j$  is not in the highest class, it means the highest class currently has no runnable processes. However, while  $P_i$  is running, some other process may become runnable and be moved to highest class. In which case this process rather than  $P_j$  will be run, strictly speaking. However, we can clearly quite easily modify or force the scheduler to run  $P_j$  next.

Other types of schedulers can be modified as well. For example, the LINUX scheduler [6] computes the “goodness” of all processes in the run queue – the process with the best goodness is the one with the best claim to the CPU. The scheduler walks through the task list, tracking the process with the best goodness. Instead, track the best two processes, and use the knowledge of the second best process as described above.

The question of how far into the future the prefetch engine needs information about is related to the time quantum, and the bandwidth and latency from the next level of memory to the current level. It is also dependent on the footprint of the executing process and the amount of data that is prefetched. This is not simple to analyze, and we discuss this in the next section.

### 3 Analysis for Ideal Replacement

#### 3.1 Machine model

We shall model our system as a two-level memory hierarchy, where each of the  $C$  blocks in the faster level ( $L_k$ ) can be accessed in a  $K_1$  cycles, whereas blocks in the slower level ( $L_{k+1}$ ) can be accessed in  $K_2$  cycles, and an average of  $B$  of them can be accessed per cycle (we assume  $B$  to be no larger than 1, but no smaller than  $1/K_2$ ). We shall also initially assume a single-issue, in order processor. Out of order execution can be modelled as an approximation by appropriately reducing the average latencies  $K_1$  and  $K_2$ ; similarly multiple issue processors can be approximated with single issue processor with a faster cycle (and therefore larger latencies in terms of cycles). Finally, we shall assume that  $L_k$  is *fully associative*, i.e., any block of  $L_{k+1}$  can be mapped anywhere in the  $L_k$ .

Our objective is to understand the optimal point, during the time quantum, when to start prefetching. Starting too early will bring conflicts with the active process, while starting too late (or perhaps performing no prefetching at all) will cause the next active process to begin its time quantum with less useful data in  $L_k$ , and therefore experience a much higher initial miss rate.

We shall repeatedly use the notation  $miss_{[t_1, t_2]}(s)$  to denote the total number of misses during the interval  $[t_1, t_2]$  on a  $L_k$  of size  $s$ .

Obviously, in order to be able to prefetch, there must be some “free” bandwidth that can be used for this purpose, i.e., we must have  $B \cdot (t_2 - t_1) > miss_{[t_1, t_2]}(C)$  where  $B$  is the total bandwidth in blocks per unit of time (part of this bandwidth will be consumed to service misses of the current process). We shall assume this is always the case hereafter.

#### 3.2 Ideal replacement

A statistic of interest is the *Ideal Footprint* (IF) of a process in a given time interval. Informally, it is the number of blocks which, at any point in time, have already been accessed and will be accessed again. Intuitively, the ideal footprint of a process between the beginning and the end of a time quantum corresponds to the minimum  $L_k$  space required to experience only cold misses during the time quantum if an optimal replacement policy is used.



**Definition 1** *The Ideal Footprint of a process  $P$  at time  $t$  during the time interval  $[t_0, t_f]$ ,  $IF_{[t_0, t_f]}^P(t)$  is the number of variables which have been accessed (read or written) in between time  $t_0$  and time  $t$  and will be read again between time  $t$  and time  $t_f$ .*

The IF of a stationary process can be computed analytically from the miss rate versus  $L_k$  size curves ([10]).

We shall analyze the case in which the replacement policy is optimal. We shall assume that the process which will become active after the context switch initially has no data in  $L_k$ . This assumption can be removed, though we will not treat the general case here.

In the case of ideal replacement it is easy to prove that an optimal prefetch strategy will never interfere with the active process  $P$ , i.e., will never prefetch so much that  $P$  will not hold, at any given time, at least as much space in  $L_k$  as its IF at that time (or the whole of  $L_k$  if it is smaller than the IF of  $P$ ). In fact, any further block prefetched will necessarily eject a block which  $P$  will reuse before the end of the time quantum, causing at least one additional miss, while saving no more than a single cold miss after the context switch. Let  $C$  be the size of the fast memory and  $t_{switch}$  the time of the context switch. Then the amount of data prefetched by an optimal strategy will be no greater than

$$max\_prefetch = C - \max_t (IF_{[t_0, t_f]}^P(t) + miss_{[t, t_{switch}]}(C) - B \cdot (t_{switch} - t)). \quad (1)$$

