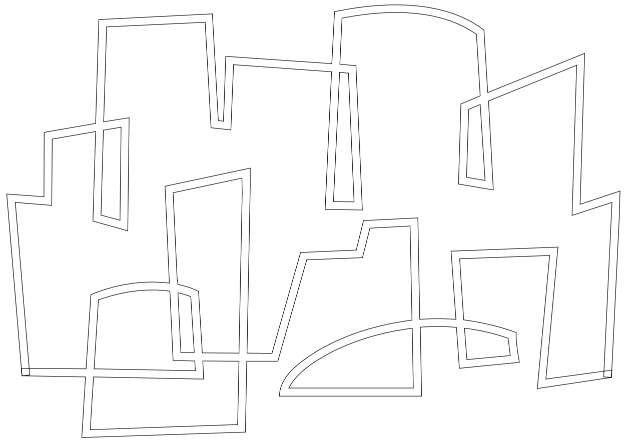
Scheduler-Based prefetching for Multilevel Memories

Derek Chiou, Srinivas Devadas, Josh Jacobs, Prabhat Jain,
Vinson Lee, Enoch Peserico, Peter Portante, Larry Rudolph

2001, July

Computation Structures Group
Memo 444



Scheduler-Based Prefetching for Multilevel Memories

Computation Structures Group Memo 444
July 2001

Derek Chiou

Srinivas Devadas

Josh Jacobs

Prabhat Jain

Vinson Lee

Enoch Peserico

Peter Portante

Larry Rudolph

G. Edward Suh

Dan Willenson

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.

Scheduler-Based Prefetching for Multilevel Memories

Derek Chiou Srinivas Devadas Josh Jacobs Prabhat Jain
Vinson Lee Enoch Peserico Peter Portante Larry Rudolph
G. Edward Suh Dan Willenson
Laboratory for Computer Science
MIT, Cambridge, MA

Abstract

Memory latency is a significant bottleneck in modern computing systems. With few exceptions, the process/thread/task currently running on the CPU implicitly owns all caches and main memory. Essentially, the memory is scheduled with the CPU. Generally, however, at least one memory resource is larger than any one process/thread/task and thus there is no benefit to assign that entire level of the memory hierarchy to that process/thread/task.

In this paper, we propose that memory scheduling, potentially all levels of the memory hierarchy, generally drives CPU scheduling rather than the other way around as it is done in most systems. While the CPU is running one job, cache/memory that is not being used by the currently running job are scheduled for a future job, generally the next job to run. Data is prefetched into those regions of memory using dedicated prefetch engines while the current job is executing on the processor(s). Prefetching requests have strictly lower priority than the running job's requests to eliminate impact to the running job's performance. Our prefetching is initiated by the scheduler, rather than in the application code.

To optimize overall performance, the data prefetched has to be judiciously selected, the amount of prefetched data has to be limited, and the prefetch has to execute at appropriate times. Since the cache/memory prefetch scheduler drives the CPU scheduler, however, at least correct timing is relatively straightforward. These decisions are made differently for different levels in the memory hierarchy. To implement the proposed scheme, we program a prefetch engine with instructions that bring data to a specified level in the memory hierarchy. We describe a hardware implementation of this scheme.

We present experimental results using prefetching in simulation applications, prefetching on a Web-server application, and prefetching when multiple processes timeshare a CPU. We show that significant performance improvements are possible using our methods. We believe that applications can be redesigned in a manner such that scheduler-based prefetching potentially eliminates all memory bottlenecks.

1 Introduction

Memory latency and bandwidth are significant bottlenecks in the performance of modern computing systems. Memory latency is ever increasing in proportion to processor clock periods. Similarly, package pin count is not increasing as quickly as die transistor count, making bandwidth in/out of the processors grow more slowly than raw processor performance.

For the most part, cache/memory is scheduled along with the CPU. Traditionally, the CPU was the critical resource and thus any job given the CPU was also given most other system resources. As CPUs became quicker, memory latencies started to add up, increasing their impact on overall performance. Techniques to tolerate latency and reduce memory bandwidth requirements (though there is no substitute for real bandwidth, many applications' bandwidth requirements can be somewhat alleviated with some of the same techniques that reduce latency, notably caches) became more and more essential.

Memory latencies are either tolerated (multithreading, out-of-order execution, prefetching, etc.), eliminated (caching, scratchpad RAMs, etc.) or both. In the case of prefetching, the processor and/or memory-aware application speculates on the addresses that will be accessed in the near future and attempts to load the corresponding data into registers, or prefetch it into a cache. Prefetching methods come in many different flavors. To be effective they must be implemented in such a way that they are timely, useful, and introduce little overhead [36, 13]. Software prefetch methods typically distribute prefetch requests throughout the application in an attempt to bring the data in before it is accessed. The main problem is to time software prefetches so the data arrives in time, but not too early to cause cache pollution. Hardware prefetch methods learn about memory access patterns and then attempt to predict them, and usually have less of a problem with timing. While compiler-based software prefetching can effectively predict future accesses, hardware prefetch methods are limited because they can only support relatively simple learn-and-predict schemes. On the other hand, hardware has full knowledge of the actual dynamic reference streams, while the compiler obviously does not.

In order to improve performance further, multi-threaded computers have been proposed and built [2, 27, 1, 5]. These processors store multiple contexts corresponding to different threads or processes, and may switch from one thread to another on a cache miss. While the processor is working on the new thread, data required for the old thread is brought to the processor or a nearby cache. In order for the processor to remain active, the data corresponding to at least one thread should be immediately available to the processor at all times. Of course, this one thread alone may not fully utilize the computational resources or functional units of the processor. To fully exploit the computational resources of the processor, many threads can be simultaneously executed [35], which in turn means that data corresponding to multiple threads should be available. One problem with switching very often between threads is cache contention across threads, which have to share a potentially scarce memory resource. Generally speaking it is better to "warm up" the memory system by executing the same thread, or set of threads for a while to get high hit rates, and not suffer misses because data that was accessed earlier gets replaced by other threads' data before being accessed again.

In this paper, we propose that memory be scheduled explicitly and that CPU scheduling implicitly follow the memory schedule, instead of the other way around as is done in virtually all modern computer systems. The CPU schedule may even be changed due to memory scheduling issues. Memory schedulers select the next job[1] to execute from a list of jobs that the CPU scheduler provides, preallocate cache/memory for it, and start to fill that preallocated memory. Requiring the CPU schedule to *follow* (of course there are cases where the CPU cannot follow the memory scheduler, but those cases are rare) the memory schedule ensures that the latter knows what job will be scheduled next. The cache/memory where the prefetched data is inserted will depend mainly on the amount of data that is prefetched and the space available in each level of the memory hierarchy. The prefetching of data will take place during the execution of another job, thus requiring careful scheduling of the memory resources so as to not interfere with the running job. Thus, the memory

[1] We will assume that multiple tasks comprise a thread, multiple threads comprise a process, and multiple processes time-share a CPU. For brevity, we will use the term job to refer to a task, thread, or process.

and CPU scheduler work together to optimize performance and resource utilization.

The advantage of our scheme is simple; by working with the CPU scheduler, we are generally able to get perfectly accurate information about the future which is generally unachievable for traditional software-based prefetching and hardware-based prefetching. Our scheme allows us to look far far enough ahead to mitigate many of the timing problems in traditional software prefetching and dramatically improve on the predictive abilities of purely hardware-based schemes. Since schedulers have global information about the readiness and latency of jobs, our prefetching methodology uses and influences all schedulers, including application-level, thread-level and/or process-level schedulers.

