INFORMING LOADS: ENABLING SOFTWARE TO OBSERVE AND REACT TO MEMORY BEHAVIOR

Mark Horowitz
Margaret Martonosi
Todd C. Mowry
Michael D. Smith

Technical Report No. CSL-TR-95-673
(also numbered STAN-CS-95-673)

July 1995

This research has been supported by ARPA contract DABT63-94-C-0054.
In addition, Margaret Martonosi is supported in part by a National Science Foundation Career Award (CCR-9502516). Todd C. Mowry is supported by a Research Grant from the Natural Sciences and Engineering Research Council of Canada. Michael D. Smith is supported by the National Science Foundation under a Young Investigator Grant No. CCR-9457779.

INFORMING LOADS: ENABLING SOFTWARE TO OBSERVE AND REACT TO MEMORY BEHAVIOR

Mark Horowitz
Computer Systems Laboratory
Stanford University

Margaret Martonosi
Department of Electrical Engineering
Princeton University

Todd C. Mowry
Department of Electrical and Computer Engineering
University of Toronto

Michael D. Smith
Division of Applied Sciences
Harvard University

Technical Report: CSL-TR-95-673
(also numbered STAN-CS-95-673)

July 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Memory latency is an important bottleneck in system performance that cannot be adequately solved by hardware alone. Several promising software techniques have been shown to address this problem successfully in specific situations. However, the generality of these software approaches has been limited because current architectures do not provide a fine-grained, low-overhead mechanism to observe memory behavior directly. To fill this need, we propose a new set of memory operations called *informing* memory operations, and in particular, we describe the design and functionality of an *informing load instruction*. This instruction serves as a primitive that allows the software to observe cache misses and to act upon this information inexpensively (i.e. under the miss, when the processor would typically be idle) within the current software context.

Informing loads enable new solutions to several important software problems. We demonstrate this through examples that show their usefulness in (i) the collection of fine-grained memory profiles with high precision and low overhead and (ii) the automatic improvement of memory system performance through compiler techniques that take advantage of cache-miss information. Overall, we find that the apparent benefit of an informing load instruction is quite high, while the hardware cost of this functionality is quite modest. In fact, the bulk of the required hardware support is already present in today's high-performance processors.

Key Words and Phrases: **Memory latency, Performance monitoring, Prefetching, Processor architecture, Cache miss notification.**

1.0 Introduction

As the gap between processor and memory speeds continues to widen, memory latency has become a dominant bottleneck in overall application execution time. In current uniprocessor machines, a reference to main memory takes on the order of 50 processor cycles [DEK+92], while in shared-memory multiprocessors, latencies to remote memory can be three to four times larger [KOH+94]. In the future, these latencies are expected to increase even further [CWN92]. To cope with memory latency, most computer systems today rely on their cache hierarchy to reduce the effective memory access time. While caches are an important step toward addressing this problem, neither they nor other purely hardware-based mechanisms (e.g., stream buffers [Jou90]) are complete solutions.

In addition to hardware mechanisms, a number of software techniques have been proposed for avoiding or tolerating memory latency. For example, performance monitoring tools attempt to measure where memory bottlenecks lie and give indications of how programmers could fix them. Automatic compiler transformations attempt to generate code with improved locality and better latency tolerance. Operating systems attempt to adjust page coloring and migration strategies in response to memory referencing behavior. While these techniques are successful in many cases, they are handicapped by the fact that *software cannot directly observe the behavior of the memory system*. Because of this basic limitation, monitoring tools are forced either to measure memory overhead indirectly (often at a coarser granularity than desirable) [GH93] or to rely on simulated data about memory system behavior [Mar93, LW94]; compiler optimizations are driven by static guesses rather than dynamic observations of caching behavior [Mow94]; and for some page migration and mapping optimizations, operating systems are turning to specialized add-on hardware support such as bus-based hardware monitors [CDV+94] and cache miss lookaside buffers[BLRC94].

The fundamental problem is that load instructions were defined when memory hierarchies were flat, and memory latency was not a prime concern. The model they present is one of a uniform high-speed memory. Given that the memory system has become such an important bottleneck, it is time to re-examine the semantics of memory instructions. We propose a new set of memory operations, *informing memory operations*, which serve as primitives that allow software to observe cache misses directly, and to act upon this knowledge inexpensively within the current software context. An informing load instruction, for example, gives software the ability to react differently to the unexpected case of the load *missing* in the primary cache and execute this code under the miss, when the processor would normally stall. Note that informing loads represent new load instructions with added functionality—they are not intended to supplant any existing types of load instructions.[1] In particular, informing loads do not replace the need for prefetch instructions, which are inserted by the compiler when it believes that a subsequent load operation is likely to miss in the cache. Although we concentrate on loads in this paper, one could also define other “informing” memory operations, such as *informing stores*, which indicate when the store misses in the cache, or *informing prefetches* which software could use to detect the unexpected (and undesirable) case of a prefetch *hitting* in the cache.

While there are potentially many different types of informing memory operations and many ways to implement their functionality in hardware, all share a number of common characteristics. Section 2.0 presents these characteristics in the context of an informing load instruction, and it contrasts our approach with existing approaches for observing and reacting to memory behavior. Section 3.0 focuses on how the *observability* of misses makes it feasible and inexpensive to collect a wide variety of memory profiling information. This information is not only useful to the pro-

1. When software does not care whether a particular load hits or misses, it is free to use regular (rather than informing) loads.

grammer, but can also be used by the compiler or the operating system to automatically improve performance, as we demonstrate in Section 4.0. Section 4.0 also shows that the informing load primitive can be used not only to update miss statistics, but also to take *active measures* to improve memory performance under a miss, such as prefetching the next data item expected to miss. Having demonstrated the utility of this mechanism, Section 5.0 then describes a number of implementations of informing loads and shows that the added implementation cost is modest. Section 6.0 summarizes the findings of this work, and describes our continuing efforts in this area.

2.0 Architectural Characteristics of Informing Loads

While the computing industry largely agrees on the critical importance of the growing processor-memory performance gap, it has not yet reached a consensus on the type or degree of hardware support appropriate for monitoring memory behavior. Overall, a number of disjoint and specialized solutions have been proposed for different parts of the problem. For example, on-chip hardware miss counters have been implemented to offer at least minimal support to compilers and performance monitors [DEC92], while off-chip hardware like the cache miss lookaside buffer has been proposed to support operating systems activities like page coloring [BLRC94]. Neither of these approaches is general or efficient enough to support the needs of both the compiler and the operating system. Furthermore, the existing hodgepodge of approaches contains solutions that are often either heavyweight (i.e. access to the monitoring information greatly disrupts the behavior of the monitored program) or coarse-grained (i.e. the monitoring information is only a summary of the actual memory system behavior). These characteristics are inappropriate for software techniques requiring on-the-fly, high-precision observation and reaction to memory system behavior; instead, a fine-grained, lightweight mechanism is needed. These reasons, along with the fact that our work spans a number of disciplines and computer architectures, motivated us to define a memory system monitoring mechanism with the following characteristics:

- *general:* independent of a particular hardware organization;
- *fine-grained:* enables low-level observation of the memory system;
- *selective notification:* invoked only when triggering action occurs;
- *low overhead:* introduces very little perturbation in the monitored program when not invoked;
- *instantaneous notification*: supports fast response to triggering action;
- *primitive:* provides only notification of triggering action.