Also, prefetching *less* data than  $max\_prefetch$  will not reduce the total number of misses (as long as this data will be used in the next time quantum), since if  $P$  always holds at least as much fast memory as its IF, it will only experience cold misses, which no extra space will avoid. Therefore, if the total amount of data which will be used in the time quantum after the context switch and is currently in  $L_{k+1}$  is at least  $max\_prefetch$ , the optimal time to start prefetching is the time  $t_{max\_prefetch}$  which will allow prefetching of exactly  $max\_prefetch$  data, satisfying the equation:

$$B \cdot (t_{switch} - t_{max\_prefetch}) - miss_{[t_{max\_prefetch}, t_{switch}]}(C) = max\_prefetch \quad (2)$$

which can be solved iteratively, or directly if the miss rate is independent of time, i.e.,  $B \cdot (t_{switch} - t_{max\_prefetch}) - miss_{[t_{max\_prefetch}, t_{switch}]}(C)$  is proportional to  $(t_{switch} - t_{max\_prefetch})$ .

If the amount of data available to be prefetched ( $data$ ) is less than  $max\_prefetch$ , then an optimal time to start prefetching is any time no earlier than  $t_{max\_prefetch}$  and no later than the last instant  $t_{min\_prefetch}$  which will still allow to prefetch all  $data$ , given by the equation

$$B \cdot (t_{switch} - t_{min\_prefetch}) - miss_{[t_{min\_prefetch}, t_{switch}]}(C) = data \quad (3)$$

which has the same form as equation 2 and can be solved in the same way.

### 3.3 Premature context switches

It is important to take into account the performance impact of prefetching in those cases when the context switch occurs earlier than the time  $t_{switch}$  on which our prediction of the optimal prefetch start time  $t_{prefetch}$  is based - a common case being a page fault, which will cause the current process to yield the processor to another process before the end of its allotted time quantum. Even though  $t_{prefetch}$  is probably no longer the optimal time to begin prefetching, we would at least like any prefetching done beginning at  $t_{prefetch}$  not to cause performance degradation compared to the case of no prefetching at all. We can prove that this is true for the Ideal Replacement case:

**Theorem 1** *In the Ideal Replacement case, if the context switch occurs at time  $t_{switch}$  the total number of misses incurred by starting prefetching at time  $t_{prefetch}$  based on an estimated context switch time  $t'_{switch} > t_{switch}$  is no more than that incurred by not prefetching at all.*

**Proof 1** *In the Ideal Replacement case, no data in the cache which will be used before  $t'_{switch}$  is ejected by prefetched data - therefore no data used before  $t_{switch}$  is ejected and the process active before the context switch experiences no additional misses due to prefetching. On the other hand, additional data in the cache for the process which becomes active after the context switch cannot increase its miss rate. Prefetching in case of a premature context switch, in the Ideal Replacement case, is therefore at least as efficient as no prefetching.*

## 4 Model

Without an ideal replacement policy, it is nearly impossible to tell exactly when the last use of each memory block for a time quantum is. However, it is possible to estimate the probability for

a memory block to be accessed. In this section, we take a probabilistic approach based on an analytical memory model developed elsewhere [13] to evaluate the effect of prefetching memory blocks of the next process and determine the best time to start the prefetching. First, we will briefly summarize the memory model, which is the basis of our discussion. Then, the model will be extended to a job prefetching problem.

#### 4.1 The LRU Model Review

As briefly discussed in the theoretical analysis, the footprint of job  $i$  ( $s_i(t)$ ), the amount of data in the memory at time  $t$ , plays the central role in modeling the number of misses for each job when multiple jobs time-share the memory. In this section, we make use of subscript  $i$  to represent job  $i$ . Assuming the miss-rate of a job only depends on the memory space it has, the number of misses for job  $i$  over one time quantum (from time 0 to  $T_i$ ) can be obtained as follows:

$$\text{miss}_i = \int_0^{T_i} M_i(s_i(t))dt \quad (4)$$

where  $M_i(s)$  is the miss-rate of job  $i$  as a function of memory space. Essentially, the memory space for a job determines the probability for a memory reference to miss at a time, and the integral of the probability results in the expected number of misses over time. Note that this equation assumes that the time is measured in terms of the number of memory references. One time unit corresponds to one memory reference.