Note that our methodology can be used with both traditional software and hardware prefetch schemes. Such schemes have been shown to be effective, especially for programs with regular reference patterns. Such schemes could work in conjunction with our proposed scheme, passing information about what data will be useful in the future to avoid cold cache misses (software) and/or could run within the application itself to further improve performance (software and hardware). Our methodology has the potentially significant benefit of not requiring a recompilation to have the compiler insert prefetch instructions and has much better visability into the future than a pure hardware scheme.

To implement the proposed schemes, we use instructions that bring data to a specified level in the memory hierarchy.[2] Cache-load instructions will bring in blocks into the cache, whose sizes may be specified in the instruction, or may correspond to the block size of the cache that data is brought into. We describe how a hardware prefetch engine can be programmed with these *cache-load* instructions. Software prefetching can be used in place of the hardware prefetch engine. We mitigate pollution caused by prefetching by limiting the amount of data prefetched by the engine, and can control the bandwidth requirements by prioritizing demand data requests by the currently-executing job higher than prefetch data requests.

The rest of this paper is organized as follows. In Section 2, we first describe the overall philosophy of scheduler-based prefetching and memory management. In Section 3, we detail scheduler-based prefetching schemes for each software layer that we are considering. In Section 4 we describe prefetch hardware that uses cache-load instructions, and describe techniques to mitigate cache pollution and alleviate bandwidth requirements. In Section 5, we present experimental results that illustrate the varied ways in which we can use scheduler-based prefetching – within a single application, within a threaded application, and within a time-shared system with multiple processes. Related work is discussed in Section 6, and Section 7 concludes the paper.

2 Schedulers and Prefetching

We propose to schedule memory in a way very similar to the way the CPU is scheduled today. We wish to remove memory bottlenecks by looking ahead into the future to produce a memory schedule that is potentially different than, but related to, the CPU schedule for jobs. In order to look ahead with high clarity, we propose that the CPU schedule *follow* the memory schedule. The basic idea is illustrated in Figure 1.

[2] Such instructions are part of the IA-64 instruction set.

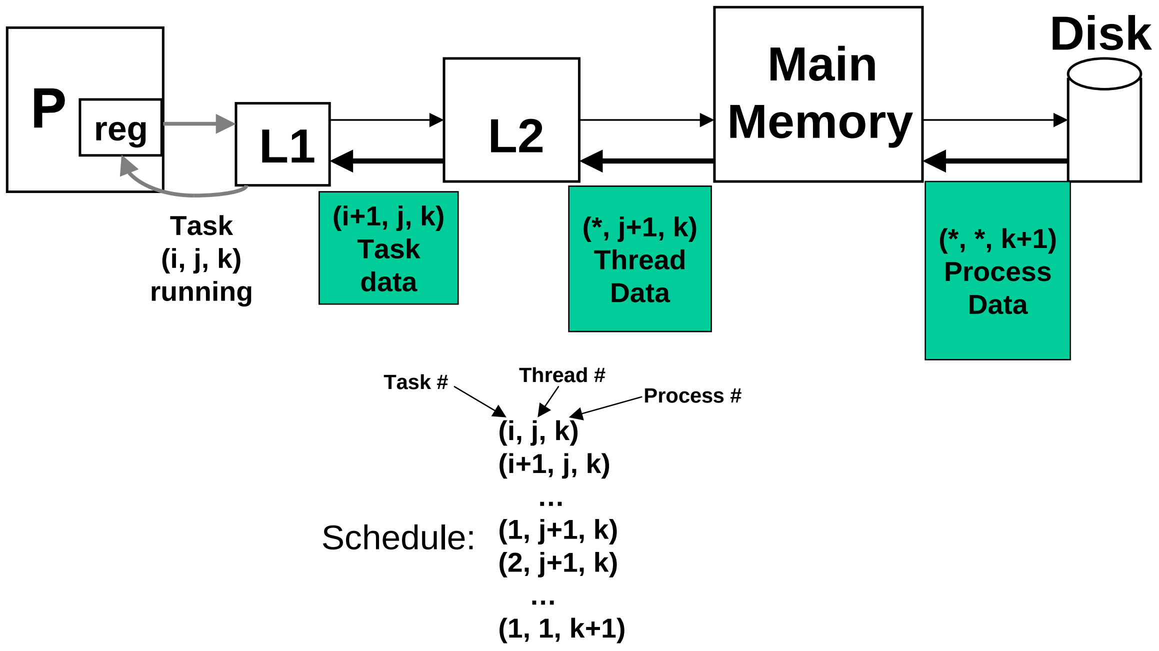


Figure 1: Removing Memory Bottlenecks Using Memory Scheduling. Multiple tasks comprise a thread, multiple threads may comprise a process, and multiple processes time-share the CPU. Each task is represented as a 3-tuple with (task #, thread #, process #). Note that the CPU schedule, for a set of processes with constituent threads and tasks, *follows* the memory schedule. In an ideal scenario, the currently-executing task only has to access L1 for all its data. Jobs (tasks) that will run in the immediate future are moved to the highest level of memory (L1), jobs (threads) that will run in the near future are moved to higher levels of memory (L2), and jobs (processes) that will run in the future are moved to lower levels of memory (main memory). Most of the memory resources are scheduled for jobs that will execute in the future.

How will we implement this idea? Given a set of jobs, which may be tasks within an application or thread, threads within a process, or processes themselves, schedulers schedule the CPU resources to execute these jobs. These schedulers have knowledge of jobs that are ready to run, and partially or completely control the order in which they will run. We propose that these schedulers schedule memory resources as well, i.e., these schedulers will move data which will accessed by jobs that will run in the near future *upward* in the memory hierarchy. CPU resources are scheduled, almost by default, to jobs that already have their data in the higher levels of the memory hierarchy.

Clearly, we can apply this strategy at different software layers and many different levels of cache. We illustrate an implementation of memory scheduling in Figure 2. Here, schedulers in different layers generate prefetch requests for data that will be accessed by the next job. The scheduler in each software layer is associated with *particular* levels of cache, and generates prefetch requests only for those levels.

We now relate the different software layers and associated schedulers to the type of job, size of footprints, and prefetch requests.

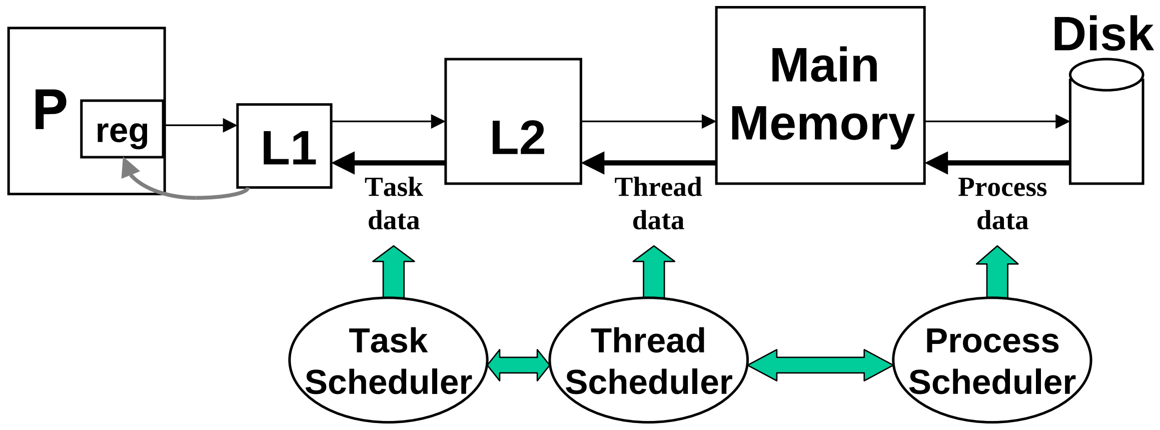


Figure 2: Strategy for scheduler-based prefetching. At each software layer, namely, task, thread, and process, we may have one or more schedulers that have knowledge of tasks or processes that are ready to run, and at least partially control the order in which they will run. We propose that prefetch requests be generated from within these schedulers. The scheduler in each layer is associated with particular levels of cache, and generates prefetch requests for those levels.