Our proposal to meet these goals is an informing load instruction—a non-blocking load instruction that is capable of squashing (i.e. inhibiting) the execution of the instruction that immediately follows the informing load in the sequential program. (In a single-issue RISC machine, this would be the instruction in the delay slot of the informing load instruction.) In some sense, an informing load is akin to the squashing branches in the SPARC architecture [Pau94] or the nullifying operations in the HP PA-RISC architecture [HP92]. An informing load operation *squashes* its delay slot instruction if the load *hits* in the cache. Informing loads are non-blocking loads so that the software can invoke an alternate action during the processing of an informing load that misses in the cache. In other words, the instruction following an informing load is only executed when the informing load misses in the cache. This functionality allows a single instruction (perhaps a speculative prefetch of a related load that may also miss) to be inlined with the informing load. We can also implement more complex instruction sequences by filling the informing load slot with a jump to other code, such as a monitoring routine or a starting point for alternatively scheduled code.

With this functionality, informing loads exhibit a lower overhead than other existing approaches for monitoring the memory system. The major difficulty in observing and reacting to memory behavior has always been that memory references are interspersed throughout the code at a very fine granularity. Instrumentation techniques that attempt to trap or read privileged timers on every memory reference clearly fail; both the execution time overhead and the cache perturbation of these techniques are prohibitive. More recently, machines have offered high-resolution timers to programmers. While these timers support loop-level monitoring of code, as in Mtool [GH93], they still have too high an overhead to be used to time and record individual memory reference latencies. Another attempt to address this problem is the inclusion of various forms of hardware cache-miss counters. For example, the Alpha 21064 includes a performance counter that can be configured to cause an interrupt after either 256 or 4096 cache misses have occurred [DEC92]. However, because of the overhead and cache perturbation of handling a full interrupt, these techniques are least intrusive with the larger count. Even if processor architects were willing to define a miss counter as part of their architecture (so it could be read at user level) the overhead in reading the counter values before and after the load and doing the comparison would be prohibitive for monitoring every load. As we quantify in the next section, our low overhead notification allows such fine-grained memory performance monitoring. This finer granularity then opens up new possible uses for memory performance information, such as the adaptive prefetching discussed in Section 4.0.

The low overhead of an informing load instruction is coupled with an instantaneous notification to the *currently executing thread*. (Other latency-hiding techniques often force a light-weight context switch on a cache miss [LGH94].) Of course, the notification of a cache miss during an informing load could be used to implement a light-weight context switch functionality. In addition to light-weight context switches, informing loads support a wide variety of application-level responses to a cache miss. For example, one could speculatively execute additional code on cache misses, dynamically control the degree of parallelism (by running code to increase or reduce the number of parallel application threads), or even execute an alternative instruction schedule that was statically optimized for the case of a cache miss. In a broader sense, if the code executed in response to an informing load miss has access to the referencing application's context, we are not simply hiding machine latency by executing an independent (and likely unrelated) thread. Rather, individual users can get benefits from informing loads, even on sequential code.

Overall, an informing load instruction is a primitive that supports a variety of proposed latency tolerance and avoidance techniques in a unified way. A preliminary list would include: (i) program performance monitoring, (ii) software-controlled prefetching, (iii) multithreading, (iv) speculative execution, (v) operating system page coloring and migration, and (vi) dynamic instruction rescheduling. In the following two sections, we examine and evaluate implementations of several of these techniques using informing loads.

3.0 Monitoring Program Performance

A number of performance tools have been proposed to monitor program caching behavior [BM89, GH93, LW94, Mar93]. One of the major stumbling-blocks in building such tools is gathering appropriately detailed memory statistics with low runtime overheads and minimal perturbations of the monitored program. For example, Mtool gathers memory statistics for loop nests by comparing basic block execution times from program runs with estimates of execution times based on ideal memory behavior. In this way, it is able to use techniques based on program-counter sampling to gather program memory statistics. The main drawback to approaches like Mtool is that statistics at a loop or basic block granularity are often too coarse-grained to be useful in understanding program memory bottlenecks. For

example, blocked matrix multiply codes access three matrices within their main loop nest. Of these three matrices, it is the blocked matrix that is most susceptible to poor memory performance due to conflict misses [LRW91]. Loop-level statistics will report this as a problem with the entire loop, rather than pinpointing the bottleneck to a particular data structure or reference point.

For finer-grained memory statistics, other tools rely on dedicated hardware to monitor memory references. Burkhart, et al. [BM89] and others have implemented tools based on data collected by special hardware bus monitors. These approaches are increasingly difficult due to the levels of integration in modern processors. With first-level and perhaps second-level caches on-chip, cache performance monitoring warrants integrated processor support.

Some techniques monitor memory behavior by trapping on particular accesses and simulating them [RHL+93], or by trapping based on values in sampled hardware miss counters [DEC92]. Such trap-based techniques incur overheads of 200 cycles or more on each trapped event just to get to monitoring code. In addition, the cache perturbation due to trap handling can be significant. In contrast, informing loads will have "trap" costs of 0 to 5 cycles, depending on their implementation. This low overhead greatly increases flexibility in monitoring style. Furthermore, from both philosophical and efficiency standpoints, there is no clear justification for requiring applications to use operating system services to monitor their own performance.

Because of the drawbacks of hardware-based and trap-based monitoring, tools such as MemSpy [Mar93] and CProf [LW94] are based on direct-execution simulation. These approaches require no dedicated hardware, but unfortunately even streamlined implementations of simple simulators impose slowdown factors of three to five on application execution time. While these overheads may be acceptable when there are no alternatives for gathering the needed data, there is an unavoidable tradeoff between the accuracy at which the memory system can be simulated and the tool's runtime overhead. A final significant drawback is that for *multiprocessors*, the overheads of simulation-based approaches scale almost linearly with the number of simulated processors, due to the fine-grained synchronization necessary for parallel simulation.

Informing loads are a primitive for capturing program memory performance information; they avoid these high simulation overheads, and also have good scaling behavior for monitoring parallel programs. They are integrated into the processor design, avoiding the black-box problem faced by dedicated hardware bus monitors with today's highly integrated processors. Finally, their low overhead allows the development of *fine-grained* monitoring tools.

3.1 A Memory Performance Monitor Based on Informing Loads

Because informing loads are a general primitive, one can imagine using them to implement many different tools. These tools can range from extremely inexpensive techniques such as program miss counts using sampling to extremely detailed techniques including high-level program semantics (e.g. correlating misses with surrounding loop iterations or data addresses). To demonstrate the utility of informing loads for performance monitoring, we implemented a fairly simple, but low overhead, memory performance tool that gives precise counts of cache misses for different reference points in the code. For all program loads that are not accessing memory off the stack pointer, an informing load is used.[2] In the informing load delay slot, we place a procedure call (a jump and link) to the monitor-

2. Although we omit stack frame references since they generally hit in the cache, one could clearly monitor them as well.

ing code. If the load turns out to be a cache hit, this procedure call is squashed, and application execution continues. If the load turns out to be a cache miss, the procedure call (to a short sequence of monitoring code) is executed.

The action taken on a monitoring call-out depends on the degree of sampling. With no sampling, every informing load miss results in a hash table lookup and a subsequent increment of the appropriate counter. Each application program counter value that generates an informing load miss has a separate count. That is, statistics are kept per static reference point in the code. Informing loads let monitoring code execute directly in the application's context, and this tool takes advantage of that fact—the return address register within the monitoring procedure uniquely identifies the reference that suffered the miss, and we use this value to index a hash table. For efficiency and for access to this return address register, monitoring code is written in assembly language.

With sampling, every Nth miss results in these actions—the other misses simply decrement a sampling counter. We show results for N equal to 1 (no sampling), 100, and 1000. Although other memory monitoring approaches have also used sampling, they are restricted in their choice of sampling frequency either to reduce execution time overhead or to improve accuracy. Trap-based techniques often monitor very infrequently in order to reduce overhead and cache perturbation. This limits their ability to keep fine-grained statistics. On the other hand, sampling with simulation-based techniques must overcome a cold-start effect at the beginning of each sample; thus large chunks of memory references must be simulated in each sample. Because informing loads are both low-overhead and low-perturbation, we have flexibility to choose sampling frequency from a much broader range of values.

