Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications

MARK HOROWITZ
Stanford University
MARGARET MARTONOSI
Princeton University
TODD C. MOWRY
Carnegie Mellon University
and
MICHAEL D. SMITH
Harvard University

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 for observing and reacting to memory behavior directly. To fill this need, this article proposes a new class of memory operations called *informing memory operations*, which essentially consist of a memory operation combined (either implicitly or explicitly) with a conditional branch-and-link operation that is taken only if the reference suffers a cache miss. This article describes two different implementations of informing memory operations. One is based on a *cache-outcome condition code*, and the other is based on *low-overhead traps*. We find that modern in-order-issue and out-of-order-issue superscalar processors already contain the bulk of the necessary hardware support. We describe how a number of software-based memory optimizations can exploit informing memory operations to enhance performance, and we look at cache coherence with fine-grained access control as a

Mark Horowitz is supported by ARPA Contract DABT63-94-C-0054. Margaret Martonosi is supported in part by a National Science Foundation Career Award (CCR-9502516). Todd C. Mowry is partially supported by a Faculty Development Award from IBM. Michael D. Smith is supported by a National Science Foundation Young Investigator award (CCR-9457779).
Authors' addresses: M. Horowitz, Computer Systems Laboratory, Stanford University, Stanford, CA 94305; email: horowitz@ee.stanford.edu; M. Martonosi, Department of Electrical Engineering, Princeton University, Princeton, NJ 08544-5263; email: mrm@ee.princeton.edu; T. C. Mowry, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; email: tcm@cs.cmu.edu; M. D. Smith, Division of Engineering and Applied Sciences, Harvard University, 29 Oxford Street, Cambridge, MA 02138; email: smith@eecs.harvard.edu.

case study. Our performance results demonstrate that the runtime overhead of invoking the informing mechanism on the Alpha 21164 and MIPS R10000 processors is generally small enough to provide considerable flexibility to hardware and software designers, and that the cache coherence application has improved performance compared to other current solutions. We believe that the inclusion of informing memory operations in future processors may spur even more innovative performance optimizations.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*; C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Cache miss notification, memory latency, processor architecture

1. 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 can be 50 or more processor cycles; multiprocessor latencies are even higher. 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 [Jouppi 1990]) are complete solutions.

In addition to hardware mechanisms, a number of promising software techniques have been proposed to avoid or tolerate memory latency. These software techniques have resorted to a variety of different approaches for gathering information and reasoning about memory performance. Compiler-based techniques, such as cache blocking [Abu-Sufah et al. 1979; Gallivan et al. 1987; Wolf and Lam 1991] and prefetching [Mowry et al. 1992; Porterfield 1989] use static program analysis to predict which references are likely to suffer misses. Memory performance tools have relied on sampling or simulation-based approaches to gather memory statistics [Covington et al. 1988; Dongarra et al. 1990; Goldberg and Hennessy 1993; Lebeck and Wood 1994; Martonosi et al. 1995]. Operating systems have used coarse-grained system information to reduce latencies by adjusting page coloring and migration strategies [Bershad et al. 1994; Chandra et al. 1994]. Knowledge about memory-referencing behavior is also important for cache coherence and data access control; for example, Wisconsin's Blizzard systems implement fine-grained access control in parallel programs by instrumenting all shared data references or by modifying ECC fault handlers to detect when data are potentially shared by multiple processors [Reinhardt et al. 1994].

While solutions exist for gathering some of the needed memory performance information, they are handicapped by the fact that software cannot directly observe its own memory behavior. The fundamental problem is that load and store instructions were defined when memory hierarchies were flat, and the abstraction they present to software is one of a uniform high-speed memory. Unlike branch instructions—which observably alter control flow depending on which path is taken—loads and stores offer no direct mechanism for software to determine if a particular reference was a hit or a miss. Improving memory performance observability can lead to improvements not just in performance monitoring tools, but also in other areas, including prefetching, page mapping, and cache coherence.

With memory performance becoming more important, machine designers are providing at least limited hardware support for observing memory behavior. For example, hardware bus monitors have been used by some applications to gather statistics about references visible on an external bus [Bershad et al. 1994; Burkhart and Millen 1989; Chandra et al. 1994; Singhal and Goldberg 1994]. A bus monitor's architectural independence allows for flexible implementations, but also can result in high overhead to access the monitoring hardware. Furthermore, such monitors only observe memory behavior beyond the second- or even third-level cache. More recently, CPU designers have provided user-accessible monitoring support on microprocessor chips themselves. For example, the Pentium processor has several performance counters, including reference and cache miss counters, and the MIPS R10000 and Alpha 21064 and 21164 also include memory performance counters [Digital Equipment 1992; Edmonson et al. 1995; Heinrich 1995; Mathisen 1994]. Compared to bus monitors, on-chip counters allow finer-grained views of cache memory behavior. Unfortunately, it is still difficult to use these counters to determine if a particular reference hits or misses in the cache. For example, to determine if a particular reference is a miss (e.g., to guide prefetching or context-switching decisions), one reads the miss counter value just before and after each time that reference is executed. This is extremely slow, and in addition, counters must be carefully designed to give such fine-grained information correctly. In an out-of-order-issue machine like the MIPS R10000, one must not reorder counter accesses around loads or stores; in fact, counter accesses in the R10000 serialize the pipeline.

Overall, a number of disjoint and specialized solutions have been proposed for different problems. Existing hardware-monitoring support is 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 undesirable for software requiring on-the-fly, high-precision observation and reaction to memory system behavior. As an alternative, we propose informing memory operations: mechanisms specifically designed to help software make fine-grained observations of its own memory referencing behavior, and to act upon this knowledge inexpensively within the current software context. An informing memory operation lets software react differently to the unexpected case of the reference missing in the cache and execute handler code under the miss, when the processor would normally stall.

There are many alternative methods of achieving informing memory operations, and Section 2 describes three possible approaches: (1) a cache outcome condition code, (2) a memory operation with a slot that is squashed if the reference hits, and (3) a low-overhead cache miss trap. In Section 3 we describe implementation issues in the context of an in-order-issue and out-of-order-issue machine. The hardware cost for these mechanisms is modest, yet they provide much lower overhead and more flexible methods of obtaining information about the memory system than current counter-based approaches. As discussed in Section 4, informing memory operations enable a wide range of possible applications—from fine-grained cache miss counting that guides application prefetching decisions to more elaborate cache miss handlers that enforce cache coherence or implement context-switch-on-a-miss multithreading. Section 5 gives more detailed case studies of three such applications. Section 6 summarizes our findings.

2. INFORMING MEMORY OPERATIONS

In contrast to current methods for collecting information about the memory system, an informing memory operation can provide detailed memory system performance information to the application with very little runtime overhead. Ideally, an informing memory operation incurs no more overhead than a normal memory operation if the reference is a primary data cache hit. On a cache miss, the informing memory operation causes the processor to transfer control of the program to code specific to the memory operation that missed, thus providing a mechanism for fine-grained observation of memory system activity.

We can essentially decompose the informing memory mechanism into a memory operation and a conditional branch-and-link operation whose execution is predicated on the outcome of the memory operation's hit/miss signal. If the memory operation hits in the cache, the transfer-of-control portion is nullified. If the memory operation misses, control is transferred to the indicated target address. Overall, our work has focused mainly on three architectural methods of implementing fine-grained, low-overhead informing memory operations. Though similar in functionality, each method differs in how it transfers control after a data cache miss. In our first method, based on a cache outcome condition code bit, the conditional branch-and-link operation is an explicit instruction in the instruction stream, and we create user state that records the hit/miss status of the previously executed reference. By allowing the branch-and-link operation to test the value of this state bit, we provide the software with a mechanism to react to memory system activity. Though simple, this mechanism offers quicker control transfers than current cache miss counters.

Our second method removes the explicit user state for the hit/miss information, but retains the explicit dispatch instruction. In this case, the machine "notifies" software that the informing operation was a cache hit by squashing the instruction in the issue slot following that informing operation. If the reference is a cache miss, this slot instruction is executed. By placing a conventional branch-and-link instruction in the slot, we effectively create a branch-and-link-if-miss operation. Our experiments have shown that this second method has similar performance to the cache condition code approach, but with a slightly more complex hardware implementation. It requires two different sets of memory operations (since users rarely want all memory operations to be informing), and it suffers from hardware designers' aversions to delay slot semantics in architectures for superscalar machines. For these reasons we do not consider it further here.

Our third informing memory operation mechanism, low-overhead cache miss traps, removes both explicit user state for hit/miss information and the explicit dispatch instruction. Here, a low-overhead trap to user space is triggered on a primary data cache miss. The target of this trap and the return address are kept in special machine registers.