2.1 Process

Multiple processes time-share computer systems. Process schedulers are implemented in the operating system and typically use priority-based schemes to decide which process to run next. This scheduler can easily predict what processes are likely to run after the immediate next process completes its time quantum and prefetch data for those processes. Further, it is relatively easy to *force* the scheduler to respect previous predictions, implying that the prediction accuracy can be 100% – a concrete example of the memory schedule driving the CPU schedule.

The footprint of a process has to fit on disk but may or may not fit into main memory. Quite often, the sum of the footprints of all active processes do not fit into main memory. The process scheduler will initiate prefetch requests to transfer data from disk to main memory for a process that it predicts will run in the near future. This data will *not* be pulled into higher levels of cache by the process-level scheduler. [3]

The prefetch requests are serviced during the execution of the current process. After the process has finished executing, the predicted process will already have its data in main memory, generally dramatically reducing startup page faults.

2.2 Thread

A process or application may be comprised of multiple threads. Applications either have internal thread schedulers that determine which thread(s) will run next or rely on a thread scheduler provided by the operating system. Examples of this are webservers and other interactive applications. Like process schedulers, thread schedulers can predict the future schedule of threads, and can be modified to follow previous predictions.

If many threads comprise a process, these threads will often have smaller footprints than the process. The footprint of a thread, or all threads associated with an application, will usually fit into

[3]However, it may be pulled closer to the CPU by thread-level or task-level schedulers, and, of course, when the data is actually accessed.

main memory, but may or may not fit into an L2 cache. The thread scheduler will prefetch data from main memory to the L2 (or L3) cache, but will not pull the data to higher levels, e.g., L1. Thus, when threads begin executing, their data will be in L2 (or L3).

2.3 Task

We will deem a thread or an application to be comprised of a set of tasks. These tasks could correspond to invocations of a procedure or subroutine, or an invocation of a collection of procedures, with particular input stimuli. Some applications, such as simulators and signal processing applications, have a scheduler embedded within them. This scheduler determines which task is going to run next according to some metric or algorithm. The tasks that can run are those that have their inputs available, and the scheduler has a choice as to what task to run, using knowledge about the partial ordering of tasks. For example, in a timing simulator, a task corresponds to the simulation of a module. The simulator uses an event time-wheel to schedule tasks. Multiple tasks attached to the same time can be processed in any order.

The footprint or data set associated with a task is typically small and will usually fit into L2 cache, but may or may not fit into a typical L1 cache. The task scheduler will generate prefetch requests to bring task footprints or parts of task footprints into the L1 cache, before the tasks are executed.

3 Prefetching and Cache Management

In this section, we first focus on the issues relating to the management of the cache resource under prefetching schemes (cf. Section 3.1). We consider detailed prefetching schemes for the process, task, and thread levels in Sections 3.2, 3.3, and 3.4, respectively.

3.1 Cache Usage

We first consider a simple scenario of a round-robin schedule of jobs[4], and we focus on a single cache within the system or the main memory. We will refer to the execution of a job as a *run* – a particular job may execute many times in a schedule and will therefore have different runs. The data in the cache are dependent on the order of jobs that have run thus far. In other words, cache allocation is completely determined by the jobs that are running or have run in the past. A snapshot of the cache is shown in Figure 3(a), where the data associated with different jobs, A, B, and C, are illustrated. In Figure 3(b), we illustrate the basic prefetching mechanism that schedules the cache resource based on the upcoming (or future) schedule of processes.

We considered a round-robin schedule of jobs because it is predictable and corresponds to a very simple scheduler. The jobs themselves could be processes or threads time-sharing a CPU, or the jobs can be tasks within a loop. Of course, in general, we may not have iteration or round-robin scheduling, just a sequence of tasks, threads, or processes time-sharing a cache. Still, the data corresponding to the last few tasks that have run will occupy the cache in a similar fashion.

[4]Recall that when we refer to a job, we could mean process, thread, or application task.

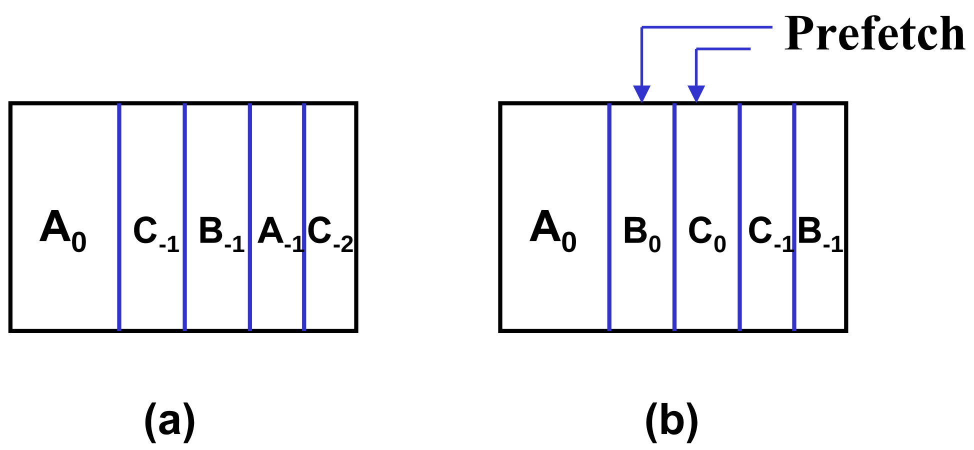


Figure 3: Cache Snapshots. The round-robin schedule is A, B, and C. (a) When A runs, there is data in the cache corresponding to the current run of A marked A_0. The previous runs of C, B, and A are marked C_{-1}, B_{-1} and A_{-1}. It is also possible that data from earlier runs such as C_{-2} is still in the cache. The running jobs completely determine cache allocation. (b) Given knowledge of the schedule of jobs, when a particular job, say A, is running, we can schedule parts of the cache for B and C, marked B_0 and C_0 in the figure. Data that the next run of B and C may access are brought into the cache when A is running.

3.1.1 Cache Footprints

Let us now look more closely at the cache usage of a particular job, which may be a task, thread, or process.

There are essentially two cases. When a job, say A, runs, it could use up the entire cache or a fraction of the cache. Which of these occurs depends on

1. the amount of time A has executed for, call it T, [5]
2. the number of memory references it makes, and
3. the size and organization of the cache.

The *footprint* of a job at any time in a particular run is the set of data words corresponding to the job present in the cache at that time. The *reuse footprint* of a job at any time in a particular run is defined as the set of data words already in the cache which will be *reused* by the job before the end of the run. By definition, the reuse footprint of a job is empty at the end of its run, though it might not be empty at the beginning – for example, data is prefetched before the beginning of the run. Also, again by definition, the reuse footprint is never larger than the footprint but may be smaller, (e.g., Figure 4(a)) since "dead" data which will not be used before the end of the run remain part of the footprint but not of the reuse footprint.