3.2 Results

We have run our tool on a collection of SPEC and other applications [Dix92,Smi92]. For each, we collected the per-static-reference data as described above. In the sections below, we present information on the execution time overhead of the informing-load-based tool. We also quantify data cache perturbation induced by running monitoring code interspersed with the application code.

The results presented here, and in later sections presenting simulation results, were collected using a simulator based on *pixie* [Smi91]. We model a single-issue processor with split, direct-mapped primary instruction and data caches (each 8KB), and a 256KB, direct-mapped, unified secondary cache. A primary miss satisfied by the secondary cache takes 12 cycles, and a primary miss going all the way to memory takes a total of 75 cycles. Floating point stalls are not modelled. This execution-time overhead we measure assumes the essentially worst-case behavior of the simple, single-issue pipeline described in Section 5.1. In superscalar machines where there are typically free instruction fetch and execute positions available, the overheads presented here would decrease even further.

3.2.1 Initial Tool Overhead

Figure 3.1 shows the normalized execution times when using the memory tool to monitor several applications. The bars are normalized to the execution time of unmonitored code (which would be 100%). The three columns give tool overheads for different levels of monitoring and sampling. Execution time overheads for the unsampled (N=1) implementation range from 2% for grep to 355% for espresso. The wide range in overheads stems from variance in application reference rate, cache miss rate, and the degree that the monitoring code perturbs application caching behavior. With the N=100 sampling implementation, overheads drop to a range from 0.6% for grep to 25% for compress.

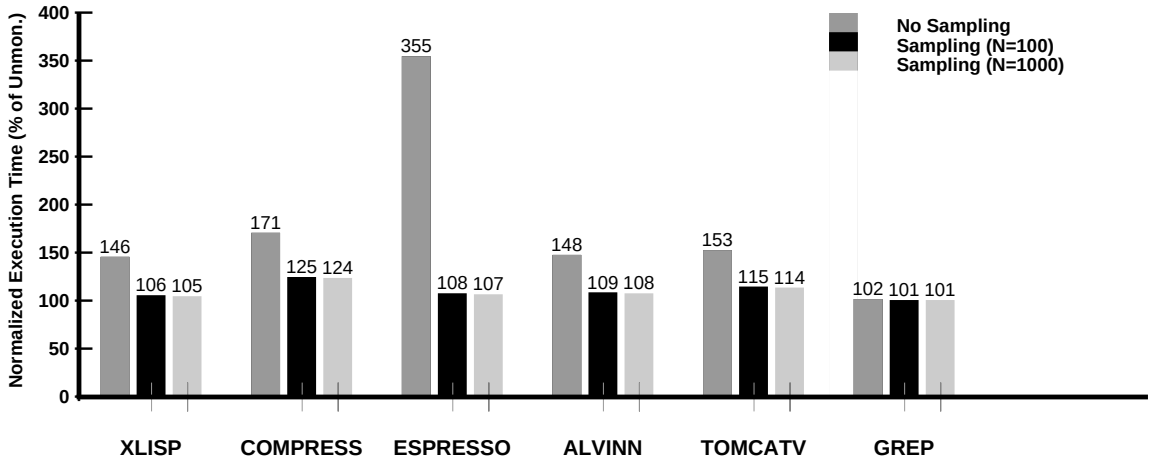


FIGURE 3.1. Normalized execution times for monitored applications with different levels of sampling. The "No Sampling" bar represents the monitoring of every informing load miss. The "N=100" and "N=1000" cases keep statistics on only every hundredth and every thousandth informing load miss, respectively.

For comparison, Mtool produces much less detailed statistics, but in spite of this still reports overheads in the range of 3% to 15% (for a subset of these applications) [GH93]. MemSpy uses a simulation-based approach to collect detailed memory statistics, and its execution time overheads typically range from 700% to 1600%.[3] Overall, performance overheads for an informing-load-based tool are competitive with high-level tools offering much less detailed statistics, and they are superior to simulation-based tools offering similar levels of detail in their statistics. Most importantly, we generate statistics based on *true executions of the program*—not cache simulations. Our statistics reflect the impact of operating system references and multiprogramming on program cache behavior.

3.2.2 Data Cache Perturbations

Because we are monitoring a real program execution, our tool can potentially perturb program behavior. Data cache effects can have a two-fold impact. First, monitoring can potentially perturb the application data cache behavior enough that the measured memory statistics do not closely correspond to the memory behavior of the unmonitored application. Second, if monitoring substantially increases the data cache miss rate, it may slow down the application execution, which would appear as another form of tool runtime overhead.

To indicate the impact of data cache perturbation, Figure 3.2 compares the application data cache behavior for monitored and unmonitored code. For each application, the leftmost bar gives the data load miss rate without monitoring. The next three bars show the *application* data load miss rate with monitoring occurring at different sampling frequencies. For these data, we *simulate* the *combined* cache effects of both the monitoring and application code, but

3. MemSpy keeps data-oriented statistics, but approximately 50% to 90% of MemSpy's execution time is spent in cache simulation and lightweight context switching, so overheads do not drop significantly if data oriented statistics are omitted. There is also a version of MemSpy that, like the tool described here, uses sampling to reduce overheads. Simulating one-tenth of the total reference stream in 20 to 30 large samples, overheads are 250% to 900%. Because MemSpy is a simulation-based approach, it has to overcome a cold-cache effect as part of trace sampling; as such, the aggressive sampling feasible with informing loads (1/100 or 1/1000 of total misses) leads to intolerable inaccuracy within MemSpy.

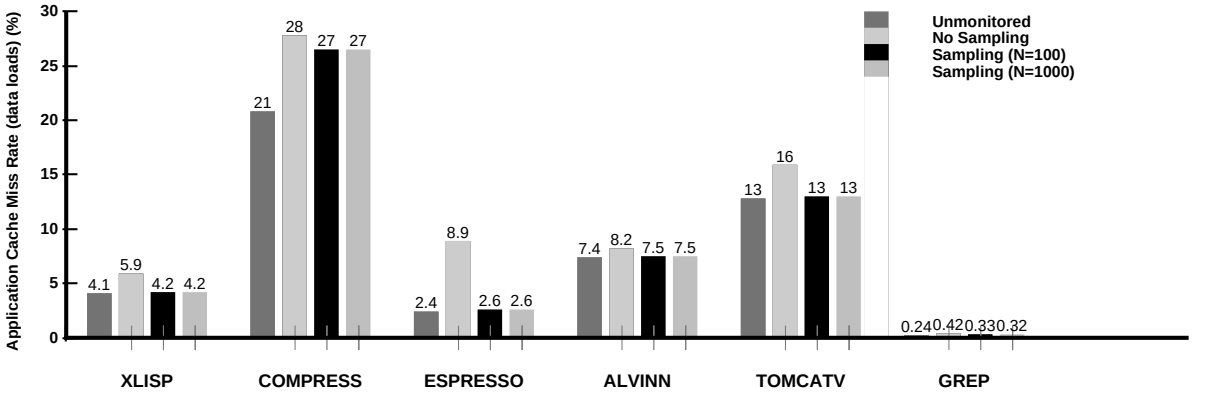


FIGURE 3.2. Effect of monitoring on application cache miss rates.

keep *separate statistics* on the misses incurred on behalf of the application. This isolates the effect monitoring has on application caching behavior. For very frequent performance monitoring, the data indicate that performance monitoring can indeed impact data cache behavior. For the N=1 case, data load miss rates are elevated by 10% to 277%. With even moderate sampling (N=100) though, cache perturbation drops dramatically.