This remainder of this section describes the cache condition code and low-overhead cache miss trap methods in more detail. We discuss the required hardware, instruction overheads, and critical software issues for both methods. Section 3 presents implementation specifics for one of these methods in both in-order-issue and out-of-order-issue superscalar processors.

2.1 Cache Outcome Condition Code

In our first method, all memory operations become informing memory operations by default. The hardware simply records hit/miss results of each data memory operation in user-visible state, and then relies on the software to place explicit checks of this state in the program code where desired. To support this explicit check, we add a new conditional branch-and-link instruction to the instruction set. This new instruction tests the hit/miss result of the previous memory operation, and it transfers control to the encoded target address if that memory operation was a miss.

Most of the hardware needed to implement this functionality is already in the base machine, especially if the machine supports condition codes. Here, the cache miss result simply becomes another bit in the condition code register. One only needs to add hardware to store this condition code (and do the proper bypassing/renaming that is needed in a modern processor).

In terms of runtime overhead, an application fetches an extra instruction for every informing memory operation of interest—i.e., the conditional branch-and-link instruction. To minimize the cycle-count overhead of this instruction, we would want to optimize its execution for the common case, which is a data cache hit. In other words, we should predict the conditional branch-and-link instruction to be not taken. Therefore, the normal branch mispredict penalty applies only to the cache miss case.

The strength of this scheme is its simplicity. The explicit check instruction allows flexibility in several key areas. Because every memory operation is potentially informing and because we can easily specialize the action taken on any static data memory reference (through a unique target address in the hit/miss test instruction) we have a low, constant overhead per static reference of interest over the entire range of instrumentation granularity. (The overhead arises from the explicit check instruction that follows each memory operation of interest; this instruction uses a fetch slot and must be placed before another memory operation is issued.) Furthermore, we can extend this mechanism to support gathering information about any level of the memory hierarchy. This would entail the definition of new condition code bits that represent the outcome results for the other hierarchy levels.

2.2 Low-Overhead Cache Miss Traps

Alternatively, we can design an informing mechanism that removes the instruction overhead of an explicit miss check. To accomplish this, our low-overhead cache miss trap method defines the informing memory operation as a memory operation that triggers a low-overhead trap to user space on a primary data cache miss. The allure of this method is that a trapping mechanism potentially incurs no overhead for cache hits. We avoid traditional trap mechanisms that require hundreds of cycles to context switch into the operating system and invoke the trap dispatch code, which then context switches again to run the actual handler.

To reduce the overhead on a cache miss, we propose a cache miss trap that is more like a conditional branch than a conventional trap. The trap only changes the program counter of the running application; it does not invoke any operating system code and saves only a single user-visible machine register. To implement the trap, we propose adding two user-visible registers to the machine architecture. One register is the Miss Handler Address Register (MHAR), which contains the instruction address of the handler to invoke on an informing memory operation cache miss. The other register is the Miss Handler Return Register (MHRR), into which the return address is written when the trap occurs (i.e., the address of the instruction after the memory operation that missed). In addition to the registers, we define an instruction to load the MHAR and another instruction to jump to the address in the MHRR. We assume that a zero value in the MHAR disables trapping (when observing cache misses is unwanted). This implementation involves minimal hardware additions and provides sufficient functionality for the applications of the mechanism that we have considered; further experience in writing new types of miss handlers may point to additional hardware support that would be useful (e.g., a register containing the data address of the miss, separate MHARs to invoke different miss handlers upon load versus store versus prefetch misses, etc.).

There are two main issues involved in the implementation of a low-overhead cache miss trap: the wiring of the MHAR and MHRR into the existing processor datapath, and the implicit control flow change on a cache miss. The implementation of the MHAR and MHRR is quite simple. The registers are located in the execution unit (or in the PC unit or in both for speed) and operate like other "special" machine registers. The MHRR captures the next PC value and updates the PC on a low-overhead trap return using the standard branch-and-link/jump-to-register-contents hardware paths and control.

Changing control flow on data cache misses means that the machine must execute an implicit jump. This requires the machine to nullify the subsequent instructions in the pipeline and direct the fetcher to obtain a new instruction stream. This functionality is the same as a conditional-branch-and-link operation, so we use the same mechanism to implement it. Conceptually, imagine that the decode of an informing operation inserts two instructions into the pipeline: (1) the memory operation to the memory unit and (2) a branch-and-link instruction that is predicated on the outcome of the hit/miss result of the memory operation to the branch unit. The target address for the branch-and-link is generated from the MHAR; the branch-and-link destination is the MHRR; and the predicate is assumed to be false (i.e., no dispatch) for an in-order-issue machine. (We present a detailed description of the hardware implementation later in Section 3.)

The software overheads of this method depend on how it is used. For collecting aggregate data about the memory system, a single or small number of handlers is sufficient. If the handler address does not need to be changed very often, we can achieve our goal of no overhead for cache hits. When more detailed information is required, a user can choose between a range of options. Users could set the MHAR before each memory operation, invoking a separate handler for each miss, and incurring the one instruction of overhead per memory operation. Alternatively, one could create a single handler that uses the return address to index into a hash table to determine which instruction missed. This solution has a higher miss cost (since you need to do the hash table lookup), but has no overhead on a cache hit. We will quantify the overhead of frequently changing the MHAR later in Section 4.

2.3 Summary

Compared to current approaches, the methods considered here have several advantages:

—*general*: independent of a particular hardware organization;

—*fine grained*: allow low-level memory system observation;

—*selective notification*: invoked only on triggering action events;

—*low overhead*: little program perturbation unless invoked.

Due to their generality, informing memory operations can have the same software interface even when implemented on different hardware platforms. Their fine granularity allows for pinpoint memory observation, as opposed to the coarser observation needed with counters and other techniques. The last two characteristics listed—selective notification and low overhead—allow for selective monitoring of "interesting" system events without undue overhead or perturbation.

All of the proposed methods have similar performance and reasonable hardware costs. The simplest approach—relying on a cache outcome condition code—adds an instruction to check the status of the previous memory operation (in program order). This method incurs an overhead of one instruction in the case of a cache hit. Later in the article we show that the cost of the extra instruction per informing memory operation is modest, but not zero.

If zero overhead in the hit case is desired, the low-overhead cache miss trap can be used. This approach is slightly more complex, since the control transfer instruction is implicit, rather than explicit as in the first approach. While more complex, the trap-based mechanism has a significant advantage: one can monitor whole system behavior without program instrumentation. To accomplish this, one allows the MHAR to default to a general handler for all running processes. In contrast, if informing memory operations were implemented using cache outcome condition codes, programs would need be compiled or instrumented with the new conditional branches in order to take advantage of the mechanism.

When considering a multilevel hierarchy of caches, a combination of approaches can be useful. The cache miss trap offers zero overhead in the hit case, so it can be desirable for the level-one cache. Subsequent levels of the cache hierarchy can have cache outcome condition codes. These codes could be read and acted on as part of the miss handler for the level-one cache.

3. IMPLEMENTATION ISSUES

While the mechanisms proposed are architecturally different, the complexity of each hardware implementation is similar and focuses on logic issues involved in the safe and efficient changing of control flow when cache misses occur. We use the implementation of low-overhead cache miss traps (discussed earlier in Section 2.2) to clarify these issues. Most of the hardware support necessary for implementing this approach (as well as the other informing memory options) already exists for handling branches and exceptions in current machines. The rest of this section outlines the small hardware changes that need to be implemented to support low-overhead cache miss traps in the Alpha 21164, an in-order-issue superscalar microprocessor, and the MIPS R10000, a more complex out-of-order microprocessor.

3.1 In-Order-Issue Machines

The 21164 is a superscalar implementation of the Alpha architecture [Edmondson et al. 1995] that can execute up to 4 instructions per cycle in program order. Its integer pipeline is 7 stages in length and implements an interesting stall model. All register dependences are handled before an instruction is issued (by using presence bits on the register file); once an instruction is issued (i.e., left the fourth pipeline stage) it cannot be stalled. Difficult situations are handled using a "replay trap," and one such situation already involves the cache. Namely, if a load has been issued, and an instruction that uses this data is waiting to issue, the machine will issue the dependent instruction at the correct timing for a cache hit (two cycles after the load). If the load does not hit in the cache, the machine is in trouble, since it does not have the data for the dependent instruction and there is no way to stall this already issued instruction. To handle this situation, the machine takes a replay trap, which flushes the pipeline, killing the load-dependent instruction and all subsequent instructions. The trap then restarts this instruction. The machine is then free to stall this restarted instruction in the issue stage if the load data is still not yet available from the memory system.