Figure 4 shows *reuse footprint* curves as a function of time for two main cases, (a) when a job does not require the entire cache to run efficiently and (b) when the job does require the entire cache. In the figure, T corresponds to the duration of the run.

More complicated cases than the above two cases in the figure may occur, but we will not consider them here.

[5] In the case of the job, being a process this is the time quantum of the process.

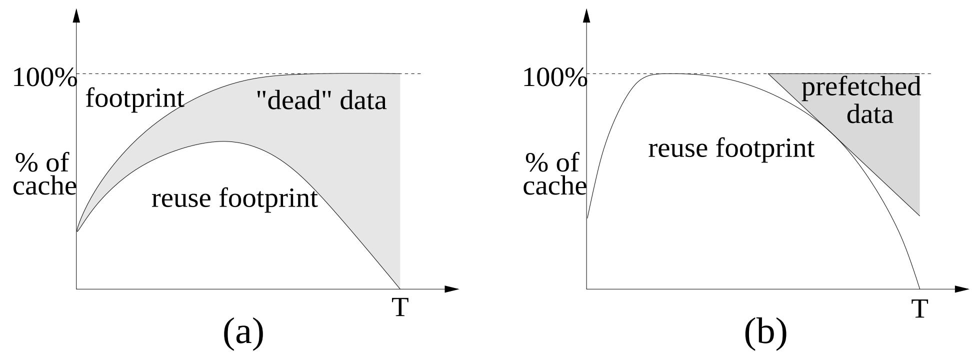


Figure 4: Reuse footprint of a job as a function of time. (a) The job initially begins with some data (or perhaps none) in the cache and gradually increases the amount of cache that it occupies. Toward the end of the run, data in the cache, or data brought in by the job, effectively becomes dead, i.e., is not part of its reuse footprint because the job will not touch the data before it is switched out. (b) Similar to (a), except that the job fills up the entire cache during the time that it runs. Even in this case, we can prefetch data if we start late in the time quantum. The slope of the "prefetched data" line is the bandwidth that we can bring data into the cache.

3.1.2 Amount of Prefetch Data and Pollution

It is obviously advantageous, if sufficient bandwidth is available, to replace "dead" data with data prefetched for the next job(s), since such data will not be used again until the current job is scheduled again. Since the reuse footprint goes to zero at the end of the time quantum, it should always be possible to find space in the cache to accommodate prefetched data for future jobs, even if the reuse footprint of the current job fills the entire cache at some time during the run (e.g., Figure 4(b)), as long as prefetching begins sufficiently late that interference with the current job is avoided. Of course, determining which cache-lines to replace may be difficult.

It is easy to see that, if we begin to prefetch data at time $T_{prefetch}$ [6], the amount of data D that can be prefetched at time $t > T_{prefetch}$ without interfering with the currently running job, given an available bandwidth W, will be:

$$D(T_{prefetch}, t) \leq \int_{T_{prefetch}}^{t} W(\tau) d\tau \qquad (1)$$

$D(T_{prefetch}, t)$ can only grow the further back in the past $T_{prefetch}$ is, i.e., the earlier we start prefetching. We want, therefore, $T_{prefetch}$ to be the *earliest* time such that, from then to the end of the run, there will be enough space in the cache to accommodate both the prefetched data and the reuse footprint of the current job (see Figure 4(b)). Therefore, for any t, such that $T_{prefetch} < t < T$, if we denote with $RF(t)$ the reuse footprint of the current job at time t, with C the size of the cache, and with D the actual amount of data that we prefetch, we require:

$$D(T_{prefetch}, t) + RF(t) \leq C, \ for \ T_{prefetch} < t < T \qquad (2)$$

We find the earliest $T_{prefetch}$ that satisfies Eqns. 2; then, the maximum amount of data we shall be able to prefetch is $D(T_{prefetch}, T)$.

[6] hereafter we shall assume that $T_{prefetch}$ is the time when prefetched data begins *arriving* into the cache. If the latency involved is L, then we must obviously begin issuing the prefetches at time $T_{prefetch} - L$

If such an amount is less than what we would want to prefetch for the next job, we might schedule a different, less "data hungry" job. Conversely, if $D(T_{prefetch}, T)$ is more than what is necessary for the next job, we might begin prefetching of data of jobs still further in the future or schedule a job that requires more data to be prefetched. The timing and amount of data obviously depends on whether we are prefetching for processes, threads, or tasks (cf. Sections 3.2 through 3.4).

3.2 Process-Level Prefetch

A process scheduler determines the amount of time that each process will run, i.e., the assigned time quantum of the process. A process may not finish its time quantum because of an interrupt, a page fault, or an exception.

The timing of the prefetch is dependent on whether the process corresponds to case (a) or (b) in Figure 4 for the length of time it *actually* runs, namely T. We will have to guess whether the process corresponds to case (a) or (b), i.e., estimate the reuse footprint size and compare it to the size of the cache or memory we are prefetching into. We can start prefetching early in the time quantum if the process corresponds to case (a) late if the process corresponds to case (b). The operating system typically gives indications to the current process about elapsed time, which can be used to trigger prefetching.

Next, we describe the steps of scheduling of the next job, estimation of the reuse footprint size, and how the data to be prefetched is selected.

3.2.1 Scheduling of the Next Process

Rather than determining the immediate next process to run, we will modify the CPU scheduler to determine a set of processes that can be run in the next time slot. The memory scheduler then selects a process from that set and informs the CPU scheduler of its decision. That process should generally be used as by the CPU scheduler as the next process to be scheduled.

Schedulers can be relatively easily modified to include this feature. Round-robin schedulers are a trivial case, and we can obtain 100% prediction accuracy. Priority schedulers with priority classes that assign varying time quanta to processes based on past history [34] are also easy to modify. Consider a priority scheduler with priority classes. Runnable processes in the highest class are run before processes in lower classes. Further, the class of a process typically changes after each run. The scheduler can easily determine the next two highest priority processes. The only issue is that while the first is running, the second process (which should run next) may be preempted, i.e., a process may become runnable and be placed in a higher class. Many preemptions can be eliminated by simply forcing the earlier decision to be used; unless there are hard real-time constraints, there is generally no penalty for doing so. In general, enforcing the decision of the next process to run should not affect the performance of the scheduler, which is typically trying to optimize some metric relating to response time for interactive applications and completion time for all processes. Even if a hard real-time constraint dictates that the predicted process cannot run, then we will have prefetched data for the wrong process for just this one instance.

If we know the memory characteristics of processes, we may wish to use the information to control how far into the future a particular process may run. For example, a process P that touches a lot of data may require a lot of prefetching for its performance to improve. This means that we should probably run one or more processes that have footprints of the type corresponding to Figure 4(a) before this process. This way, we will have cache/memory space, and plenty of time to prefetch P's data in advance of P running. Also, in common OSes today, jobs are organized into priority queues, with the least CPU intensive jobs being given the highest priority, and being scheduled more often and for shorter periods, while CPU intensive jobs are scheduled less often, but retain the CPU for more time. With memory and bandwidth being the bottleneck, we may wish to schedule jobs based on how *memory intensive* they are, with the most "memory hungry" jobs being scheduled less often and yet for longer periods, avoiding excessive swapping and using the memory system fairly but efficiently.

3.2.2 Estimating Footprint and Reuse Footprint Size

We need to know the reuse footprint size to decide when to prefetch for a process, early or late, in the time quantum.

We estimate the footprint and reuse footprint size based on the history of the process in its previous runs. Computing the footprint of a process is easy. We can maintain a table of accessed pages, and the cardinality of the table is the footprint size at any given point of time.