In some applications like compress, perturbation at N=100 is still not negligible. In that particular case, perturbation remains because of frequent conflicts between an important program variable, and the location used to save away one application register before performing a sampling count check. To avoid these sorts of perturbation, we also implemented a version of the tool using uncached loads and stores for all memory references performed by the monitoring code. Since monitoring references are not cached, they never directly perturb application data cache behavior. The version with uncached loads allows virtually perfect reproduction of the application's original memory behavior. Application memory behavior perturbations will be due to secondary effects. For example, time dilation due to monitoring may mean that the application undergoes more context switches than it otherwise would have.

For the version with uncached references, execution overhead is higher than for the tool using cached references, ranging from 13% for grep to 700% for compress. Relative to the cached-references tool, overhead has increased by factors of 17 to 60. Despite this overhead increase, the uncached reference version is useful for applications where cached monitoring code causes significant perturbation. Users could opt for this tool only when they notice a discrepancy between memory behavior predicted by the cached-references version and a coarser-grained (loop-level), lower perturbation tool also implemented with informing loads. Finally, our studies use direct-mapped caches at both the first and second levels of the memory hierarchy; two or four-way set associativity should decrease cache conflicts between application and monitoring code, and expand the set of applications amenable to the faster cached-reference monitoring. For parallel applications, monitoring with uncached loads makes it easier to account for monitoring time; at synchronization points, tools can correct for monitoring time and ensure that dynamic task assignments are identical for monitored and unmonitored code.

The tool implementations described here are case studies that highlight the strengths of informing loads. By providing a very low overhead means of observing and reacting to cache misses, informing loads give crucial support to fine-grained memory performance monitoring tools. Our initial cached-reference tool monitors individual memory references in the code with low overheads (0.6% to 25%) and tolerable data cache perturbations. The version using uncached references has higher overhead (13% to 700%) but eliminates data cache perturbation.

4.0 Improving Memory Performance Automatically

To improve an application's performance using informing loads requires us to merge the methods described in the previous section with software techniques for improving memory performance. These techniques include compiler optimization like *blocking* [ASKL79, GJMS87, MC69, WL91, GL89] and *software-controlled prefetching* [Mow94, Por89, CMCH91] and the operating system optimizations like *page coloring* [KH92, BLRC94] and *page migration* [CDV+94, LE91, CF89, BFS89]. Without informing loads, the success of these automatic techniques depend heavily on how well the compiler can predict caching behavior ahead of time. Unfortunately, predicting dynamic caching behavior with only static information is quite difficult, and appears to be tractable only for array-based scientific codes which have regular and predictable access patterns. Even for these regular codes, complications such as unknown loop bounds and set-associative caches make it difficult to model caching behavior accurately. Therefore we expect the incorporation of dynamic information into the decision-making process of these techniques to be an important step toward overcoming their current limitations.

Dynamic information can be fed back into the optimization process in two ways: (i) *between* runs of a program, whereby we might recompile for a second run based on the behavior of the first run, or (ii) *during* the run of a program, whereby the code is able to monitor and adapt to dynamic information "on the fly." An advantage of the latter approach is that even the first run of a program can benefit from dynamic information—a potential disadvantage is the runtime overhead of processing and reacting to the dynamic information. In the following two subsections, we will use software-controlled prefetching as an example to demonstrate how both of these approaches can be implemented using informing loads.

4.1 Using Dynamic Memory Information at Compile-Time

The idea of using dynamic information at compile-time is not new. Compilers have historically used *control-flow* feedback (also known as "branch profiling") to perform aggressive instruction scheduling across branches [Fis81, Smi92]. Given that informing loads make it practical to collect accurate per-reference miss rates across entire applications (as demonstrated earlier in Section 3.0), a similar feedback methodology can be used to enhance aggressive memory optimizations, such as software-controlled prefetching.

Previous studies have demonstrated that for codes with regular access patterns, compiler-inserted prefetching can effectively hide memory latency, thus improving overall execution time by as much as twofold on both uniprocessor and multiprocessor systems [MLG92,Mow94]. A key step in the compiler algorithm is using *locality analysis* to predict which dynamic references are likely to suffer cache misses, and therefore should be prefetched. While locality analysis is helpful in reducing prefetch overhead, its scope is limited to affine array references, and its accuracy is limited by the abstract nature of the model. Therefore to enhance the predictions of locality analysis for regular access

patterns and to have *any* prediction of whether irregular accesses hit or miss in the cache, we would like to exploit memory feedback information.

Although precise per-reference miss rates may sound like perfect information, a subtle issue is how to handle references with intermediate miss rates (between 0% and 100%). Ideally we would like to prefetch such references only when they miss, but unfortunately the information relating individual misses to when they occur is lost in the course of summarizing them as a single miss rate. Perhaps the simplest approach is to prefetch such references either all the time or not at all, depending on whether their contribution to total misses exceeds a certain threshold.[4] More sophisticated approaches would involve reasoning about when the misses were likely to have occurred. For regular access patterns, a combination of locality analysis and control-flow feedback may be helpful. For example, consider the code in Figure 4.1(a). Assume that each element of `a` is 8 bytes, a cache line contains 32 bytes, the primary cache size is 8 Kbytes, and that memory feedback tells us that the load of `a[j]` suffered an 8.3% miss rate. From locality analysis, we would expect `a[j]` to have spatial locality along the inner loop, and possibly temporal locality along the outer loop, but that would depend on whether `m` was large relative to the cache size. If control-flow feedback indicates that the `i` and `j` loops had average trip counts of three and 100 iterations, respectively, then we would expect `a[j]` to miss only on the first of the three `i` iterations and only on every fourth `j` iteration, thus explaining the 8.3% miss rate. Therefore we could isolate these misses by peeling off the first iteration of `i` and unrolling `j` by a factor of four.

(a)
```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
      a[j] = a[j] + foo(i);
```

(b)
```
if (x > 0) {
    for (i = 0; i < n; i++)
      a[i] = foo(i);
}
x = *p;
```

FIGURE 4.1. Examples of references that may suffer misses only occasionally.

While locality analysis and control-flow feedback may shed some light on when misses occur, they cannot recognize all regular access patterns, and they do not address irregular access patterns (which are beyond the scope of locality analysis). For example, the 8.3% miss rate of `a[j]` in Figure 4.1(a) may correspond to at least two different miss patterns. One possibility is the combination of temporal and spatial locality described above. However, another possibility is that the `a[j]` locations were already in the cache when the loop nest was entered, and the misses occurred sporadically across *all* iterations due to occasional conflicts with other references in `foo()`.

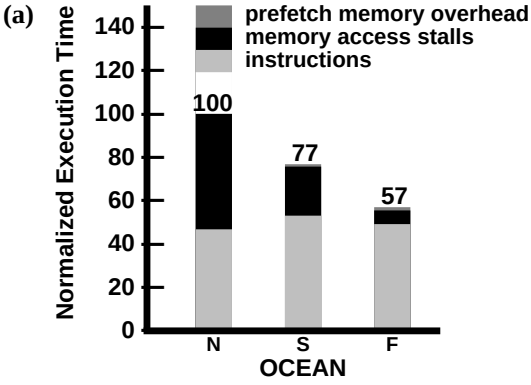
To further improve the information content of the memory profile, we would like to correlate the misses with the "dynamic context" in which they occur. For array-based codes, a useful dynamic context would distinguish the first loop iteration from the remaining iterations (to capture temporal locality), and also the loop iteration modulo the cache line size (to capture spatial locality). For example, if `a[j]` in Figure 4.1(a) had the combination of temporal and spatial locality we described earlier, we would notice that all misses occurred on the first iteration of `i` and on every fourth iteration of `j`. For the sporadic miss pattern due to conflicts, we would notice that the misses were scattered across all iterations. For irregular access patterns, such as the dereference of pointer `p` in Figure 4.1(b), the dynamic context might consist of paths in the control-flow graph that arrive at that point. For example, if we discover

4. Note that miss rate alone can be a misleading metric, since a reference can have a relatively low miss rate but still cause the majority of misses in an application if it is frequently executed.

that dereferencing `p` results in a 10% miss rate, but that these misses correspond directly to the "then" part of the "`x > 0`" conditional statement being taken 10% of the time, we can schedule the prefetch only along the "then" path, thus minimizing instruction overhead. The properties of informing loads—i.e. flexibility, low overhead, and complete access to the current software context—make it feasible to collect a profile where misses are correlated with their dynamic contexts.