When the memory is managed by the LRU replacement policy without prefetching, the footprint of a job increases by one memory block on a miss assuming the job's blocks are the MRU part of the memory, and remains the same on a hit. With this insight, footprint  $s_i(t)$  can be approximated by

$$s_i(t) = \text{MIN}[K_i^{-1}(t + K_i(s(0))), C] \quad (5)$$

where  $K_i(s)$  is the integral of  $M_i(s)$ ,  $K_i(s) = \int_{x=0}^s M_i(x)dx$ , and  $K_i^{-1}(s)$  represents the inverse function of  $K_i(s)$ .

From the two equations above, the number of misses for job  $i$  over time  $T_i$  with the LRU replacement policy can be written as a function of the initial amount of data in the memory  $s_i(0)$ :

$$\text{miss}_{\text{LRU},i}(s_i(0)) = \int_0^{T_i} M_i(\text{MIN}[K_i^{-1}(t + K_i(s_i(0))), C])dt. \quad (6)$$

## 4.2 Tradeoffs in Prefetching

For job  $i$  that has a given miss-rate as a function of memory space curve ( $M_i(s)$ ), the number of misses for the job over a time quantum is given as a function of the initial amount of data  $s_i(0)$  (Equation 6). Therefore, it is clear how prefetching reduces the number of misses. As we prefetch more memory blocks, a job starts its time quantum with more memory blocks ( $s_i(0)$  increases), which results in less expected number of misses. The expected number of misses saved by prefetching is written by

$$\text{gain}_i(s_{\text{LRU},i}(0), s_{\text{PF},i}) = \text{miss}_{\text{LRU},i}(s_{\text{LRU},i}(0)) - \text{miss}_{\text{LRU},i}(s_{\text{LRU},i}(0) + s_{\text{PF},i}). \quad (7)$$

where  $s_{\text{LRU},i}(0)$  is the amount of job  $i$ 's data without prefetching, and  $s_{\text{PF},i}$  is the number of memory blocks that are prefetched for job  $i$ .

Prefetching can cause additional misses if it evicts the active job's memory block while loading blocks for the next job. Let us consider a case when we prefetch  $s_{\text{PF},i}$  blocks for job  $i$  while job  $j$  is executing. Assuming that we keep prefetched blocks until the context switch from job  $j$  to job  $i$ , the memory space for the current job ( $j$ ) is limited by the prefetched blocks and the current job may experience additional misses compared to the case without prefetching.

To estimate this loss, we should first know the number of blocks that are prefetched over time. If we simply ignore the accesses from the current job, the number of prefetched blocks at time  $t$  is given by

$$\text{fetched}_i(t) = \begin{cases} 0 & \text{if } t < s_{\text{PF},i}/B \\ B \cdot (t - T_j) + s_{\text{PF},i} & \text{otherwise.} \end{cases} \quad (8)$$

Since the number of job  $j$ 's blocks is limited by the number of prefetched blocks at each time, the number of misses for job  $j$  when prefetching for job  $i$  can be written by

$$\text{miss}_{\text{PF},j}(s_j(0), s_{\text{PF},i}) = \int_0^{T_j} M_j(\text{MIN}[K_j^{-1}(t + K_j(s_i(0))), C - \text{fetched}_i(t)]) dt. \quad (9)$$

Finally, the loss caused by prefetching the next job's blocks is the difference between the number of misses with prefetching and without prefetching:

$$\text{loss}_j(s_j(0), s_{\text{PF},i}) = \text{miss}_{\text{LRU},j}(s_j(0)) - \text{miss}_{\text{PF},j}(s_j(0), s_{\text{PF},i}) \quad (10)$$

### 4.3 Job-Prefetching Decision

Now, let us discuss how to decide the time to start prefetching for each time quantum based on the model. Since it is intractable to consider the effect of prefetching over all following time quanta, we make a decision only considering the effect on the current time quantum and the next time quantum where prefetching has direct impact.