To compute the reuse footprint, we maintain a table of accessed pages, and store two additional numbers in the table. The first corresponds to the first time that the page is accessed, and the second the last time it was accessed. The page number is placed in the table when it is accessed first, and, in this case, the two numbers are identical, and equal to the time that it was first accessed. Once the time quantum ends, for each chosen time point we simply scan the table to determine the pages that are part of the reuse footprint. We get a reuse footprint size for each time point. The granularity of time that we are interested in is quite coarse – we merely want a few points through the time quantum. Thus, the storage requirements are quite small.

A process may not run for the same amount of time in each run because its assigned time quanta may be different, or it may be prempted due to an exception or interrupt. Let us say that we have a reuse footprint curve for a process P computed after a particular run of length T_0. Assume that P executes identical instructions in a second run, but runs for a time $T < T_0$. Then, the reuse footprint curve for the shorter run will be strictly *below* the curve for the longer run. The longer run's reuse footprint serves an upper bound for the shorter run, and we can use it to determine prefetch timing without worrying about causing pollution.

Thus, if we have a reuse footprint curve over time T_0 for process P from a previous run, then we only use it to decide prefetch timing as described in Section 3.1.2 when P is running with an assigned time quantum $T \leq T_0$.

3.2.3 What to Prefetch?

We will prefetch to main memory (and possibly into L3). We will prefetch pages of data corresponding to a process. The amount of data to prefetch and the timing of the prefetch is determined using the reuse footprint curve as described earlier.

The operating system will maintain the least recently used (LRU) information, or some approximation thereof, of the pages corresponding to each process. Either at the beginning of the time quantum or toward the end of the time quantum, we will start prefetching the most recently used

(MRU) pages for the predicted next-to-run process or thread, and continue to prefetch till all the data has been prefetched, or the currently-running process is swapped out. Note that the prefetching into memory will occur when there is bandwidth available because the currently-running process is not using up all the available bandwidth from disk to memory. This will most likely be the case when the prefetch accuracy is high.

If an L3 cache exists in the memory hierarchy, we can pull a subset of the pages brought into main memory into L3. However, we will not consider L3 in this paper.

3.3 Task-Level Prefetch

Applications have a control structure that determines the tasks that will be executed when the application is run with specific input stimuli. As a simple example, an application may continually read in inputs and process the inputs in order. Alternately, the application may have a set of objects that need to be processed, and a "scheduler" may determine the processing order of these objects. Or an interactive application may wait for inputs and perform computations based on the inputs.

To perform task-level prefetching we have two main requirements. First, each task has to have an identifiable data set or footprint.[7] We will then know the data that has to be prefetched for a task that will run in the near future. Next, the application, or parts of the application, have to be structured in such a way that sequences of tasks can be predicted reasonably accurately. Many applications that do not satisfy this requirement can be re-written to enable task-level prefetching.

For example, when an application reads in inputs from a file, and processes them in order, the sequence of tasks simply corresponds to, calling a given routine with different inputs from the file. The "scheduler" is then a simple looping construct. We know the n tasks that will be executed, if we read n inputs from the file. If we can relate data objects to these n tasks, we can prefetch some or all of them.

An application may have an embedded scheduler that deals with input events, or schedules computations based on data structure traversal. Branching search, depth-first graph traversal, logic simulation, etc., are specific examples. We can buffer input events or use look-ahead in the traversal to predict sequences of future computations.

If the application satisfies our two requirements, we perform task-level prefetching in a similar manner to process-level prefetching, but with some differences. Since we do not have an assigned time quantum for a task, we will not use the reuse footprint to determine prefetch timing. The scheduler predicts tasks, and the task software routines relate task inputs to data objects. Since we are modifying the task scheduler, we will have access to the task software data objects. We can estimate the size of the tasks' data objects, to determine if we can prefetch all of the data or only part of it. Since we will typically be dealing with a small L1 cache, we will have to be careful not to pollute it, and we will limit the amount of prefetching to a fraction of its size.

In Section 4.4, we describe ways to limit or eliminate cache pollution. Prefetching will occur when tasks are running, however, the current task's data will always have priority. We discuss this in Section 4.3. Limiting pollution and prioritizing bandwidth requirements ensure that the performance of the current task is not degraded.

[7]Not all tasks in the application need to satisfy this requirements – only those that we will prefetch data for.

3.4 Thread-Level Prefetch

Threads are similar to processes except that they share address spaces and I/O descriptors. One main objective of threads is to enable fast context switches without changing address spaces. In terms of scheduling, threads can be supported by either an operating system or a user-level library.

If threads are mananged only by the kernel, thread-level prefetching will be the same as process-level prefetching, with only the targeted level of memory being different. However, if we use a user-level library to manage threads, we have a situation akin to task-level scheduling, since the application/scheduler has more information about latency and parallelism. Page faults and I/O are still supported by kernel, and the kernel can still preempt an application for time sharing. Within the assigned time quantum, however, threads are scheduled based on the characteristics of the work (as in the case of tasks) and not based on time quanta. For example, in a webserver, there will be a number of connections, and since it takes some time to receive a reply over the network, threads will be written to yield a CPU just after sending out a message.

The internal scheduler knows what threads are ready to run, and will select threads to run, in some order. The internal scheduler also has knowledge of some of the associated data for each thread, for instance, the stack, and the control structure. The programmer can also explicitly associate other data that a thread touches to the thread. We will prefetch all known associated data for threads that will run in the future, using the same "next two or more" prediction scheme used in process schedulers. Typically, footprints of multiple threads will fit into L2, and we will simply bring in the entire data set of the thread(s) that we believe will execute after the immediate next thread.

4 Cache-Load Instructions and Implementation

The memory scheduler driving prefetch engines relies on efficient and precise prefetch instructions. There are no prefetch instructions in any widely available general-purpose processors that do exactly what we want, notably prefetch into a specific level or levels of the memory hierarchy. In this section we describe existing prefetch instructions, then go on to describe how one might implement the prefetch instructions we need, the bandwidth considerations and how pollution can be controlled.

4.1 Existing Cache Prefetch Instructions

To our knowledge, there are no cache prefetch instructions implemented in current commercial, general-purpose microcprocessors, that allow the user to target a specific cache-level. There are, however, several processors that implement prefetch instructions, which load data into caches (assuming the data is not in the caches at the time the instruction is issued). The PowerPC instruction set [20], for example, includes `dcbt`/`dcbtst` (data cache block touch and data cache block touch for store respectively). The instructions are prefetching hints to the processor; the processor can decide whether or not to prefetch the cache-line corresponding to the address provided to the instruction. The PowerPC 604, for example, will perform the prefetch if the address hits in the TLB or BAT (block address translation table) and the address is cachable and accessable by the running process.

The Alpha instruction set [11] contains similar instructions to the PowerPC, but provides an additional prefetch instruction that sets the prefetched cache-line to be the first to be replaced.

The UltraSparc-II processor [33] implements the V9 prefetch instruction, prefetching data into the external L2 cache and not into the on-chip L1 cache.

The Intel Itanium processor [14] implements line prefetch instructions (`lfetch`) that contain an additional 2b hint completer field that could potentially be used to specify a level or set of levels of the cache hierarchy to fetch into.

4.2 Cache-load instruction hardware

Prefetch engines are associated with each cache (including memory). The cache-load instruction passes a command to one or more cache prefetch engines that then prefetch the requested data into the appropriate cache(s).