Having discussed a range of possible implementations, we now present experimental results to demonstrate the performance benefits of exploiting dynamic memory information at compile-time. For these experiments, we focus on regular array-based codes. Informing loads are used to collect the miss rates of all load references (similar to the monitoring code described in Section 3.1), but misses are not correlated with *when* they occur. We then augment the compiler algorithm presented in [MLG92] to use these miss rates as follows. After performing locality analysis, the *predicted* and *observed* miss rates are compared for each reference. If they disagree beyond a certain margin, the locality analysis model is adjusted taking factors such as uncertainty and control-flow feedback into account to find an explanation for the miss rate that is consistent with the intrinsic data reuse (for further details, see [*Mow94*]). This allows the compiler to reason about intermediate miss rates and schedule prefetches only for the dynamic instances that are expected to miss.

We simulated the same array-based scientific codes presented in an earlier prefetching study [MLG92] using the same architectural assumptions. One of the cases improved significantly using memory feedback: OCEAN, which is a uniprocessor version of a SPLASH application [SWG91]. Figure 4.2(a) shows the performance of OCEAN, which has been broken down into three categories: time spent executing instructions (including the instruction overhead of prefetching), stall time due to data misses, and stall time due to memory overheads caused by prefetching (which is primarily contention for the primary cache tags during prefetch fills).



(b)

```
for j = 2 to jm-1
    laplacalc(j);


laplacalc(int col){
  ...
  for i = 2 to im-1
    z(col,i) = fact*(x(col,i+1) + x(col,i-1)
                + x(col+1,i) + x(col-1,i)
                - 4*x(col,i));
}
```

FIGURE 4.2. Performance of OCEAN using memory feedback at compile-time. (N = no prefetching, S = prefetching with static analysis only, F = prefetching with feedback).

The overall performance of OCEAN speeds up by 35% when memory feedback is used rather than using static information alone. The reason why static analysis fails in this case is because the critical loop nest is split across separate files, with the outer loop in one file, and the inner loop inside a procedure call in another file. (Figure 4.2(b) shows a simplified version of this scenario.) Since our version of the SUIF compiler [TWL+91] does not perform interprocedural analysis across separate files, the prefetching algorithm does not recognize the group locality due to the outer loop, and therefore issues too many prefetches. Once feedback information is available, the compiler immediately recognizes the group locality, thus avoiding the unnecessary prefetches. Interestingly enough, eliminating

prefetches actually reduces the memory stall time in this case by eliminating register spilling, since the spilled references were often conflicting with other data references.

OCEAN illustrates that even codes with regular access patterns (where we would normally expect static analysis to perform well) can benefit from using dynamic information at compile-time. Our experience with compiling other array-based codes indicates that reasoning about intermediate miss rates is the most challenging part of using memory feedback, and that greater gains could be achieved if informing loads were fully exploited to correlate misses with their dynamic contexts. Finally, we would expect that irregular codes would show even greater benefit from memory feedback, since there is currently no viable means of predicting misses with only static information for such codes.

4.2 Using Dynamic Memory Information at Run-Time

While memory feedback gives the compiler more information to reason with, it has a few shortcomings. First, the feedback process itself is a bit cumbersome, since it requires that the program be compiled twice. Second, it may be difficult (or impossible) to capture a representative dynamic profile, particularly if the behavior depends critically on whether the data set fits in the cache or if the data set size is determined only at run-time.

Rather than generating code with a fixed memory optimization strategy, another possibility is to generate code that adapts dynamically at run-time. For example, if informing loads indicate that more misses are occurring than expected, the code might adapt by issuing more prefetches. Similarly, the code might reduce the number of prefetches if it detects that many of them are hitting in the cache. Although tailoring code for every possible contingency would theoretically result in exponential code growth, the good news is that in practice there appear to be only a small number of different cases to specialize for. Intuitively, this is because the key distinction is whether or not the data set fits in the cache, which typically results in just two distinct prefetching strategies. Therefore when the compiler is uncertain, it could potentially generate both cases and choose the appropriate one to execute at run-time.

Since a potential drawback of adapting at run-time is the additional overhead of processing and reacting to the dynamic information, a key concern is how frequently the code would need to adapt its strategy. For array-based scientific codes, the regular and repetitive nature of the computation allows us to detect trends with only infrequent checks of miss counts, as we will demonstrate in Section 4.2.1. In contrast, the unpredictable nature of irregular codes means that we may want to adapt our strategy as often as every miss. While this may sound daunting, the properties of informing loads allow us to react at this fine granularity with minimal overhead—we simply schedule code that *actively* improves performance in the informing load delay slots, as we will describe later in Section 4.2.2.

4.2.1 Adapting Regular Codes Using Miss Counts

The regular access patterns of array-based codes make it easy to detect dynamic trends, since miss patterns often recur in a given pass through a loop. Therefore a reasonable way to adapt the prefetching strategy for a loop is to instrument an initial set of iterations, and let their overall behavior guide the approach to handling the remaining iterations. Figure 4.3 illustrates how this could be implemented with very little run-time overhead. Assuming that two elements of A fit in a cache line, all 1000 elements of A can potentially fit in the cache, and ten loop iterations are sufficient to hide memory latency, Figure 4.3(b) shows the code to prefetch all elements of A. However, if A was already in the cache before entering this loop, these prefetches would result in unnecessary overhead.

To hide cache miss latency without paying for unnecessary prefetches, we can modify our prefetching strategy as shown in Figure 4.3(c). Here we use *informing prefetches* (analogous to informing loads) to test whether the initial elements of A are already in the cache. If so, we discontinue prefetching—otherwise, we continue prefetching as usual. Note that by using informing prefetches rather than informing loads in this case, we are able to hide the latency of the first several iterations while testing for the presence of the data. (If hardware miss counters[5] could be directly manipulated without transferring to a general-purpose register, they would suffice for this simple example. Informing prefetches, however, are advantageous for loops with multiple array references since we can maintain separate miss counts for each reference, thus providing greater flexibility in our adaptiveness.)

We now present experimental results for the BCOPY library routine, which is used to copy a block of data from one location to another. Although BCOPY is a simple routine, it is interesting for two reasons. First, since it is a library routine, the compiler cannot make any assumptions about the input parameters and cannot analyze the call sites for locality. Second, since BCOPY is frequently executed by the operating system, improving its performance may significantly improve system performance. We rewrote BCOPY by hand to use adaptive prefetching, similar to the code in Figure 4.3(c). We used a simple workload to drive BCOPY, since we were mainly interested in testing the ends of the spectrum. The workload consisted of a loop that repeatedly called BCOPY with two distinct arrays and a given block size as arguments. Since identical block copies are performed on subsequent iterations, there can potentially be a temporal locality benefit if the blocks can fit in the 8 Kbyte cache. Figure 4.4 shows the performance of BCOPY using various block sizes and loop iteration counts.