The problem is to decide the time to start prefetching blocks for the next job ( $i$ ) at the beginning of a time quantum (for job  $j$ ). We assume that the miss-rate as a function of cache space curves for both job  $i$  and  $j$  ( $M_i(s)$ ,  $M_j(s)$ ) are known from previous time quanta. Also, the number of memory blocks for each job can be easily counted. Therefore,  $s_j(0)$  is given. Since the gain for job  $i$  (Equation 7) and the loss for job  $j$  (Equation 10) are functions of  $s_{LRU,i}(0)$ ,  $s_j(0)$ , and  $s_{PF,i}$ , the desired number of blocks to prefetch  $s_{PF,i}$  can be easily determined by a linear search once we know the number of job  $i$ 's memory blocks that remain in the memory at the end of the current time quantum without prefetching,  $s_{LRU,i}(0)$ .

The value of  $s_{LRU,i}(0)$  can be estimated from Equation 5. The equation tells us what is the expected number of job  $j$ 's blocks at the end of the current time quantum. With this estimation and the counted number of blocks for each job at the beginning of the current time quantum, we can compute how many job  $i$ 's blocks would be evicted during the time quantum and left at the end since we know the sequence of jobs executed before.

Finally, once we decide the number of blocks to prefetch, that can be directly converted to the time we want to start prefetching using the following equation.

$$\text{start\_time}_{i,j} = T_j - s_{PF,i}/B. \quad (11)$$

In summary, we decide the time to start prefetch as follows: First, count the number of memory blocks for each job at the beginning of a time quantum. Second, estimate the expected number of the next job's blocks left in the memory at the end of the current time quantum. Third, decide the desired number of blocks to be prefetched by a linear search based on Equation 7 and 10. Finally, convert the number of blocks to the time.

Name	Description	Input	Memory Usage (MB)
gzip	Compression	source	79.6
		graphic	114.6
		random	120.5
mcf	Image Combinatorial Optimization		18.9
vortex	Object-oriented Database	lendian1	45.8
		lendian3	47.9
vpr	FPGA Circuit Placement and Routing		33.4
gcc	C Programming Language Compiler	200	28.7

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

## 5 Experimental Results

Simulation is a good way to understand the quantitative effects of job prefetching. This section presents the results of trace-driven simulations for job-speculative prefetching from disk to main memory. The results demonstrate that time-sharing can severely degrade the memory performance when the memory cannot hold the entire working set of all concurrent jobs. Fortunately, it is shown that these cold misses can be almost completely eliminated by speculatively prefetching blocks for the next job when the prediction is perfect and memory space is enough for prefetching. Even with prediction errors and the lack of memory space, the simulation results demonstrate that job-speculative prefetching can still improve the memory performance noticeably.

Several programs from SPEC CPU2000 benchmark suite [5] have been used for the simulations (See Table 1). There are multiple instances of `gzip` and `vortex`, however the input data sets are different even for the same benchmark program. The benchmarks have various memory usage ranging from 20 MB to 120 MB. The memory usage is the cache space that each benchmark consumes when the memory is dedicated to the job. Note that this memory usage is for the case when a job executes for a long time, thereC Programming Language Compilerfore jobs usually consume less memory space over one time quantum. Memory traces are generated using SimpleScalar tool set [3] assuming that processors have 4-way 16-KB L1 instruction and data caches and a 8-way

256-KB L2 cache.

First, we simulated cases when eight jobs execute concurrently with memory ranging from 128 MB to 512MB. The eight jobs are the benchmarks shown in Table 1, one instance per benchmark: `gzip-source`, `gzip-graphic`, `gzip-random`, `mcf`, `vortex-lendian1`, `vortex-lendian3`, `vpr`, and `gcc`. Jobs are scheduled in a round-robin fashion with three million memory references per time quantum for large jobs (three instances of `gzip`) and one million references per time quantum for the others. Disk to memory bandwidth is assumed to be one page per memory access.

At the end of a time quantum, the simulator records the current job's pages in the memory with their LRU ordering. FoC Programming Language Compiler each time quantum, pages for a predicted next job are prefetched starting from the MRU page to the LRU page. To memorize pages, we need 20 bits per page assuming that a page is 4 KB. Therefore, about 160 KB is required for each job if the memory is 256 MB. However, this overhead is negligible since we only need to have this data structure for the current job and the next job in the memory no matter how many jobs are in memory.