There are many possible implementations. Ideally, the issuer of the cache-load instruction should not have to issue a separate instruction for each cache-line to be prefetched. Issuing potentially many prefetches will take time in itself and tracking which lines should be prefetched may negatively impact the cache. Instead, the issuer should be able to issue a block of prefetches with a single command. A simple block prefetch command would combine n continuous cache-lines starting at an address A. For byte-addressable machines, one representation of such a command would use the bottom $\log(C)$ bits to represent n, where C is the number of bytes in a cache line. Therefore, n cannot be larger than the number of bytes in a cache-line. Larger n could be provided in the same manner, but would require that blocks be aligned to multiples of cache-lines. By providing additional state (perhaps the contents of another register), finer control of the block's initial address and size can be supported.

The cache-load instruction would also encode the information as to which cache-level the data should be loaded. One way to do so is to have the cache-load instruction travel along the same path as, but using different resources than a standard load/store operation. Higher-level caches will simply pass the command down to the next lower level. Once the command reaches the target level, the prefetch engine expands the command into the appropriate number of loads and issues them. As the loads return, their data are inserted into the cache and the operations are terminated.

Ideally, the command is passed down without expansion, to reduce command bandwidth requirements, but doing so may be difficult since the requested data may be spread across multiple levels of the memory hierarchy. This would require at least partial expansion at each level or it may waste snooping bandwidth if the expansion is done at the lowest level and the snooping mechanism used to maintain conherency. In either implementation, the curious caching mechanism [10][8] is very useful.

4.3 Bandwidth Considerations

Bandwidth must not be stolen from load/store operations being issued by the currently-running job to avoid slowing it down. Thus, cache-load operations operate at a strictly lower priority than

[8]Curious caches can be made curious about address regions. When the cache snoops addresses belonging to a region it is curious about, it copies that data into itself in a manner specified by the curiosity mechanism which may be dependent on the curiosity region and may even forward the data up to the next level of the memory hierarchy.

standard load/store operations and thus use any bandwidth unused by the running job. Arbitrators at each level will enforce this constraint.

4.4 Limiting Cache Pollution

Prefetching potentially pollutes the cache with newly prefetched data. There are a few ways to prevent this.

The easiest, but not the best, way is to just limit the amount of prefetch data to be a small fraction of the cache size. In this case, we rely on the associativity of the cache and the LRU policy to limit pollution of data of the currently-running job. Further, we rely on the scheduler to limit its prefetch requests, otherwise there is no control over cache pollution.

A second way is to provide the running job and the prefetched job an independent set of cache resources. Many techniques have been described to dynamically partition a cache/memory[25, 4, 30, 10]. By prefetching into a region of the cache that is reserved for the prefetched job, we can guarantee that the running job's footprint will not be polluted. This way, the schedulers can make an arbitrary number of prefetch requests and the latest prefetched data will exist in the cache – earlier data will be overwritten. But, the currently-running job's footprint will not be affected.

Another way to control pollution is to mark old data that will not be used by the currently-running job as being *invalid* or flushing that data from the cache. Old data corresponding to jobs that have run prior to the current job can easily be marked as invalid from within the scheduler. That is, the scheduler not only generates prefetch requests, but it also marks data as being invalid, once it knows that the task associated with the data has completed. This strategy would require additional kill bits for cache blocks, and software instructions to set these bits appropriately. We can follow a strategy where prefetched data only replaces invalid data, or a strategy where invalid data is preferentially replaced over unmarked data. Kill bits for cache blocks corresponding to the currently-running job can also be set, if the scheduler knows that the job will not access this data again in the current run.

5 Experiments

5.1 Processes: Prefetching into Main Memory

In this section we present results using scheduler-based prefetching for time-shared processes. We chose benchmarks with large footprints that result in a large number of page misses when they time-share a CPU and memory, even for a memory of significant size. We demonstrate that prefetching can almost completely eliminate cold page misses in this scenario. Even if we cannot force the CPU schedule to follow the memory schedule, memory and overall performance can be improved significantly. We present results for different sizes of memory and different mixes of processes.

Several programs from SPEC CPU2000 benchmark suite [12] 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 memory consumed by each benchmark when the memory is dedicated to that process. Note that this memory usage is for the case when

Name	Description	Input	Memory Usage (MB)
`gcc`	C Programming Language Compiler	200	28.7
`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

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

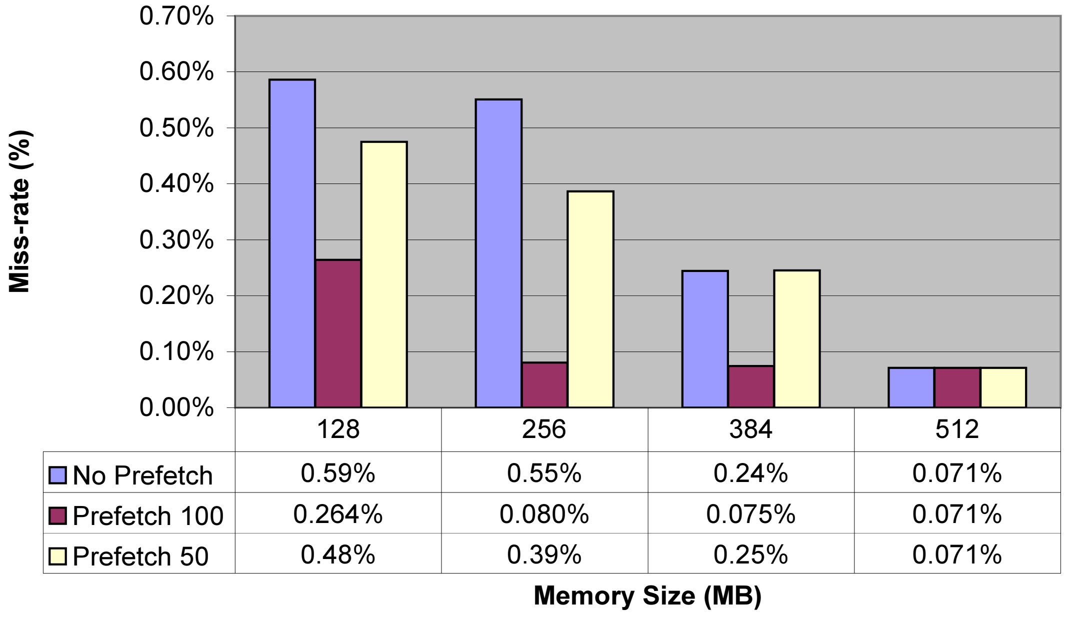
a process executes for a long time, therefore processes usually consume less memory space over one time quantum. Memory traces are generated using SimpleScalar tool set [7], assuming that processors have 4-way 16-KB L1 instruction and data caches, and an 8-way 256-KB L2 cache.

First, we simulated cases when eight processes execute concurrently with memory ranging from 128 MB to 512MB. The eight processes 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 processes (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 process's pages in the memory with their LRU ordering. It also records the latest reuse footprint curve for each process. For each time quantum, pages for a predicted next process are prefetched starting from the MRU page. The prefetch timing and amount of data is based on the reuse footprint as described in Section 3.2. To memorize pages, we need 20 bits per page assuming that a page is 4 KB. Therefore, about 160 KB is required for each process if the memory is 256 MB. However, this overhead is negligible since we only need to have this data structure for the current process and the next process in the memory no matter how many processes are in memory. The data corresponding to the reuse footprints has to be stored for all active processes, but this data is a small table with page counts for about ten time points.