As we see in the “500x10” case (where BCOPY is called ten times with 500 byte blocks) in Figure 4.4, the original code without prefetching suffers a significant amount of miss latency. These misses occur the first time the routine is called, since the data remains in the cache for subsequent calls. As we see from the middle bar, the code that

(a) Original Code

```
/* is A[i] already in cache? */
for (i = 0; i < 1000; i++)
    sum = sum + A[i];
```

(b) Code with Static Prefetching

```
/* prolog */
for (i = 0; i < 10; i+=2)
    prefetch(&A[i]);
/* steady state */
for (i = 0; i < 990; i+=2) {
    prefetch(&A[i+10]);
    sum = sum + A[i];
    sum = sum + A[i+1];
}
/* epilog */
for (i = 990; i < 1000; i++)
    sum = sum + A[i];
```

(c) Code with Adaptive Prefetching

```
pf_A_miss_count = 0;
/* issue the first several prefetches */
for (i = 0; i < 10; i+=2) {
    informing_prefetch(&A[i]);
    [pf_A_miss_count++;] /* if A[i] misses */
}
/* have the A[i] prefetches hit so far? */
if (pf_A_miss_count < SMALL_NUMBER) {
    /* if so, stop prefetching */
    for (i = 0; i < 1000; i++)
        sum = sum + A[i];
} else {
    /* otherwise, continue prefetching */
    for (i = 0; i < 990; i+=2) {
        prefetch(&A[i+10]);
        sum = sum + A[i];
        sum = sum + A[i+1];
    }
    for (i = 990; i < 1000; i++)
        sum = sum + A[i];
}
```

FIGURE 4.3. Example of how dynamic miss counts can be used to adapt prefetching in array-based codes.

5. Separate hardware miss counters should be maintained for *load* misses (which are undesirable) and *prefetch* misses (which are desirable over prefetch hits).

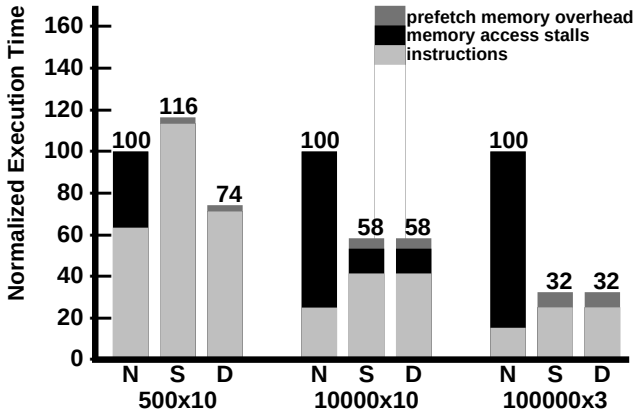


FIGURE 4.4. Results with adaptive version of BCOPY (N = no prefetching, S = statically prefetch all the time, D = adapt prefetching dynamically). "BxT" means that the same B-byte block is copied to the same destination T times. Performance is renormalized for each case.

statically prefetches the blocks all the time actually performs worse than the original case, due to the large instruction overhead of the many unnecessary prefetches. The adaptive code (shown as the righthand bar) offers the best performance, since it prefetches the data only when appropriate. The other two cases in Figure 4.4 show larger block sizes that do not fit in the cache. For these cases, stopping to check miss counts is pure overhead, since the best strategy is to prefetch all the time; however, we see that this overhead has a negligible impact on overall performance. Therefore we enjoy the benefits of adaptiveness without paying additional run-time costs.

4.2.2 Adapting Irregular Codes by Taking Active Measures Under a Miss

Irregular codes containing pointers, linked lists, etc. pose a much greater challenge to adaptive memory optimizations, since it is unclear how to detect sustained trends in the miss patterns or that such trends even exist. Therefore, in contrast to our coarse-grained approach for array-based codes (where we check and react to dynamic behavior only once in a given pass through a loop), a more fine-grained approach may be appropriate for irregular codes (where we might want to adapt our strategy as frequently as every load miss). For example, on each load miss, we may wish to launch prefetches to avoid future misses. On a multithreaded architecture, we may wish to switch threads or spawn new threads in response to a given miss. The overhead of such fine-grained reactions would normally be prohibitive, but we can hide much (if not all) of this overhead by scheduling instructions that *actively improve performance* in the

informing load delay slots. Therefore if the majority of references turn out to *hit* in the cache, the code will still execute at maximum speed since the adaptive code will not be invoked.

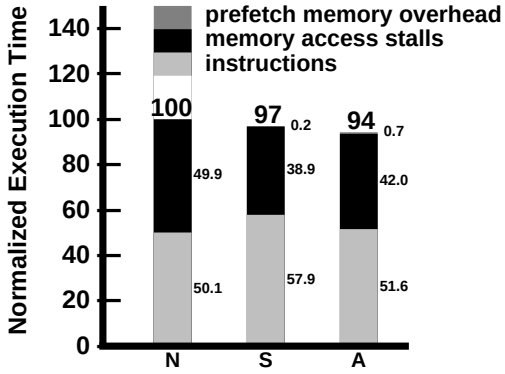


FIGURE 4.5. Performance of compress with active prefetching under a miss (N = no prefetching, S = statically prefetch all the time, A = adaptively issue prefetches only in informing load delay slots).

To experiment with this active approach, we modified the compiler developed for TORCH [Smi92] to fill informing load delay slots with a prefetch of the next likely load address whenever possible. Figure 4.5 shows the resulting performance of the compress SPEC benchmark, along with the case where those same prefetches are inserted just ahead of the same preceding loads (thus allowing misses to be pipelined). Note that for these experiments, we assume that if an informing load hits, its slot instruction does not waste any execution cycles.[6] While this is more aggressive than the base model presented in Section 5.1, it would be typical of most modern processors where instruction fetching is decoupled from execution. As we see in Figure 4.5, adaptively issuing prefetches in response to individual load misses offers twice the overall performance improvement of statically issuing those same prefetches all the time, since it results in significantly less instruction overhead.

This section has shown a number of ways that informing loads can be used to improve the performance of software-controlled prefetching. The low-overhead, selective notification of informing loads provided a building block that was used to collect the needed information with minimal performance overhead. We expect that these capabilities will also benefit other automatic memory optimizations such as cache blocking and page coloring.

5.0 Hardware Implementation

The previous sections have shown that informing loads are a useful primitive for measuring and reducing memory overhead in programs. This section describes how an informing load could be implemented in current processors. Most of the mechanisms needed for informing loads are already present in processors that support lockup-free caches and squashing branches. To demonstrate this fact, this section describes how to implement an informing load instruction in a number of different machine pipelines. We begin by describing a simple, single-issue RISC pipeline that is a simplification on the MIPS R3000 pipeline [HP90]. After reviewing this basic pipeline and machine organization, we discuss the changes needed to support informing loads. We first look at several implementations of the informing load functionality in this simple machine, and we then move our description to implementations for more complex

6. We do model all instruction cache penalties accurately. They appear as part of the “instructions” category, but happen to be negligible for this case.

machines. Although there is a spectrum of possible implementations in each case, ours are representative of the hardware and software costs of an informing load instruction.

5.1 Base Machine Model

Our base machine has a five stage pipeline: instruction fetch (IF), instruction decode and register fetch (RD), ALU execution (EX), data memory access (ME), and register write back (WB). All instructions "complete" in the WB phase of the pipeline and can be *squashed* (or turned into a null operation) by asserting a suitable signal before WB. Squashing of instructions in the pipeline is necessary to support precise exceptions [HP90]. In a normal load operation, the pipeline calculates the effective address of the load during EX and accesses the cache using this address during ME. The load data returns to the CPU before the end of the ME stage so that we can write the data to the register file during WB. Figure 5.1 illustrates this pipeline with some of the important instruction timings.

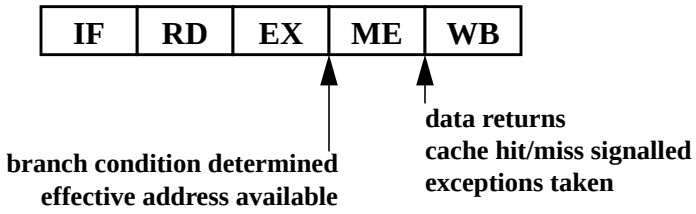


FIGURE 5.1. Pipeline of our simple machine showing when various instructions and signals occur.