Simulation results of eight-concurrent-job cases are summarized in Figure 1. For each memory size, we compared three different prefetching strategies: no prefetching, prefetching with perfect job prediction, and prefetching with 50% job prediction accuracy. No prefetching stands for the standard LRU policy and its approximations, which are used by most modern systems. Prefetching with perfect job prediction is an ideal case of job-speculative prefetching where you always know the next job for sure, yet this can be realistic when we have modified the scheduler as discussed in Section 2. Finally, prefetching with 50% job prediction accuracy stands for cases where the next job cannot be predicted accurately. We believe that 50% is a very pessimistic number, however, it is worthwhile to see what the benefit of job-speculative prefetching is, in this case.

First of all, the simulation results demonstrate that time-sharing can severely degrade the memory performance. If we compare the miss-rate of no-prefetching cases for 128 MB and 512 MB where there are no misses caused by context switching, the difference is significant. Even though there are millions of memory references per time quantum, which means very long time quanta, context switches cause significant amount of additional misses if the memory cannot hold the entire working set. Although memory is becoming larger, so are programs. In fact, larger programs

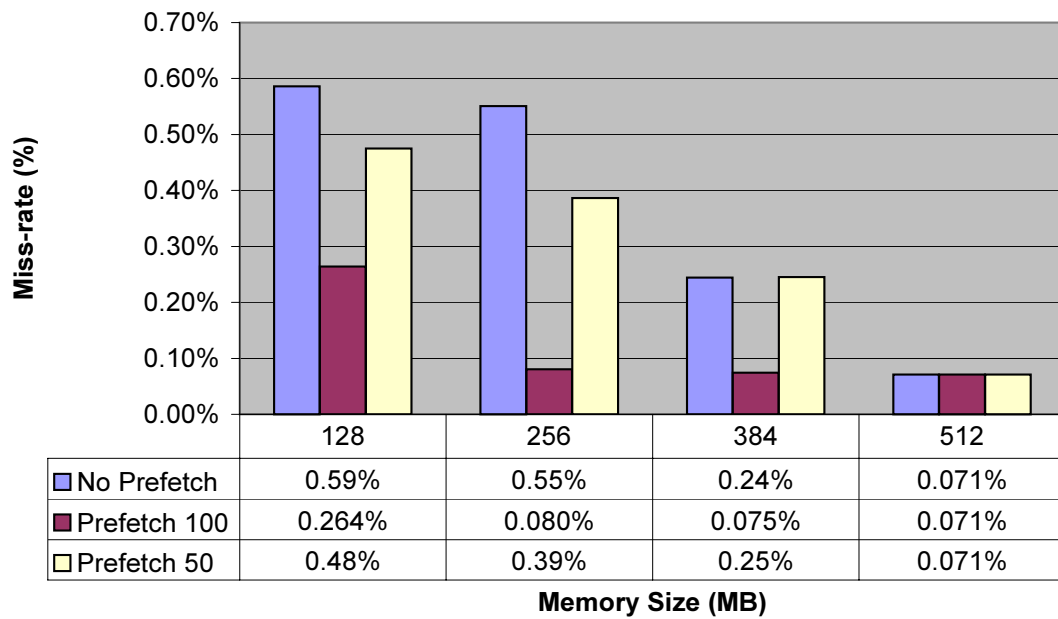


Figure 1: The effect of job-speculative prefetching on memory miss-rate: comparing miss-rates for no prefetching (No Prefetching), prefetching with perfect job prediction (Prefetching 100), and prefetching with 50% job prediction accuracy (Prefetching 50).



will cause more problems for context switching since they takes more cycles to reload a working set.

Fortunately, simulations show that job-speculative prefetching can almost completely eliminate the misses caused by context switching in some cases. For 256 MB memory, which is large enough to hold the working sets of any two jobs but not large enough to keep the entire working set, job-speculative prefetching achieves very close to the miss-rate of 512 MB memory. The miss-rate is slightly higher because prefetched blocks can be evicted before a context switch. On the other hand, if memory is too small to hold the working sets of both current job and the next job such as 128 MB in this experiment, prefetching pages for the next job can harm the performance of the current job and we cannot prefetch all blocks for the next job. However, we can still obtain noticeable improvement of the miss-rate, by timing the prefetching properly using the methods outlined in Section 4. The only case when job-speculative prefetching does not help is in the case of a large memory that can hold the entire working set of all live processes.