Simulation results of eight-concurrent-process cases are summarized in Figure 5. For each memory size, we compared three different prefetching strategies: no prefetching, prefetching with perfect process prediction, and prefetching with 50% process prediction accuracy. No prefetching stands for the standard LRU page-replacement policy which are used by most modern systems. Prefetching with perfect process prediction is an ideal case of the CPU schedule following the memory schedule; this can be realistic when we have modified the scheduler as discussed in Section 3.2.1. Finally, prefetching with 50% process prediction accuracy stands for cases where the next process cannot be predicted accurately, for example, because of hard real-time constraints. We believe that 50% is a very pessimistic number. However, it is worthwhile to see what the benefit of process scheduler-based prefetching is for this case.

First of all, the simulation results demonstrate that time-sharing can severely degrade the mem-



	128	256	384	512
🟦 No Prefetch	0.59%	0.55%	0.24%	0.071%
🟥 Prefetch 100	0.264%	0.080%	0.075%	0.071%
🟨 Prefetch 50	0.48%	0.39%	0.25%	0.071%

Figure 5: The effect of process-level prefetching on memory miss-rate: comparing miss-rates for no prefetching (No Prefetching), prefetching with perfect process prediction (Prefetching 100), and prefetching with 50% process prediction accuracy (Prefetching 50).

ory 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 (1 million references equals to 1ms assuming a 1GHz processor that issues a memory reference every cycle), context switches cause a significant amount of additional misses if the memory cannot hold the entire working set. Although memory is becoming larger, so are programs and their working sets. In fact, larger programs will cause more problems for context switching since they require more cycles to reload a working set.

Fortunately, simulations show that in many cases scheduler-based process prefetching can almost completely eliminate the misses caused by context switching. For 256 MB memory, which is large enough to hold the working sets of any two processes but not large enough to keep the entire working set, 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 process and the next process such as 128 MB in this experiment, prefetching pages for the next process can harm the performance of the current process and we cannot prefetch all blocks for the next process. However, we can still obtain noticeable improvement of the miss-rate, by timing the prefetching properly using the methods outlined in Section 3.1.2 and 3.2.3. The only case when 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 process are spculatively prefetched.

Finally, the simulation results show that scheduler-based prefetching can still eliminate a signifi-

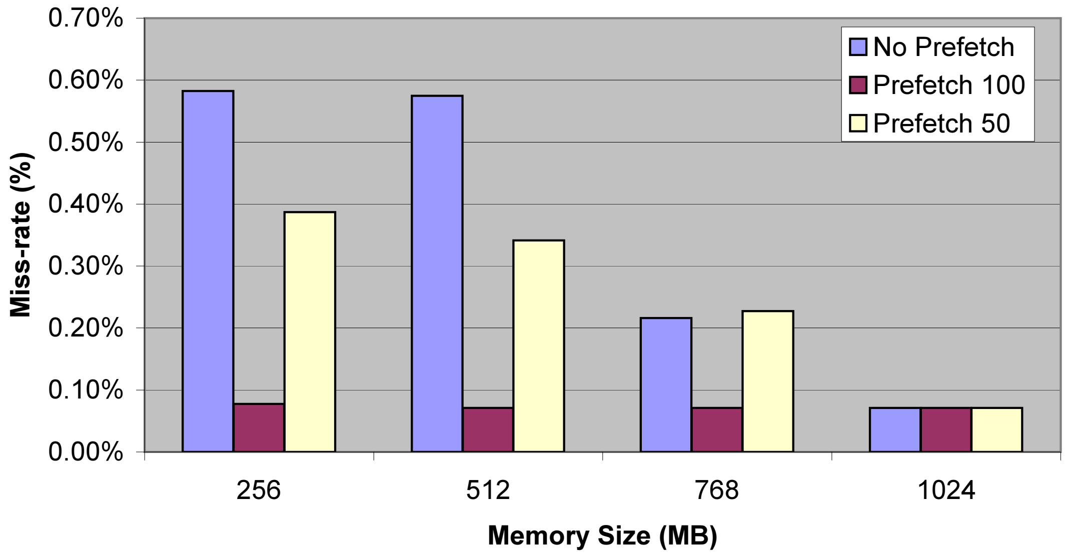


Figure 6: The effect of scheduler-based process prefetching on memory miss-rate when the number of processes is doubled: comparing miss-rates for no prefetching (No Prefetching), prefetching with perfect process prediction (Prefetching 100), and prefetching with 50% process prediction accuracy (Prefetching 50).

cant portion of misses caused by context switches even if the next process prediction is not correct. This implies that there is very little harm to the miss-rate of the current process even though we prefetch blocks of the wrong process. This is because most of the time we are evicting blocks of processes that have run in the past, or evicting the least recently used blocks of the current process. In some cases, the next process that is going to run already has some data left in memory, and this data might be evicted. This, too, happens infrequently.

The memory traces used in this experiment have memory usages smaller than 120 MB. As a result, time sharing did not matter for main memory that was 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 processes increase, the size of the entire working set also increases. Therefore, for larger applications, or a larger number of applications, the memory size where process-level prefetching can help will scale up. For example, we just increased the number of processes by duplicating the traces used before. As the result in Figure 6 indicates, process-level prefetching significantly improves the miss-rate even for 512 MB memory.

Process-level 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 since, for modern processors, there is a factor of 10^3 to 10^4 in the access time for memory versus disk. Therefore, an improvement in miss-rate from 0.55% to 0.08% (256MB memory in Figure 5), implies a memory performance improvement in the range of 3.6X to 6.2X.

5.2 Prefetching into L2

We present results using the Apache webserver in this section. Our experimental setup has the Apache webserver running on a machine, with SpecWeb running on another machine, generating requests for the webserver. There were over a hundred connections made. We use SimICS [19] to generate traces for the Web server. Modifying the LINUX kernel, we generated a trace using SIMICS whose addresses were tagged with the process ID's of the child processes forked by the Webserver.

We conducted a similar experiment to that of Section 5.1, except that the processes all belong to the Webserver. We simulated the trace using a detailed cache simulator with a 32KB 4-way L1 cache, a 4-way 2MB L2 cache, and an infinite-sized memory. We maintained an LRU ordering of the pages for each process, and we prefetched pages for a predicted next process are prefetched starting from the MRU page. As before, the reuse footprint was used to decide prefetch timing and the amount of data.

Unfortunately, we were unable to fully verify the results of our experiments in time for this submission. A complete set of experiments, with varying cache sizes and organization, for multiple traces with different Specweb usages, will be presented in the final paper.

5.3 Tasks in Applications: Prefetching into L1

This subsection presents a simple experiment using task-scheduler-based prefetching into L1.

We show that the hit rate of an application with an internal task scheduler can be improved, if the scheduler can request prefetches of task footprints, or parts of task footprints. We present these preliminary results more to illustrate the viability of task-scheduler-based prefetching, rather than to show a benchmark for which memory performance improves significantly.

Our targeted application for prefetching into L1 is `csim`, a cache simulator. By simulating cache accesses to set-associative caches using the LRU replacement policy, `csim` generates aggregate hit and miss rates of an input stream of memory addresses. `csim` has a simple scheduler, which reads input addresses from a trace file, and simulates them, one by one. We have modified this scheduler to read groups of addresses from the input trace file. Since these addresses have to be sequentially simulated, we can look ahead in the group to determine which input address will be simulated next. The scheduler determines which sets of the simulated cache these addresses will access and the data structures corresponding to these simulated sets can then be prefetched. Therefore, when these addresses are simulated by `csim`, the data structures these addresses need are already in L1.

Next, we ran `csim` on `sim-fast`, one of the simulators from version 3.0b of the Simplescalar tool set [7]. (Minor modifications were added to `sim-fast` to output traces containing memory accesses and prefetch hints.) The resulting traces from `sim-fast` were used as inputs to a detailed cache simulator, to generate results on various cache configurations.