If the load misses in the cache, the pipeline generally stalls until the load data is brought into the cache and returned to the processor. There are two options on when to stall the machine. The simplest option, *stall on miss*, stalls the machine when the miss is detected and holds the pipeline until the data is returned. This stall model is actually a little conservative, since if the instructions after the load don't use the load data, the processor could execute them during the processing of the cache miss. The other stall model allows the machine to continue to operate until the missing data is referenced. This option, *stall on use*, is becoming more common [D+92, Gwe94] because of its potential performance benefits. The principle cost of this alternative stall model is that the cache must be able to handle references while the miss is being serviced. This requires a *lockup-free cache*—i.e. a cache that can handle requests while a miss is outstanding. In the simplest case the cache can only handle one miss, while in more sophisticated designs the cache can handle multiple outstanding misses [Kro81,Lau94].

A machine with informing loads must execute instructions during a cache miss, and therefore the machine must stall on *use* rather than stall on miss. Changing the stall model (if necessary) is the hardest part of implementing an informing load; the stall circuitry and lockup-free cache design are complex. But since most modern machines already implement this stall model, we will focus on the other changes that are required. These changes mostly relate to how and when the instruction in the informing load delay slot is squashed.

5.2 Squashing the Slot Instruction

An informing load differs from normal squashing instructions (like squashing branches in the SPARC architecture or nullifying operations in the HP PA-RISC) because the miss signal comes late—not until the end of ME at the earliest. However for this simple machine, if the miss condition arrives before the end of ME, we can easily cancel the effects of the instruction in the informing load delay slot. In this pipeline all changes (both from branches and ALU operations) do not occur until EX of the slot instruction, which is ME of the informing load. Since we already

have the hardware to cancel the effects of this instruction due to an exception in a normal load operation, cancelling the slot instruction is not difficult.[7] Similarly, since exceptions are not signaled until the ME stage of an instruction, the instruction in the delay slot of an informing load cannot cause an extraneous exception.

In some machines that use a direct-mapped cache, the miss signal becomes valid after the data, and this means it does not settle until the beginning of WB. Since these machines start executing the WB cycle of load before a miss is indicated, they must provide some way to re-execute this cycle if a cache miss occurs. This facility is needed to handle the situation, shown in Figure 5.2, where the instruction in the EX phase is using the result of the load. Since the later instruction received the wrong value, it has to be re-executed when the correct data is returned. We can use the same mechanism to implement the informing load. Since the common case is a cache hit, we squash the load delay slot instruction by default before its ME. If the informing load misses, we re-execute that cycle to allow the slot instruction to complete its ME. The overhead of this approach is one lost cycle when a miss occurs. In many machines, this extra cycle is not even necessary since one could make the nullification decision at the end of the informing load slot's ME. A store in the informing load slot might seem like it would change state too soon under this model, but in fact, most stores take two memory cycles (i.e. tag probe then store data), so the memory state is not actually changed until WB.

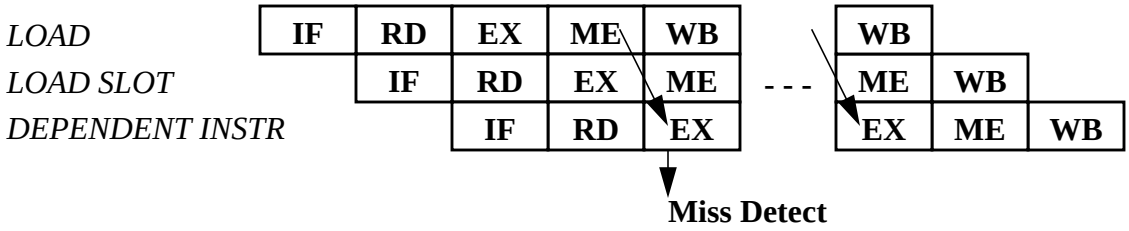


FIGURE 5.2. Pipeline diagram showing a machine with a late miss signal.

The simple pipeline in Figure 5.1 has a branch delay of two cycles. In many machines, including the R3000 and the R4600, the branch point is sooner and occurs after RD. For these machines a branch, or other control transfer instruction (CTI), is troublesome because it changes state early in the pipeline, in particular before hit/miss is determined for the informing load (see Figure 5.3). Since we do not want to update the program counter with the outcome of the branch in the informing load delay slot unless the informing load misses, we have a dilemma. The solution is similar to the one proposed for late miss signals. When a CTI is encountered in the slot, it is ignored (assumed squashed), but a bit is set indicating that the instruction in the slot was a CTI. If the informing load does miss and the slot instruction was a CTI, the instructions in the pipeline behind the informing load are all squashed, and the machine starts fetching the CTI instruction and treats it normally. The bulk of the hardware needed to support this approach (keeping the addresses of the instructions in the pipeline and squashing all the instructions currently needed for execution) is already present to support exception handling. The only additional hardware is the state bit which indicates whether the slot instruction is a branch.

7. We are ignoring subtle but important effects on performance like the small amount of extra capacitive loading on the squash signal due to the extra squash condition. Also, like all architectural mechanisms with slots, handling exceptions around an informing load requires some care to ensure that the machine always knows whether to execute the slot instruction or not. For informing loads this is easy, since if an interrupt occurs when the load completes, one can set the return PC to the correct location (slot or slot+1) depending on the status of the load.

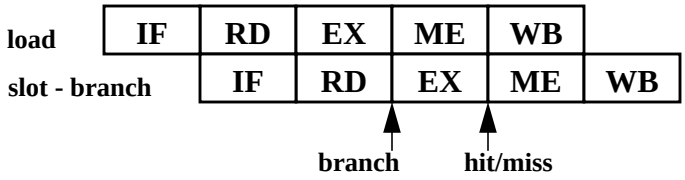


FIGURE 5.3. Diagram showing the problem in machines with early commit CTI instructions.

The cost of this simple solution is that the machine loses three cycles while the branch instruction is re-executed. This overhead only occurs when the informing load misses; this is the infrequent case and the small overhead can be tolerated since the miss has already disrupted the pipeline. Still, we could reduce this delay by the inclusion of an additional register in the PC unit of the machine. This register would store the calculated target of the CTI when it is executed. If a miss occurs, rather than restarting at the CTI instruction, the machine would load this register into the PC and restart from the target of the branch.

The situation for in-order issue superscalar machine is similar to the scalar processor. In fact our ability to deal with late miss information allows the slot instruction to be issued either *with* the informing load or in the cycle following the load. The informing-load delay-slot instruction is assumed to be squashed, and then re-executed after the informing load misses. Surprisingly, the situation for out-of-order issue machines is in some ways even simpler. These machines do dependency analysis and scheduling in hardware, so the issue of the informing load slot instruction can be held until the cache hit/miss resolution is known. At this point the instruction is either issued (on a miss) or squashed. Since the instruction is not issued to the execution unit until the miss is known, we never need to back up the pipeline when a miss occurs. The complexity in this case is forming the dependency between the informing-load slot instruction and the load, since it is different than the standard case. While this is not hard in principle, adding anything to an out-of-order machine is not trivial.

Overall, the addition of an informing load instruction to an architecture that already supports squashing branches and hit-under-miss caches is not difficult. The only complexity arises in situations where the hit/miss information is not available as early as desired. Here, the appropriate approach is to assume that the informing load will hit and optimize the pipeline actions for this common case.