Job-speculative prefetching increases the effective memory size. Since we can prefetch pages that would not otherwise be kept in the memory, it is the same as having a larger memory that can actually keep those pages. In this experiment, 256 MB memory effectively becomes close to 512 MB if blocks for the next job are speculatively prefetched.

Finally, the simulation results show that job-speculative prefetching can still eliminate a significant portion of misses caused by context switches even if the next job prediction is not correct. This implies that there is very little harm for the miss-rate of the current job even though we prefetch blocks of the wrong job. This is because our algorithms are based on the conservative approach described in Sections 3 and 4. First, if there are no blocks for the next job left in the memory, prefetching can harm the performance only by evicting the current job's blocks, which we very carefully control using our analytical model. On the other hand, if blocks of the next job are left in the memory, prefetching for a wrong job can do harm by evicting blocks that are going to accessed in the next time quantum. However, the probability of this happening is very small.

Memory traces used in this experiment have memory usage smaller than 120 MB. As a result, time sharing did not matter for memory larger than 512 MB. However, there are many applications that have very large footprints sometimes even larger than main memory. Moreover, if the number of concurrent jobs increases, the size of the entire working set also increases. Therefore, for larger

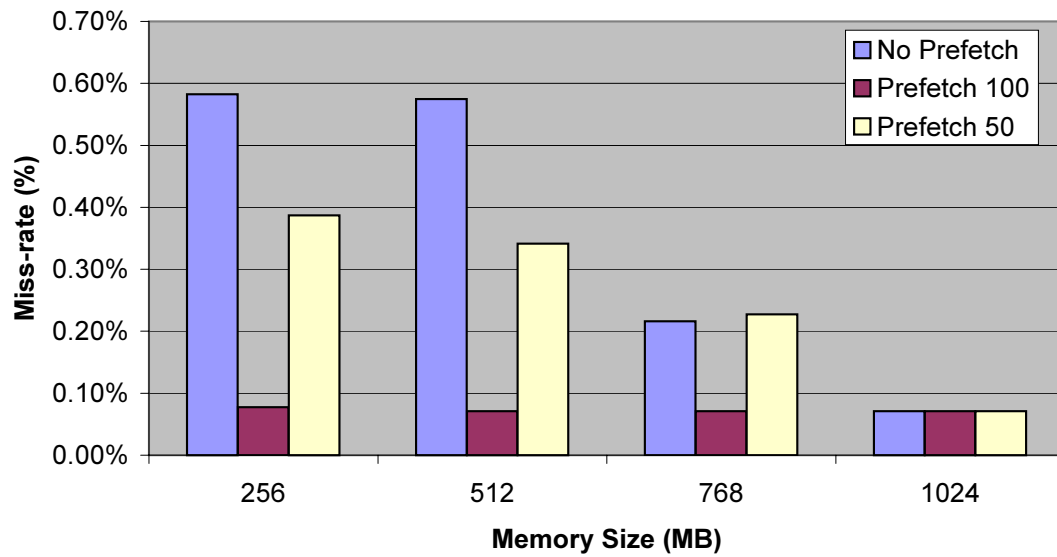


Figure 2: The effect of job-speculative prefetching on memory miss-rate when the number of jobs is doubled: comparing miss-rates for no prefetching (No Prefetching), prefetching with perfect job prediction (Prefetching 100), and prefetching with 50% job prediction accuracy (Prefetching 50).

applications or a larger number of applications, the memory size where job-speculative prefetching can help will scale up. For example, we just increased the number of jobs by duplicating the traces used before. As the result in Figure 2 indicates, job-speculative prefetching significantly improves the miss-rate even for 512 MB memory.

The key point is that job-speculative prefetching can *increase* the effective memory size, when the running processes use up a significant fraction or all of the memory. The improvement in miss-rates for memory are very significant because, for modern processors, there is a factor of 1000 in the access time for memory versus disk. For example, an improvement in miss-rate from 0.60% to 0.07% (256MB memory in Figure 2), implies a performance improvement of 4.1X.

## 6 Related Work

### 6.1 Analytical Models

Several early investigations of the effects of context switches use analytical models. Thiébaut and Stone [15] 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 [7] studied the effect of context switches through trace-driven simulations. Using a timesharing system simulator, their research shows that system calls, page faults, and a scheduler are the main sources of context switches. They also evaluated the effect of context switches on cycles per instruction (CPI) as well as the cache miss-rate. Depending on cache parameters, the cost of a context switch appears to be in the thousands of cycles, or tens to hundreds of microseconds in their simulations.