We use this same replay trap mechanism to implement our low-overhead traps. In this case, the replay trap occurs simply because of a cache miss signal and not in response to the speculative issuing of a data-dependent instruction. Notice that this new replay trap occurs for both load and store instructions (that are informing). Effectively, the instruction occurring immediately after an informing memory operation is marked as dependent upon the informing memory operation's hit/miss signal, and the memory operation is predicted to hit in the cache. If the operation misses, a replay trap occurs. Rather than reissuing the marked instruction, however, the machine issues the implicit branch-and-link instruction with a PC address equal to the informing memory operation (so that the MHRR is loaded with the appropriate return address). The nonblocking memory operation completes in this scenario, since the replay trap occurs on the next instruction. This implementation is slightly complicated by the fact that the cache miss information is not saved in any user-visible state, so the hardware must ensure that the informing memory operation and the trap are atomic; hence external exceptions must either occur before the memory operation or after the trap occurs. Though this issue does make the exception control slightly more complex, it is a solvable problem.

3.2 Out-of-Order-Issue Machines

Compared to the implementation for an in-order-issue machine, the hardware for low-overhead cache miss traps in an out-of-order machine is more complex. We explore this hardware by describing the implementation of low-overhead traps for informing memory operations in the MIPS R10000 [Yeager 1996]. The key problem is keeping track of dependences and ordering, since the execution ordering of operations is relaxed. As with the in-order machine, we mainly reuse existing machine mechanisms to implement this functionality.

There are two types of dependences that out-of-order-issue machines normally track. Since the machine must look like a simple in-order machine, it tracks instruction order and maintains the ability to execute precise interrupts. It also must track true data dependences between the instructions, to ensure that it lets instructions execute only when all the inputs are available. The former constraints are tracked in the reorder buffer, which holds instructions after they are fetched until they "graduate" (i.e., are committed to the architectural state of the processor). The latter constraints (data dependences) are tracked by the renaming logic. This logic creates a new space for a register each time it is written, and gives this identifier to all subsequent instructions that depend on this value.

Branches and exceptions are normally some of the more difficult situations to handle in out-of-order machines. When a branch occurs, the machine's fetch unit makes a prediction and continues to fetch instructions. These speculative instructions are then entered into the renaming logic and the reorder buffer. If the branch is mispredicted, all these speculative instructions must be squashed. In a similar manner, when an exception occurs, all the instructions after the instruction that excepts must be squashed. The R10000 uses two different mechanisms to handle these situations. For branches, it uses shadow state in the renaming logic. Each time the renaming logic sees a branch, it creates a shadow copy of its state and increments a basic-block counter. For branch misprediction, the rename state is rolled back to its state before the branch, and all instructions in the machine with a basic-block count greater than the branch are squashed. This allows the machine to recover from misprediction as quickly as possible, but requires substantial hardware support. In contrast, on an exception, the key aspect of the hardware mechanism is not to minimize the delay between recognizing the exception and starting the resulting action (as in the branch misprediction case), since exceptions occur only rarely. Instead, we must guarantee that all preceding instructions complete successfully before invoking the exception handler. To accomplish this requirement, the R10000 waits until the excepting instruction is at the top of the graduation queue. At this point all previous instructions have completed, and all instructions still in the machine need to be squashed; thus the machine is then cleared, and the exception vector is fetched.

Low-overhead cache miss traps can be implemented using either the branch or exception mechanism in this machine. Using the branch mechanism reduces the overhead on a cache miss, but has more significant hardware implications. We convert the memory reference into a reference and branch combination; the branch is dependent on the cache missing and is predicted not-taken. The major hardware cost is not adding new hardware functionality to the machine, but rather that we need more of the existing resources, because we consume them much faster than before. Since each branch requires the machine to shadow the entire renaming space, the R10000 currently allows only four predicted branches in execution at any time. If each reference becomes a potential branch, we will need about three times as much shadow state to hold the same number of issued instructions, since there are typically two memory operations per branch.

If this additional hardware is too costly, the low-overhead cache miss trap can be treated more like an exception than a branch. In this case the handler invocation time is longer since the machine waits until the reference is at the top of the graduation queue before the handler is started. The hardware needed for this scheme is quite modest. The reorder buffer records whether a memory operation missed. When an instruction that suffered a miss graduates, the machine is flushed as though an exception happened on the next instruction, and the MHAR is loaded into the PC. As we mention later, during our experiments in Section 4, we have observed a noticeable but not enormous performance difference between these branch- and exception-based techniques (a 7%–9% performance loss in the integer SPEC92 COMPRESS benchmark).

3.3 Cache as Visible State

Since informing memory operations enable users to get information about the state of the cache, they make it possible for a program's *actions* (and not just its performance) to be influenced by the contents of the cache. This new capability can create a problem since (for efficiency) hardware designers often allow speculative operations to affect the cache state. For applications that use informing memory operations as a way to get feedback on memory performance, the speculative update causes no problems—the application gets the desired information on its cache behavior. However, the situation is more complicated for applications such as cache coherence (discussed later in Section 5) which view the cache as a collection of data that has passed some form of access check. While it is acceptable for data to be flushed from the cache speculatively (which is unavoidable), the application will fail if data can enter the cache without invoking the associated miss-handling code. Unfortunately, given the description of informing operations so far, this situation is not only possible, but also quite likely. All that needs to occur is for a speculative informing load to issue, bring its data into the cache, and then be squashed; although this does not produce what we normally think of as side-effects (i.e., the register state remains unchanged, and the miss handler is not invoked), it does produce the side-effect of bringing data into the cache without invoking the miss handler. This section describes the hardware that must be added to an out-of-order-issue machine to permit speculative execution of load instructions but yet prevent these speculative loads from silently updating the first-level cache state.

Typically, the problem of speculative update occurs only with load instructions. Most processors do not allow stores to probe the cache until they commit, and thus the implementation described in this section assumes that store probes are not done speculatively. However, it is straightforward to extend our implementation to handle speculative stores as well as loads.

The basic problem to address then is the updating of the cache when an informing load operation misses and the load is later invalidated. This situation arises in two ways: the informing load was control-dependent on a preceding branch that was mispredicted; or a preceding instruction signaled an exception flushing the pipeline and the informing load. In both cases, the informing load must have executed out-of-order and suffered a cache miss (no problem exists for in-order-issue machines, since memory requests are sent out only if the operation will complete). The miss will start the execution of the cache miss handler, only to have the reference and the miss handler squashed.

For performance reasons, our solution to this problem must allow for the expedient return of speculative load data to the out-of-order execution engine. However, we must guarantee that the data updates the first-level cache only when the informing load operation commits. Our solution actually permits the speculative informing load operation to update the cache on a miss. More frequently than not, the speculation is correct, and thus we have optimized for the common case. To handle the case where the load operation was squashed, we must now invalidate the data in the first-level cache. This approach can reduce the cache performance by flushing potentially useful data. However, this data often still resides in the second-level cache, and we have effectively prefetched the informing load data into the second-level cache.

To implement our invalidate mechanism, we extend the lifetime of the Miss Status Handling Registers (MSHR), the data structure used to track the outstanding misses in a lockup-free cache [Farkas and Jouppi 1994]. This structure contains the address of the miss, the destination register identifier, and other bookkeeping information. Normally, when data returns to the cache, the MSHR forwards the correct data to the processor; if the load is squashed before the data returns, the MSHR is notified to prevent it from forwarding data to a stale destination. We slightly modify this functionality to remove unwanted speculative load data. Since the MSHR already prevents the result of a miss fetch from entering the first-level cache if a load instruction is squashed before the data returns, we simply have to solve the problem of invalidating a cache line if an informing load miss completes before the load is ultimately squashed. To do this, we extend the lifetime of the MSHR so that it is freed only after a memory instruction is either squashed or graduates. If the load is squashed (for either reason listed above), the address in the MSHR is used to invalidate the line in the cache (i.e., change the tag state) before the MSHR is freed for reuse. The cost of this change is modest. In the common case where the load is not invalidated, we simply hold on to the MSHR longer than normal. In our simulation studies, this longer capture of an MSHR did not change the required number of MSHRs—eight were sufficient in all cases.

4. USING INFORMING MEMORY OPERATIONS: AN OVERVIEW

We now focus on how informing memory operations can be used to enhance performance. We begin with short descriptions of software techniques that can exploit informing memory operations. We then quantify the execution overheads of invoking the informing mechanism in modern superscalar processors. Later, in Section 5, we present detailed case studies of several of these applications.

4.1 Description of Software Techniques

Informing memory operations can benefit a wide variety of software-based memory optimizations, and we present only a partial list of such techniques in this section. While the performance benefit of each technique varies, the runtime overhead of using the informing mechanism is largely dictated by (1) the amount of work to be performed in response to a miss and (2) how frequently the handler address must be changed. The former property affects the overhead per miss (the effect is not strictly linear, since some handler code may be overlapped with the miss), and the latter property dictates the overhead even on hits. (Recall that low-overhead traps eliminate hit overhead only if we reuse the same miss handler). We address both of these issues in our discussion of each technique, and we quantify in Section 4 their impact on performance.