5.3 Performance Overhead

Section 5.2 showed that the hardware cost to support an informing load instruction is quite small. Unfortunately, for simple single-issue pipelines, this simple implementation has a performance cost associated with it. For simple scalar machines, this cost is set by the need to fetch the slot instruction of the informing load. Since these machines can fetch and execute only one instruction per cycle, the usually-squashed slot instruction increases the instruction count of any application that uses informing loads. The cost of adding this instruction is not quite one instruction per informing load, since in some cases the load delay slot was already filled with a NOP instruction (or with a dependent instruction that causes a load interlock).

We found that the actual overhead ranged from a high of 0.99 extra instructions per informing load for a program like *compress* that has relatively few NOPs per load to a low of 0.08 extra instructions per informing load for a program like *grep* where the MIPS compiler cannot usefully fill the load delay slots. The overhead is slightly smaller

than one gets from simply counting the fill rate of load delay slots since adding an instruction after a load may cause the former load slot instruction to replace a NOP in a branch delay slot. For a collection of SPEC benchmarks and UNIX utilities [Dix92,Smi92], we found that the overhead averaged around 0.6 instructions per informing load.

On a superscalar machine however, the informing load overhead is dramatically lower on average because data dependences typically limit the average execution rate of instructions to a value below the peak rate of the machine. This means that there exists many free fetch and execute positions for the informing load slot instructions. We used an aggressive superscalar compiler that was built for TORCH [Smi92], a simple dual-issue machine with limited parallel resources. We then compared the cycle counts of the original code to code where we replaced all loads with informing loads. For these cycle counts, we assumed that the caches always hit so that the informing-load slot instructions are pure overhead. The average cost was now much lower—0.31 instructions per informing load for *compress* and 0.03 instructions per informing load for *grep*. Increasing the issue width of the superscalar machine to four and providing wide variety of parallel functional units basically reduces the overhead to zero. Thus for current superscalar machines, informing loads provide a truly low overhead method of monitoring memory system performance.

6.0 Conclusions

Many proposals have been made to provide software with feedback about memory system performance, each directed at solving a particular problem. Rather than providing specific support for one particular application, informing loads provide programmers with a general mechanism for extracting detailed information about the memory system, in particular notification that the load operation did not hit in the cache. The overhead of this general mechanism is small enough (especially in superscalar machines) that informing loads can be used throughout a program in a fine-grained fashion to collect information or react to a situation that requires a fast and extremely lightweight response. As a primitive that is easily added to any architecture, informing memory operations allow the programmer to decide what information is needed and how that information should be used.

To demonstrate the utility of an informing load instruction, this paper described how to improve the functionality of two important software techniques: performance monitoring and software-controlled prefetching. We have shown that detailed memory profiles can be captured with much less overhead (only 0.6% to 25% increase in execution time over the un-instrumented code) than other methods. We have also demonstrated that informing loads can be exploited in a variety of ways by the compiler to automatically improve the performance offered by software-controlled prefetching beyond what is possible with only static information. Clearly more work can be done in both areas. For example in performance monitoring, we could use informing loads to categorize cache misses by different data types or to develop extremely efficient tools for parallel memory performance monitoring. We also expect other automatic memory optimizations such as cache-blocking and page-coloring to benefit from informing loads.

Unlike other latency-hiding techniques that include other (potentially complex) functionality, we designed informing memory operations as a primitive. The goal was to provide memory system information back to the user in as simple a form as possible, giving each application the flexibility to choose both the information to collect and actions to take. Our preliminary results indicate that we have been successful in achieving our goals. Informing loads can provide software with information about memory system performance at a low cost. This paper has identified and evaluated several ways in which software can take advantage of informing loads. We feel that the general availability of informing loads in real hardware will spur even more innovative uses.

7.0 Acknowledgments

We thank Chi-keung Luk at the University of Toronto for his help in modifying the compiler and providing simulation results. This research has been supported in part by ARPA contract DABT63-94-C-0054. In addition, Margaret Martonosi is supported in part by a National Science Foundation Career Award (CCR-9502516). Michael D. Smith is supported by the National Science Foundation under a Young Investigator Grant No. CCR-9457779. Todd C. Mowry is supported by a Research Grant from the Natural Sciences and Engineering Research Council of Canada.

8.0 References

[ASKL79] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. Automatic program transformations for virtual memory computers. *Proc. of the 1979 National Computer Conference*, pages 969–974, June 1979.

[BFS89] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 19–31, 1989.

[BLRC94] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

[BM89] Helmar Burkhart and Roland Millen. Performance-Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[CDV+94] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–24, October 1994.

[CF89] Alan L. Cox and Robert J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 32–44, 1989.

[CMCH91] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of Microcomputing 24*, 1991.

[CWN92] Richard Comerford, George F. Watson, and Ray Ng. Special Report: Memory. *IEEE Spectrum*, 29(10):34–57, October 1992.

[DEC92] DEC. DECChip 21064 RISC Microprocessor Preliminary Data Sheet. Technical report, 1992.

[DEK+92] Todd Dutton, Daniel Eiref, Hugh Kurth, James Reisert, and Robin Stewart. The Design of the DEC 3000 AXP Systems, Two High-performance Workstations. *Digital Technical Journal*, 4(4):66–81, 1992.

[Dix92] Kaivalya M. Dixit. New CPU Benchmark Suites from SPEC. In *Proc. COMPCON*, Spring 1992.

[D+92] Dan Dobberpuhl et al. A 100MHz 64b dual-issue CMOS Microprocessor. In *International Solid State Circuits Conference Digest of Technical Papers*, pages 106–107, Feb 1992.

[Fis81] Josh Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computer*, C-30(7):478–490, July 1981.

[GH93] Aaron J. Goldberg and John L. Hennessy. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 28–40, January 1993.

[GJMS87] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical Report UIUCSRD 625, University of Illinios, 1987.

[GL89] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[Gwe94] Linley Gwennap. 620 fills out PowerPC product line. *Microprocessor Report*, 8(14):12–16, Oct 1994.

[HP90] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[HP92] HP. *PA-RISC 1.1 Architecture, Instruction Set Reference Manual*. Hewlett Packard, 1992.

[Jou90] Norm Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. 17th Annual Int'l. Symposium on Computer Architecture*, pages 364–373, May 1990.

[KH92] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-index caches. *ACM TOCS*, 10(4):338–359, 1992.

[KOH+94] Jeff Kuskin, Dave Ofelt, Mark Heinrich, et al. The Stanford FLASH Multiprocessor. In *Proc. 21st Annual Int'l. Symposium on Computer Architecture*, pages 302–313, April 1994.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Symposium on Computer Architecture*, pages 81–87, 1981.

[Lau94] James Laudon. *Architectural and Implementation Tradeoffs for Multiple-Context Processors*. PhD thesis, Stanford University, 1994.

[LE91] Richard P. Jr. LaRowe and Carla Schlatter Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[LGH94] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Sixth Int'l. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 308–318, October 1994.

[LRW91] M. S. Lam, E. E. Rothberg and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LW94] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, October 1994.

[Mar93] Margaret Martonosi. *Analyzing and Tuning Memory Performance in Sequential and Parallel Programs*. PhD thesis, Stanford University, December 1993.

[MC69] A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 27, pages 62–73, October 1992.

[Mow94] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[Pau94] Richard Paul. *SPARC Architecture, Assembly Language Programming, & C*. Prentice Hall, 1994.

[Por89] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[RHL+93] S. K. Reinhardt, M. D. Hill, J. R. Larus, et al. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*. pages 48-59, May, 1993.

[SWG91] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[Smi91] Michael D. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[Smi92] M. D. Smith. *Support for Speculative Execution in High-Performance Processors*. PhD thesis, Stanford University, November 1992.

[TWL+91] S. Tjiang, M. Wolf, M. Lam, et al. *Integrating Scalar Optimizations and Parallelization*. Springer-Verlag, pp. 137-151, August 1991.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.