Stone, Turek and Wolf [12] 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 an 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.

Thiébaud, Stone and Wolf applied their partitioning work [12] to improve disk cache hit-ratios [16]. 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.

Our analytical model and partitioning differ from previous efforts that tend to focus on some specific cases of context switches. Our model works for any specific time quanta, whereas the previous models focus only on long time quanta. Also, our partitioning works for any time quanta, whereas Thiébaud's algorithms only works for very short time quanta. Moreover, the inputs of our model (miss-rates) are much easier to obtain compared to footprints or the number of unique cache blocks that previous models require.

## 6.2 Prefetching

Prefetching is an extensive area of research at each level in the memory hierarchy. Software and hardware prefetch techniques targeting L1 and L2 caches largely focus on improving the performance of a single job [17].

Research has also been done in giving control of memory management decisions to sophisticated applications. The Mach operating system supports external pagers to allow applications to control the backing storage of their data [11]. Prefetching and replacement decisions have been made in tandem in the context for I/O prefetching for file systems [4] [9].

Mowry, Demke and Krieger [8] describe a fully-automatic technique where a compiler provides information on future access patterns, and the operating system supports non-binding prefetch and release hints for managing I/O. The scheme described targeted improved performance for a single out-of-core application, and did not consider context switching and cold misses.

When multiple processes are running, with both out-of-core and interactive processes in the mix, the out-of-core applications can severely degrade the performance of the interactive ones. Brown and Mowry [2] show that carefully choosing the pages that are replaced by the out-of-core application can significantly improve performance. The amount of memory that the out-of-core application can use is restricted, since pages corresponding to interactive applications are not replaced.

Our work differs from the above in that we allow the currently executing application to replace any or all data in the current level of memory, but at the appropriate time, when it is clear that the current process will be swapped out before it touches certain data, we prefetch the next application's data and replace this data. Further, our techniques are not restricted to any particular level of memory.

## 7 Conclusion

We have proposed a method to significantly improve memory system and overall performance in time-shared computers using job-speculative prefetching. We have shown that job-speculative prefetching can significantly reduce cold misses that occur due to context switching.

A simple modification of an operating system scheduler to determine the next job that will run, while the current job is executing, enables job-speculative prefetching. We used an analytical model of cache/memory behavior to predict the pages that the next job will access, and predict which pages currently resident in physical memory that will not be accessed by the currently-executing job and can be replaced. We showed that memory system performance can be significantly improved using our techniques. Perhaps, more importantly, we have shown that we can increase the effective memory size as viewed by multiple timeshared processes, in the case where the sum total of the footprints of the live processes is greater than the actual available memory. Our prefetching techniques can be applied even in the case where one or more processes use up

the entire available memory when they run.

Ongoing work includes applying this method to higher-level memories such as L1 and L2 cache, as well as targeting improved performance for multiple interactive threads/processes such as those found on a Web server.

## References

- [1] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), May 1989.
- [2] A. D. Brown and T. C. Mowry. Taming the Memory Hogs: Using Compiler-Inserted Releases to Manage Physical Memory Intelligently. In *Proceedings of the 4<sup>th</sup> Symposium on Operating Systems Design and Implementation (OSDI'00)*, Oct. 2000.
- [3] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.
- [4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 188–197, Dec. 1995.
- [5] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, July 2000.
- [6] S. Maxwell. *LINUX Core Kernel Commentary*. Coriolis Open Press, 1999.
- [7] 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.
- [8] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2<sup>nd</sup> Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.
- [9] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching

- and Caching. In *Proceedings of 15<sup>th</sup> Symposium on Operating System Principles*, pages 79–95, Dec. 1995.
- [10] E. Peserico. Analysis of cache behavior in timeshared systems. Technical Report Computation Structures Group Memo, Massachusetts Institute of Technology, 2001.
- [11] R. Rashid, J. A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the 2<sup>nd</sup> ASPLOS*, Oct. 1987.
- [12] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9), Sept. 1992.
- [13] G. E. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Application to Cache Partitioning. In *the 15<sup>th</sup> international conference on Supercomputing*, 2001.
- [14] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.
- [15] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4), Nov. 1987.
- [16] 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.
- [17] S. P. Vanderwiel and D. J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.