4.1.1 *Performance Monitoring.* Performance-monitoring tools collect detailed information to guide either the programmer or the compiler in identifying and eliminating memory performance bottlenecks [Burkhart and Millen 1989; Goldberg and Hennessy 1993; Lebeck and Wood 1994; Martonosi et al. 1995]. A major difficulty with such tools is how to collect sufficiently detailed information quickly and without perturbing the monitored program. The high overheads of today's memory-observation techniques have resulted in tools that either provide coarse-grained information (e.g., at loop level rather than reference level), or else rely on simulation (which is relatively slow, and where speed improvements tend to reduce accuracy).

Informing memory operations enable a wide array of accurate and inexpensive monitoring tools, ranging from simply counting cache misses (a single register-increment miss handler) to correlating misses of individual static references with high-level semantic information such as the data structures being accessed and the control flow history. As is shown in Section 5 informing memory operations can be used to collect precise per-reference miss rates with low runtime overheads (typically less than a 75% increase in program runtime) and tolerable cache perturbations. This tool uses a single miss handler containing roughly 20 instructions to increment a hash table entry based on the branch-and-link return address (available in the MHRR), thus distinguishing all static references. Overall, the number of instructions in the miss handler of a performance monitoring tool might vary from one instruction to hundreds of instructions, depending on the sophistication of the tool.

4.1.2 *Software-Controlled Prefetching.* Informing memory operations are useful not only for measuring memory performance, but also for actively improving it. One example of this benefit occurs with *software-controlled prefetching*, which is a technique for tolerating memory latency by moving data lines into the cache before they are needed [Luk and Mowry 1996; Mowry 1994; Porterfield 1989]. A major challenge with prefetching is predicting which dynamic references are likely to miss, since indiscriminately prefetching all the time results in too much overhead [Mowry et al. 1992]. Informing memory operations can address this problem in two ways: (1) through feedback-directed recompilation based on a detailed memory profile captured from an earlier run (e.g., using the profiling tool described later in Section 5) or (2) through code capable of adapting its prefetching behavior "on-the-fly" based on its dynamic cache miss behavior. We demonstrate the benefit of both of these approaches later in Section 5. For now, we state as background for Section 4 that the size of a miss handler for prefetching is likely to be small (less than 10 instructions) and that the miss handler address is likely to change frequently. The handler is small because it is either launching a handful of prefetches or else recording some simple statistics; the handler address may change frequently because we often want to tailor a prefetching response to its context within the program.

4.1.3 *Enforcing Cache Coherence*. An application that demonstrates the versatility of informing memory operations is the enforcement of cache coherence with fine-grained access control. We discuss this application in detail later in Section 5, but for now, the key parameters are the following: only a single miss handler is required, which is enabled for all potentially shared references and executes 20–30 instructions to check the coherence state, and the dependence chain through these instructions is roughly 10 cycles long.

4.1.4 *Software-Controlled Multithreading*. Multithreading tolerates memory latency by switching from one thread (or "context") to another at the start of a cache miss [Agarwal et al. 1993; Alverson et al. 1990; Laudon et al. 1994; Smith 1981; Thekkath and Eggers 1994]. Multithreading implementations to date have generally relied upon hardware to manage and switch between threads. However, informing memory operations enable a software-based approach where a single miss handler could save and restart threads (all under software control) upon cache misses. Two optimizations would help the performance of this scheme. First, invoke a thread switch only on secondary (rather than primary) cache misses—such references could be isolated through a combination of static prediction and dynamic observation, thus allowing us to selectively enable the miss handler accordingly. Second, the overhead of saving and restoring register state could be minimized through compiler optimizations (e.g., statically partition the register set amongst threads, only save or restore registers that are live, etc.), or perhaps through hardware support (e.g., something similar to the SPARC register windows [Paul 1994]). A single miss handler should suffice (although it may be selectively enabled to isolate secondary misses), and its length may vary from a handful to over 100 instructions depending on how aggressively we can eliminate register-saving/restoring overhead.

Table I. Pipeline Simulation Parameters for the Superscalar Processors (roughly based on the MIPS R10000 and Alpha 21164 processors, with a few modifications)

Pipeline Parameters	Out-Of-Order	In-Order
Issue Width	4	4
Functional Units	2 Int, 2 FP, 1 Branch, 1 Mem	2 Int/Mem, 2 FP, 1 Branch
Reorder Buffer Size	32	N/A
Integer Multiply	12 cycles	12 cycles
Integer Divide	76 cycles	76 cycles
FP Divide	15 cycles	17 cycles
FP Square Root	20 cycles	20 cycles
All Other FP	2 cycles	4 cycles
Branch Prediction Scheme	4096 2-bit counters	1024 2-bit counters

4.2 Overhead of Generic Miss Handlers

Each of the techniques we just described has a benefit and a potential cost in terms of performance. While the benefit is specific to the particular technique, the cost can be abstracted as a function of the length of the miss handler and how frequently the miss handler address must be changed. In this section, we vary these parameters to measure the execution overhead of several "generic" miss handlers on modern superscalar processors. Given the ability of these processors to exploit instruction-level parallelism and overlap computation with cache misses, the translation of increased instruction count into execution overhead is not immediately obvious without experimentation.

4.2.1 *Experimental Framework.* We performed detailed cycle-by-cycle simulations of two state-of-the-art processors: an out-of-order machine based on the MIPS R10000 and an in-order machine based on the Alpha 21164. Our model varies slightly from the actual processors (e.g., we assume that all functional units are fully pipelined, and we simulate a two-level rather than a three-level cache hierarchy for the Alpha 21164), but we do model the rich details of these processors including the pipeline, register renaming, the reorder buffer (for the R10000), branch prediction, instruction fetching, branching penalties, the memory hierarchy (including contention), etc. The parameters of our two machine models are shown in Tables I and II. We simulated 14 SPEC92 benchmarks (five integer and nine floating-point) [Dixit 1992], all of which were compiled with -O2 using the standard MIPS compiler under IRIX 5.3.

We simulate 1-, 10-, and 100-instruction generic miss handlers, and we pessimistically assume that all instructions within the handlers are data-dependent on each other (hence a 10-instruction handler requires 10 cycles to execute). We simulate both a case with no overhead on hits (i.e., low-overhead cache miss traps with a single handler) and a case where a single instruction is added before every memory reference to specify the handler address or to represent the explicit cache check. We model the full details of fetching and executing all instructions associated with informing memory operations, including their impact on instruction cache perfor-

Table II. Memory Simulation Parameters for the Superscalar Processors (roughly based on the MIPS R10000 and Alpha 21164 processors, with a few modifications)

Memory Parameters	Out-Of-Order	In-Order
Primary Instruction and Data Caches	32KB, 2-way set-associative	8KB, direct-mapped
Unified Secondary Cache	2MB, 2-way set-associative	2MB, 4-way set-associative
Line Size	32B	32B
Primary-to-Secondary Miss Latency	12 cycles	11 cycles
Primary-to-Memory Miss Latency	75 cycles	50 cycles
MSHRs	8	8
Data Cache Banks	2	2
Data Cache Fill Time	4 cycles	4 cycles
Main Memory Bandwidth	1 access per 20 cycles	1 access per 20 cycles

mance, instruction fetch bandwidth, consumption of functional unit resources, branch latencies, etc. We statically predict that informing memory operations enjoy cache hits and therefore do not alter the control flow—hence an informing memory operation that suffers a cache miss results in the same penalty as a mispredicted branch. Since this prediction is done statically, informing memory operations have no impact on the accuracy of the branch prediction hardware for normal conditional branches. The instruction fetcher uses the MHRR to predict the return address from the miss handler, similar to the way that a return address stack is used to predict the target of a procedure return.

4.2.2 *Experimental Results*. Our results with 1- and 10-instruction miss handlers are shown in Figures 1 and 2. (The su2cor benchmark is shown separately in Figure 2, since it behaves differently and requires a larger y-axis scale). For each benchmark, we show five bars: the case without informing memory operations (N), and cases with a single miss handler (S) and unique miss handlers per reference (U) for both miss handler sizes. These bars represent execution time normalized to the case without informing loads, and they are broken down into three categories explaining what happened during all potential graduation slots. The bottom section is the number of slots where instructions actually graduate. The top section is any lost graduation slots that are immediately caused by the oldest instruction suffering a data cache miss, and the middle section is all other slots where instructions do not graduate. Note that the "cache stall" section is only a first-order approximation of the performance loss due to cache stalls, since these delays also exacerbate subsequent data dependence stalls.