Table 2 shows the hit rate numbers produced by running `csim` on different cache configurations. The results show that, for the application `csim`, prefetching initiated by a memory-aware scheduler can improve memory performance. The same strategy can be applied to other applications with internal schedulers.

6 Related Work

Giving control of memory management decisions to applications has been suggested by many researchers. A large fraction of the work in this area has focused on putting this functionality into

Configuration		Total Refs	Hits	Hit Rate
32 KB, 8-way,	prefetching	500,904,762	498,821,380	99.58%
32-byte cache line	no prefetching	500,904,762	495,242,315	98.87%
64 KB, 2-way,	prefetching	500,904,762	498,890,163	99.60%
32-byte cache line	no prefetching	500,904,762	495,249,001	98.87%
64 KB, 4-way,	prefetching	500,904,762	499,396,796	99.70%
32-byte cache line	no prefetching	500,904,762	495,747,305	98.97%
64 KB, 8-way,	prefetching	500,904,762	499,373,155	99.69%
32-byte cache line	no prefetching	500,904,762	495,734,263	98.97%

Table 2: Memory performance of `csim` on different L1 cache configurations with and without prefetching.

the operating system [29] [21] [32]. Giving applications more control over physical resources other than memory has also been suggested in extensible operating systems [15] [31].

Integration of prefetching and replacement decisions in the context of I/O prefetching for file systems has been proposed [8] [28]. The work of [22] chooses the next access based on information about the latency of access.

There have been many proposals for software prefetching in the literature. Many software techniques are customized toward loops or other iterative constructs (e.g., [16]). These software prefetchers rely on accurate analysis of memory access patterns to detect which memory addresses are going to be subsequently referenced [24] [18] [17] [26].

Software has to time the prefetch correctly – too early a prefetch may result in loss of performance due to the replacement of useful data, and too late a prefetch may result in a miss for the prefetched data. In our work, timing is not a significant issue because we are prefetching for the next process or thread.

Techniques to model and manage cache pollution in prefetching have been proposed (e.g., [9]). Our methods are based on the reuse footprint of processes to mitigate, if not avoid, problems caused by cache pollution.

Mowry, Demke and Krieger [23] and Brown and Mowry [6] 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 of [23] targeted improved performance for a single out-of-core application, whereas the techniques of [6] considered multiple processes with both out-of-core applications and interactive processes in the mix. Brown and Mowry [6] 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.

Load/Store instructions have been proposed [3] that branch if the data is not in the cache, but otherwise execute to completion. Such instructions should be fairly straightward to implement and allow application code to implement their own multithreading to tolerate memory latency.

Our work differs from the above in that we have considered caches and memory, and we prefetch the next process', thread's or task's data. We replace data in the current level of memory that typically does not belong to the currently-executing process or thread.

7 Conclusion

To eliminate or mitigate the problem of memory latency, we have proposed a system wherein memory scheduling drives CPU scheduling. An implementation of this concept is scheduler-based prefetching, where prefetch requests are generated from within schedulers, rather than distributing prefetch requests across application code or multiple applications. We believe this is advantageous in many cases, because schedulers have prior knowledge as well as control over what jobs will run in the near future, and we exploit this knowledge and control to improve performance. This strategy can be applied at different software layers. For each software layer there are appropriate levels of the memory hierarchy which are targeted by the scheduler when generating prefetches.

While our experimental results are preliminary, we believe that scheduler-based prefetching holds great promise in improving memory and overall performance in many different settings.

Ongoing work includes scheduler-based dead block prediction in caches, as well as investigating whether individual applications can be rewritten so task-based prefetching can be applied to improve performance.

References

[1] A. Agarwal, R. Bianchi, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The mit alewife machine: Architecture and performance. In *Proceedings ISCA '22*, pages 2–13, June 1995.

[2] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting Hetrogeneous Parallelism on a Multithreaded Multiprocessor. In *Proceedings of 1992 International Conference on Supercomputing*, July 1992.

[3] B. S. Ang et al. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. In *Advanced Topics in Dataflow Computing and Multithreading, IEEE Press*, 1995.

[4] B. K. Bershad, B. J. Chen, D. Lee, and T. H. Romer. Avoiding conflict misses dynamically in large direct-mapped caches. In *ASPLOS VI*, 1994.

[5] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885, 2001.

[6] 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^{th} *Symposium on Operating Systems Design and Implementation (OSDI'00)*, Oct. 2000.

[7] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin-Madison Computer Science Department, 1997.

[8] 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.

[9] J. P. Casmira and D. R. Kaeli. Modeling Cache Pollution. In *Proceedings of the* 2^{nd} *IASTED Conference on Modeling and Simulation*, pages 123–126, 1995.

[10] D. T. Chiou. *Extending the Reach of Microprocessors: Column and Curious Caching*. PhD thesis, Massachusetts Institute of Technology, 1999.

[11] Compaq. Compaq alphastation family.

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

[13] S. O. Hobbs, J. S. Pieper, and S. C. Root. Bandwidth-Based Prefetching for Constant-Stride Arrays. In *Proceedings of the Workshop on Memory Performance Issues, Goteborg, Sweden*, July 2001.

[14] Intel. *Intel IA-64 Architecture Software Developer's Manual: Volume 3: Instruction Set Reference, revision 1.1*, July 2000.

[15] F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the* 16^{th} *Symposium on Operating System Principles*, Oct. 1997.

[16] A. C. Klaiber and H. M. Levy. An Architecture for Software-controlled Data Prefetching. *SIGARCH Computer Architecture News*, 19(3):43–53, May 1991.

[17] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software Prefetching in Pointer- and Call-intensive Environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, pages 231–236, November 1995.

[18] C.-K. Luk and T. C. Mowry. Compiler-based Prefetching for Recursive Data Structures. *ACM SIGOPS Operating Systems Review*, 30(5):222–233, 1996.

[19] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Usenix Annual Technical Conference, New Orleans, Lousiana*, pages 101 – 116, June 1998.

[20] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[21] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Assoc. Mach Workshop*, pages 17–29, 1990.

[22] R. V. Meter and M. Gao. Latency Management in Storage Systems. In *Proceedings of the* 4^{th} *Symposium on Operating Systems Design and Implementation (OSDI'00)*, Oct. 2000.

[23] 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^{nd} *Symposium on Operating Systems Design and Implementation (OSDI'96)*, Oct. 1996.

[24] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, MA, October 1992.

[25] B. Nayfeh and Y. A. Khalidi. Us patent 5584014: Apparatus and method to preserve data in a set associative memory device, Dec. 1996.

[26] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache Miss Heuristics and Preloading Techniques for General-purpose Programs. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, November 1995.

[27] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monograph in Parallel and Distributed Computing. MIT Press, 1992.

[28] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of* 15^{th} *Symposium on Operating System Principles*, pages 79–95, Dec. 1995.

[29] 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^{nd} *ASPLOS*, Oct. 1987.

[30] T. Sherwood, B. Calder, and J. Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of the International Conference on Supercomputing, Rhodes, Greece*, June 1999.

[31] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *Proceedings of the 1996 USENIX Techical Conference*, Jan. 1996.

[32] I. Subramanian. Managing Discardable Pages with an External Pager. In *Proceedings of the USENIX Mach Symposium*, 1991.

[33] Sun Microsystems. *UltraSparc User's Manual*, July 1997.

[34] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[35] 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.

[36] S. P. Vanderwiel and D. J. Lilja. Data Prefetch Mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.