Starting with Figure 1, we see that for 12 of these 13 benchmarks (all but tomcatv), the execution overhead of using informing memory operations is less than 40% under both processor models. Even in tomcatv, the execution overhead is less than 25% in all cases except the 10-instruction miss handlers on the in-order machine. Applications that suffer more from cache

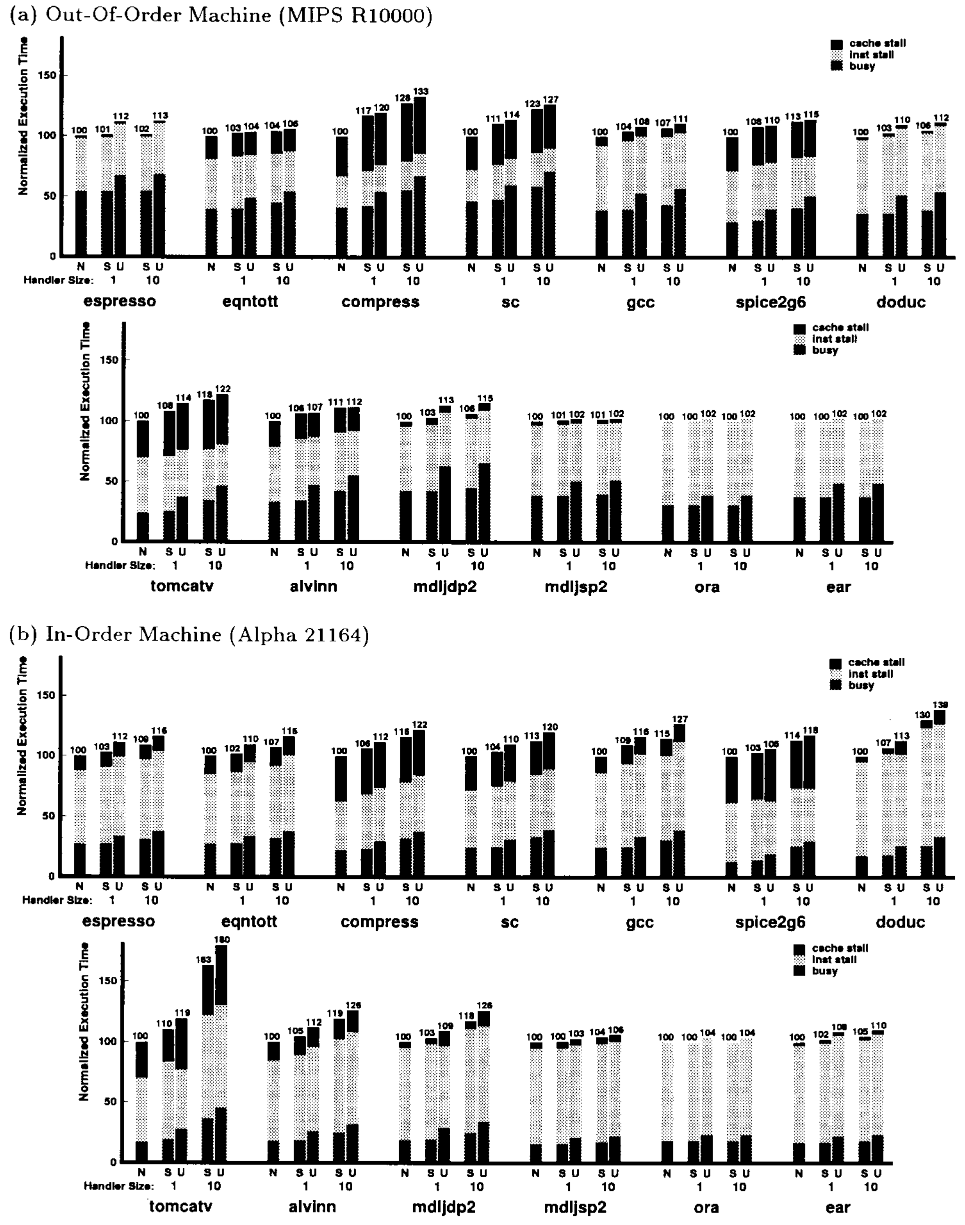


Fig. 1. Performance of generic miss handlers containing 1 and 10 instructions (**N**= no miss handler, **S** = single miss handler, **U** = unique miss handler per static reference).

stalls tend to have larger overheads, which makes sense, since they invoke the handler code more frequently. Another trend (particularly in the out-of-order model) is that a significant fraction of the additional instruc-

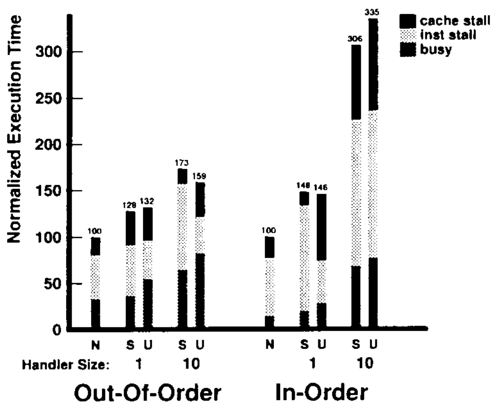


Fig. 2. Performance of the su2cor benchmark with generic miss handlers containing 1 and 10 instructions (**N** = no miss handler, **S** = single miss handler, **U**= unique miss handler per static reference).

tions needed to specify unique handler addresses can be overlapped with other computation, thereby having only a relatively small impact on execution time. For example, the instruction count for both mdljsp2 and alvinn under the out-of-order model increases by over 30%, but the execution time only increases by 1%. This means that techniques that need to modify the MHAR frequently (e.g., prefetching, multithreading, and cache coherence) will not suffer much of a performance penalty for doing so. It also means that the performance lost by using an explicit cache condition code check will be small.

Another interesting comparison is the ability of each processor model to hide the overhead of 10- versus 1-instruction miss handlers. While the results are somewhat mixed for the integer benchmarks, the clear trend in the floating-point benchmarks is that the out-of-order model suffers a much smaller relative performance loss than the in-order model with larger miss handlers. This effect is particularly dramatic in tomcatv, where the difference in overhead is under 10% for the out-of-order model, but greater than 45% for the in-order model. This behavior makes sense for the following reason. In contrast with the integer benchmarks, branches are easier to predict in floating-point programs (since they are dominated by the backward branches of loops which are almost always taken), and therefore the instruction fetcher typically does a good job of keeping the reorder buffer full. With more instructions to choose from in the reorder buffer, the machine has greater flexibility in reordering computation to hide the overhead of the larger miss handlers.

In contrast to the other 13 benchmarks, the su2cor benchmark shown in Figure 2 has considerably larger execution overheads, particularly for the in-order model. The problem in the in-order model is that su2cor suffers from severe cache conflicts in the 8KB direct-mapped primary data cache, hence triggering the 10-instruction miss handler frequently enough to quintuple the instruction count and triple the execution time. (A similar problem occurs to a lesser extent in tomcatv.) The surprising result that this application sometimes runs faster with unique handlers than with a single handler is because different handlers are not data-dependent on each other in our model, whereas a single handler is data-dependent on its last invocation. Therefore having independent handlers happens to result in more parallelism in these experiments.

We also simulated generic miss handlers containing 100 data-dependent instructions, and found that the execution times increased significantly for the applications that suffer the most from cache misses (e.g., 6 times slower for compress and 7 times slower for su2cor). For applications with few cache misses, the overheads remained low (e.g., only a 2% overhead for ora). The only technique likely to use such expensive miss handlers would be fancy performance-monitoring tools, and in that case optimizations such as sampling could be used to reduce the overhead.

All of our out-of-order experiments so far used the model where an informing trap is handled the same way as a mispredicted branch (as discussed earlier in Section 3). We also simulated the case where informing traps are treated as exceptions (i.e., the trap is postponed until the informing operation reaches the head of the reorder buffer), and found that this increased the execution times for 1- and 10-instruction handlers by 9% and 7%, respectively, for the compress benchmark. Therefore the additional complexity of handling informing traps as mispredicted branches does buy us something in terms of performance.

In summary, we have seen that the overheads of informing memory operations are generally small for 1- or 10-instruction miss handlers, but the overheads can become large in some cases. Whether the overhead is acceptably small depends on how informing memory operations are being used. For example, a performance-monitoring tool can potentially tolerate a two-fold increase in execution time provided that the tool is still providing useful information. On the other hand, an application that is attempting to improve memory performance on-the-fly (e.g., software-controlled multithreading) obviously cannot tolerate an overhead that exceeds its reduction in memory stall time. Given the wide spectrum of approaches enabled by informing memory operations, the software designer has the flexibility to choose the right balance for their particular application. As for the hardware designer, the fact that there is generally little runtime cost in executing one extra instruction per memory reference (either to check cache state or set a miss handler address) gives them considerable flexibility in how an informing mechanism is implemented.

5. USING INFORMING MEMORY OPERATIONS: CASE STUDIES

To illustrate the versatility of informing memory operations, we now present case studies on how they can be applied in the following three areas: building effective performance-monitoring tools, improving the performance of software-controlled prefetching, and accelerating cache coherence with fine-grained access control.

5.1 Performance-Monitoring Tools

A number of performance tools have been proposed to monitor program-caching behavior [Burkhart and Millen 1989; Goldberg and Hennessy 1993; Lebeck and Wood 1994; Martonosi et al. 1995]. 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 [Lam et al. 1991]. 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. More recently, DCPI has emerged as a viable, sampling-based tool for collecting detailed statistics including memory behavior [Anderson et al. 1997]. DCPI gathers detailed statistics with low overhead, but offers primarily an aggregate view over long-term executions. Informing memory operations, by constrast, can offer finer views based on fewer procedure invocations or shorter runs of the program. This allows one to respond more directly and quickly to perceived performance bottlenecks.

For finer-grained memory statistics, other tools have relied on dedicated hardware to monitor memory references. Work by Burkhart and Millen [1989] as well as others has 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. More recently, ProfileMe has been proposed as a more aggressive hardware-supported monitoring technique. Largely complementary to informing memory operations, ProfileMe offers a larger body of information, but without the ability to respond in a fine-grained way to a particular “misbehaving” instruction [Dean et al. 1997].

Because of the drawbacks in other types of monitoring, tools such as MemSpy [Martonosi et al. 1995] and CProf [Lebeck and Wood 1994] 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 trade-off 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.

5.1.1 *Performance Monitoring Based on Informing Memory Operations*. Because informing memory operations are a general primitive, one can imagine using them to implement many different types of 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 memory operations for performance monitoring, we implemented a fairly simple, low-overhead tool which collects the precise cache miss counts for all static memory reference instructions except for those that reference the stack (we ignore stack references since they are likely to enjoy cache hits). Our tool is based on the low-overhead cache miss trap mechanism and works as follows. Upon a cache miss, we use the contents of the MHRR register to uniquely identify the static memory reference instruction which suffered the miss; based on this instruction address, we perform a hash table lookup to find a unique counter associated with this instruction and increment its miss count.

To help minimize the overhead of our tool and any data cache perturbation that it might induce, we can use a sampling methodology whereby the miss handler only updates the counters once every N misses. Although sampling has also been used in previous performance-monitoring tools, they are forced to use large sampling intervals due to the heavyweight nature of the way they observe misses; these large sampling frequencies limit their ability to collect accurate fine-grain information. In contrast, with informing memory operations we have far greater flexibility in choosing our sampling rate—even a sampling rate of one (i.e., recording every miss) is feasible, as we see in our experimental results.

5.1.2 *Experimental Results*. We extended our simulator (described earlier in Section 4) to perform a detailed cycle-by-cycle timing analysis of a complete implementation of the performance tool described above. Figure 3 shows the results of our experiments on a collection of 12 SPEC92 benchmarks [Dixit 1992], where all bars are normalized to the original unmonitored performance. For each application, we show three performance bars corresponding to sampling rates of 1, 100, and 1000. (Note that $N = 1$ corresponds to the no-sampling case.) With in-order execution, execution time overheads for the no-sampling (N) implementation range from essentially no overhead for the ora benchmark, to roughly 300% additional runtime required for su2cor. The wide range in overheads stems from variance in application reference rate, cache miss rate, and the degree that

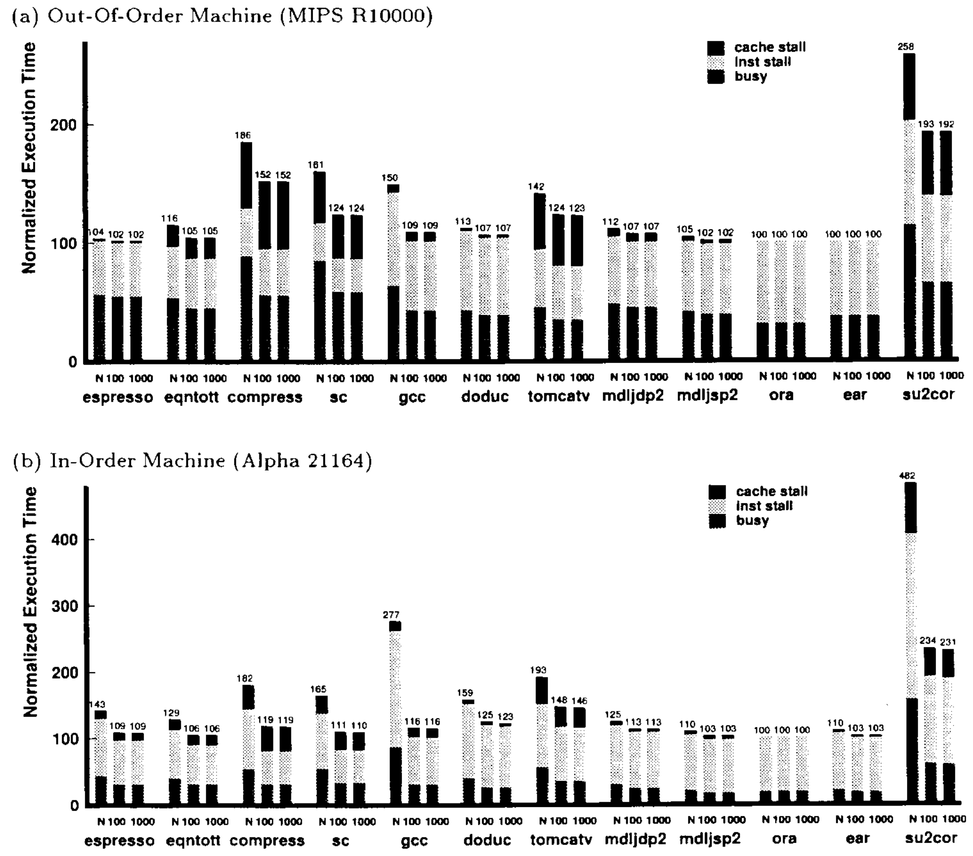


Fig. 3. Normalized execution times for miss counting tool. The bars labeled **N** correspond to monitoring every miss, with no sampling. Those labeled **100** correspond to using sampling to monitor every hundredth miss, and those labeled **1000** monitor every thousandth miss. Execution times are normalized to the runtime for the original, unmonitored code.

the monitoring code perturbs an application's caching behavior. An out-of-order execution model does a better job of hiding the monitoring overhead in what would otherwise be stall time. For the R10000 model, seven of the applications have nonsampling overheads of less than 20%.

Through the use of sampling, we can reduce monitoring overheads even further. With out-of-order execution, monitoring every thousandth reference yields overheads of under 10% in all but four applications. Even su2cor can be monitored with only a 92% slowdown. With in-order execution, overheads with sampling drop significantly as well—under 20% for all but three applications.

For comparison, Mtool produces much less detailed statistics, but in spite of this still reports overheads in the range of 3%–15% (for a subset of these applications) [Goldberg and Hennessy 1993]. MemSpy uses a simulation-based approach to collect detailed memory statistics, and its execution time overheads typically range from 700%–1600%. Overall, performance overheads for an informing-memory-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. Although we show only a single level of the memory hierarchy in these results (in contrast with simulators which typically show multiple levels), we feel that this disadvantage is outweighed by the fact that we can 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.

Overall, the tool implementation described here highlights the strengths of informing memory operations. By providing a very low overhead means of observing and reacting to cache misses, informing memory operations give crucial support to fine-grained memory-performance-monitoring tools.

5.2 Software-Controlled Prefetching

In addition to measuring memory performance, informing memory operations can also *improve* memory performance by enhancing optimizations such as software-controlled prefetching. Software-controlled prefetching tolerates latency by explicitly moving data lines into the cache before they are needed. To ensure that the overheads of prefetching do not offset the gains, it is important to issue prefetches only for those dynamic references that are likely to suffer misses [Mowry et al. 1992]. Without informing memory operations, the success of prefetching depends 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 caches with limited associativity make it difficult to model caching behavior accurately. Therefore we expect the incorporation of dynamic information into this decision-making process to be an important step toward overcoming the inherent limitations of static analysis [Mowry and Luk 1997].

Dynamic information can be fed back into the optimization process in two ways: (1) between runs of a program, whereby we might recompile for a second run based on the behavior of the first run or (2) 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 demonstrate how software-controlled prefetching can benefit from both of these approaches.

5.2.1 *Using Dynamic Memory Information at Compile-Time*. The idea of using dynamic information at compile-time is not new. Compilers have

```
(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;
```

Fig. 4. Examples containing references that may suffer misses only occasionally.

historically used control-flow feedback (also known as "branch profiling") to perform aggressive instruction scheduling across branches [Fisher 1981; Smith 1992]. Given that informing memory operations make it practical to collect accurate per-reference miss rates across entire applications (as demonstrated earlier in Section 5), a similar feedback methodology can be used to enhance 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 [Mowry 1994; Mowry et al. 1992]. A previous study has also demonstrated that compiler-inserted prefetching can improve the performance of pointer-based codes [Luk and Mowry 1996]. For the regular array-based codes, the compiler predicts caching behavior using *locality analysis*. 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 locality model. For the irregular pointer-based codes, the compiler does not attempt to model caching behavior, due to the difficulty of understanding pointer values. 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.

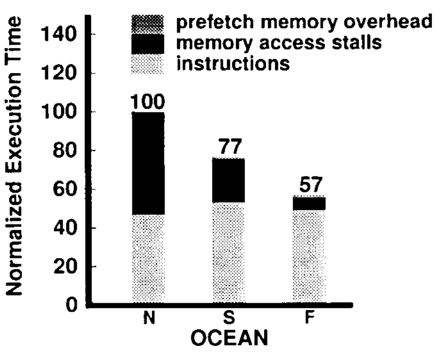
Although precise per-reference miss rates may sound like perfect information, a subtle issue is how to handle references with intermediate miss rates (between 0% and 100%). Ideally we would like to prefetch such references only when they miss, but unfortunately the information relating individual misses 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. 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(a). Assume that each element of the array `a` is 8 bytes, a cache line contains 32 bytes, the primary cache size is 8KB, 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.

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) [Luk and Mowry 1996]. For example, the 8.3% miss rate of a[j] in Figure 4(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 the loop iteration modulo the cache line size (to capture spatial locality). For example, if a[j] in Figure 4(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(b), the dynamic context might consist of paths in the control-flow graph that arrive at that point. For example, if we discover 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 memory operations—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 [Mowry and Luk 1997].

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 memory operations are used to collect the miss rates of all load references (similar to the monitoring code described in Section 5), but misses are not correlated with when they occur. We then augment the compiler algorithm presented in Mowry [1992] 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



(a) Performance

```
for(j = 2; j < jm; j++) {
  laplacalc(j);

...

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

(b) Key Loop Nest (Abstract Representation)

Fig. 5. Performance of OCEAN using memory feedback at compile-time (**N** = no prefetching, **S** = prefetching with static locality analysis only, **F** = prefetching with memory feedback).

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 Mowry [1994]). 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 [Mowry 1992] using the same architectural assumptions.[1] One of the cases improved significantly using memory feedback: OCEAN, which is a uniprocessor version of a SPLASH application [Singh et al. 1992]. Figure 5(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).

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 5(b) shows a simplified

[1]The architecture consists of a single-issue processor with an 8KB on-chip primary data cache and a 256KB unified secondary cache. Both caches are direct-mapped, "write-back write-allocate" and use 32-byte lines. The penalty of a primary cache miss that hits in the secondary cache is 12 cycles, and the total penalty of a miss that goes all the way to memory is 75 cycles. To limit the complexity of the simulation, we assume that all instructions execute in a single cycle and that all instructions hit in the primary instruction cache.

version of this scenario.) Since our version of the SUIF compiler [Tjiang and Hennessy 1992] 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 memory operations were fully exploited to correlate misses with their dynamic contexts [Mowry and Luk 1997]. Finally, as suggested in an earlier study [Luk and Mowry 1996], 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.

5.2.2 *Using Dynamic Memory Information at Runtime*. 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 runtime.

Rather than generating code with a fixed prefetching strategy, another possibility is to generate code that adapts its behavior dynamically at runtime. For example, if informing memory operations 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 runtime.

Since a potential drawback of adapting at runtime 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, we can often detect trends through relatively infrequent checks of the dynamic behavior, since miss patterns often recur in a given pass through a loop. Therefore a reasonable strategy might be to

(a) Original Code

```
/* is A[i] already in the 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

```
/* initialize prefetch miss count */
pf_A_miss_count = 0;
/* miss handler will count prefetch misses */
set_miss_handler(pf_miss_counter);

/* issue the first several prefetches */
for (i = 0; i < 10; i+=2)
  /* any misses will invoke pf_miss_counter() */
  prefetch(&A[i]);

set_miss_handler(NULL);
/* 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];
}
...

void pf_miss_counter() {
  /* increment count upon a prefetch miss */
  pf_A_miss_count++;
}
```

Fig. 6. Example of how dynamic miss counts can be used to adapt prefetching in array-based codes.

instrument an initial set of iterations, and let their overall behavior guide the approach to handling the remaining iterations. Figure 6 illustrates how this could be implemented with very little runtime overhead. Assuming that two elements of A fit within a cache line, all 1000 elements of A can potentially fit in the cache, and 10 loop iterations are sufficient to hide memory latency. Figure 6(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 6(c). Here we use

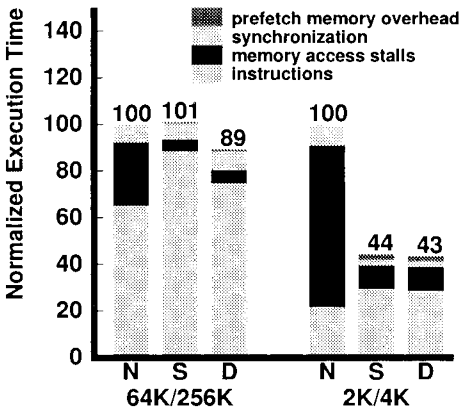


Fig. 7. Performance of a 200 × 200 LU on different cache hierarchy sizes (**N** = no prefetching, **S** = statically prefetch all the time, **D** = adapt prefetching dynamically). Performance is renormalized for each cache size.

informing *prefetches*[2] 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.

To demonstrate the performance advantages of dynamically adaptive prefetching, we now present experimental results for LU, which is a parallel shared-memory application that performs LU decomposition for dense matrices. The primary data structure in LU is the matrix being decomposed. Working from left to right, a column is used to modify all columns to its right. Once all columns to the left of a column have modified that column, it can be used to modify the remaining columns. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits until a column has been produced, and then that column is used to modify all columns that the processor owns. Once a processor completes a column, it releases any processors waiting for that column. For our experiments we performed LU decomposition on a 200 × 200 matrix. We simulated the performance of LU on a shared-memory multiprocessor resembling the Stanford DASH multiprocessor [Lenoski et al. 1992]. The architecture we model includes 16 processors with two-level cache hierarchies (64KB/256KB), and a low-latency interconnection network. Miss latencies for loads range from 15 cycles to the secondary cache to over 130 cycles for "remote dirty" lines.

[2]Informing prefetches are similar to other informing memory operations—the miss handler is invoked whenever the prefetch suffers a miss.

The complication in LU is that the same procedure is called repeatedly to apply a pivot column to other columns. If the columns are large relative to the cache size, then it is best to prefetch the pivot column each time it is referenced. On the other hand, if a column can fit in the cache, then the pivot column will only suffer misses the first time it is referenced. However, since this code is in a separate procedure, and since our compiler does not perform procedure *cloning* [Cooper et al. 1993], the only static option is to prefetch the column all the time. We modified this procedure by hand to use adaptive prefetching, similar to the code in Figure 6(c). The results of our experiments are shown in Figure 7.

We ran LU with both large and small problem-size–to–cache-size ratios. The left-hand side of Figure 7 shows the small ratio, meaning that columns tend to fit in the cache. In this case, the static approach of prefetching the columns all the time suffers from a significant amount of instruction overhead. In contrast, the adaptive code eliminates much of this unnecessary overhead while still hiding much of the memory latency, thereby resulting in the best overall performance. Looking at the right-hand side of Figure 7, where the data size is large relative to the cache, we see the interesting result that the adaptive code still offers the best performance. This is because LU performs a triangular matrix solve, and therefore the last several columns always tend to fit in the cache. The savings of not issuing unnecessary prefetches for these last several columns more than offsets the instruction overhead of the dynamic test in this case. Therefore the adaptive code is clearly the best choice for LU.

5.3 Cache Coherence with Fine-Grained Access Control

We now take a detailed look at using informing memory operations for fine-grained access control for parallel programs. Access control selectively limits read and write accesses to shared data, to guarantee that multiple processors see a coherent view of the shared data. It requires triggering handlers selectively on certain memory references, and performing actions within these handlers such as changing a block's state, sending out invalidations, etc. As discussed by Schoinas et al. [1994], there are many implementation trade-offs. In some machines, access control is implemented using bus-snooping hardware, auxiliary protocol processors, or specialized cache or memory controllers [Agarwal et al. 1995; Nowatzyk et al. 1994]. Using software-based techniques, one can instrument individual memory references with extra code to check protocol state and update it accordingly.

5.3.1 *Overview of Approach.* Our approach is similar to the Blizzard E scheme described by Schoinas et al. [1994]. In that scheme, blocks are put into an invalid state by writing them out to memory with invalid ECC. Subsequent accesses to these blocks trap to the ECC fault handler, and the coherence protocol operations are implemented within the fault handler. In our approach, informing memory operations are used to implement the block-level handlers that are invoked on read misses and on writes that

Table III. Machine and Experiment Parameters for Different Access Control Methods

Machine Parameters	16 processors 16KB L1 cache/proc (10-cycle miss penalty) 128KB L2 cache/proc (25-cycle miss penalty) 32-byte coherence unit 900-cycle 1-way message latency
Reference-Checking Approach	18-cycle lookup time 25-cycle state change time
ECC-Based Approach	250 cycles for read to invalid block 230 cycles for writes to a block on a page with any READONLY data
Informing Memory Approach	33-cycle lookup time (includes 6-cycle pipeline delay plus 9 handler cycles to determine if load or store) 25-cycle state change time

change the line's state; blocks that are invalid are evicted from the cache. When they are referenced, a cache miss occurs, which causes the informing memory operation's miss handler to run. The protocol operations are implemented in the cache miss handler; because it is tightly coupled to the cache access, the miss handler has a smaller invocation time than the ECC fault handler. The miss handler code maintains a per-cache-line data structure that summarizes protection information about the line's "state"; a line can either currently be INVALID, READONLY, or READWRITE. (As in the Blizzard systems, page-level handlers are also invoked when a page is first referenced by a processor.)

The cache miss handler performs a lookup of the current address in the protection state table. Based on whether the invoking reference is a read or a write, the handler determines if the line's current protection level is adequate for the access (i.e., READWRITE for a store, and either READONLY or READWRITE for a load). If so, then the reference continues, with no state changes and no further delay; otherwise, protocol operations are necessary. Local protocol operations are user-level changes to the state table, and some of the instructions needed for these changes may be overlapped with the miss latency itself. When remote protocol operations are needed, a handler running on one processor will need to induce an action (a cache invalidation) at another node. In our experiments, we assume remote operations are accomplished without interrupting the remote processor—e.g., using a user-level DMA engine and network interface per compute node [Blumrich et al. 1994].

5.3.2 *Results.* We present performance results for an informing memory operations implementation of access control, assuming the low-overhead cache miss trap scheme described in Sections 2 and 3. To gather statistics on parallel applications with meaningfully long runtimes, these results were generated using a parallel system simulator based on TangoLite, as opposed to the detailed uniprocessor simulator previously described. On

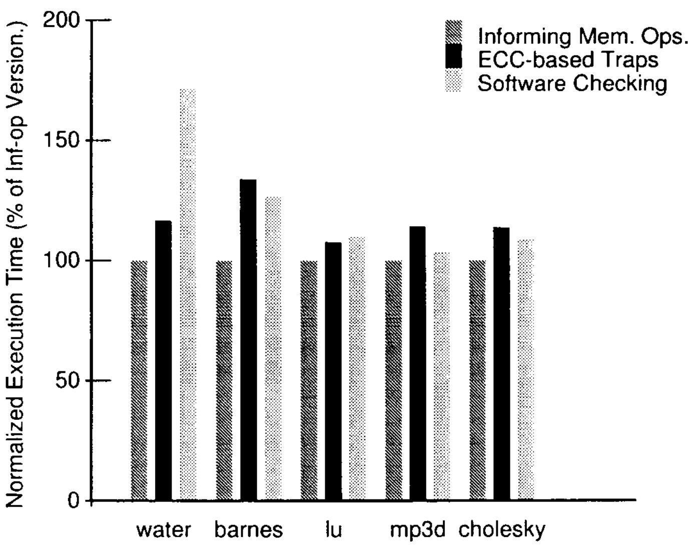


Fig. 8. Execution times for three access control methods, normalized to the scheme which uses informing memory operations.

application loads and stores, application processes incur additional delays for lookup and state change overheads when the access permissions need to be checked (on a miss). These are given in Table III. To give a feel for the relative performance of our access control method, we have also compared our approach to two other access control systems: (1) based on per-reference-checking (protection lookup on each potentially shared reference) and (2) based on ECC faults. These approaches are similar to Wisconsin Blizzard-S and Blizzard-E systems, respectively; we simulated their lookup and state-change times using parameters described in Schoinas et al. [1994] and listed in Table III.

We simulated all three access control methods using identical machine assumptions and parameters. Figure 8 shows the performance of an informing-memory-based coherence scheme compared to reference-checking or ECC-based approaches. While the relative performance of the reference-checking and ECC-based approaches fluctuates depending on application parameters (such as the frequency of reads versus writes), the informing-op-based approach always outperforms both of them. For these applications, the informing memory scheme is an average of 18% faster than the ECC-based scheme and 24% faster than the reference-checking scheme. Compared to the ECC-based scheme, the informing memory approach benefits from improved coherence action times. Compared to the reference-checking scheme, our approach benefits (in the no-coherence-action case) from performing lookups only on cache misses rather than on all references. Further experiments have also shown that either smaller network latencies or larger primary cache sizes tend to improve the relative performance of the informing memory implementation.

Clearly, access control and cache coherence are complicated issues with many trade-offs. Our main goal in this section is not the details of each of the access control implementations, but rather demonstrating that informing memory operations—included on commodity processors—provide another economical method for implementing access control.

In summary, this section has demonstrated several ways that informing memory operations can be used to improve performance. The low-overhead, selective notification of informing memory operations provided a building block that was used to collect the necessary information with minimal performance overhead. We expect that these capabilities will also benefit other automatic memory optimizations such as cache blocking and page coloring.

6. CONCLUSIONS

Informing memory operations are a general primitive for allowing software to observe and react to its own memory-referencing behavior. The primary goal of this article has been to explore these operations at a conceptual level, and to build insights on what aspects of their functionality are most important. To investigate the concepts in detail, we have evaluated two particular implementations: cache outcome condition codes and low-overhead cache miss traps. For both implementations, modern processors already contain the bulk of the necessary hardware support; it is used to support branches and exceptions. Nonetheless, selecting a particular implementation depends on both hardware and software trade-offs.

Starting with the hardware trade-offs, we note that cache outcome condition codes have minimal hardware requirements: simply a condition code bit that is set on a cache miss. Despite their simplicity, this mechanism allows lower-overhead responses than current nonarchitected miss counters, since conditionally branching on this code need not serialize the pipeline. On the other hand, even for the more general low-overhead cache miss trap, the hardware requirements are modest. For an in-order-issue machine, the main complexity is getting the trap to logically occur at the correct time. An out-of-order machine can use either its branch or exception mechanism to handle the trap, but may also need to provide guarantees about the cache state. For many applications, these guarantees are not strictly necessary, but when desired, modest hardware changes can prevent unwanted speculative data from entering the primary cache.

From a software perspective, cache outcome condition codes have the disadvantage of requiring either a compiler or a binary editing tool to explicitly insert branch-on-cache-miss instructions into the executable. In contrast, low-overhead cache miss traps can be enabled without any changes to the executable; this would facilitate monitoring commercial software for which source is unavailable, or operating system code where instrumentation may be inconvenient or impossible.

Exposing memory behavior to software is obviously important for performance monitoring. The significance of informing memory operations, however, is that they provide basic primitives which also support a much broader range of applications beyond performance tools. While it may not be cost effective for all commodity microprocessors to include separate support for memory reference counting, multithreading, access control mechanisms, etc., informing memory operations as proposed here provide basic hardware support that is general enough to apply to many such uniprocessor and multiprocessor applications. This generality makes it an attractive feature for future processors, and the availability of informing memory operations in real hardware may spur further innovative uses.

REFERENCES

ABU-SUFAH, W., KUCK, D. J., AND LAWRIE, D. H. 1979. Automatic program transformations for virtual memory computers. In *Proceedings of the 1979 National Computer Conference*, 969–974.

AGARWAL, A., BIANCHINI, R., AND CHAIKEN, D. 1995. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, June 22–24, 1995). ACM Press, New York, NY.

AGARWAL, A., KUBIATOWICZ, J., AND KRANZ, D. 1993. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro 13*, 48–61.

ALVERSON, R., CALLAHAN, D., AND CUMMINGS, D. 1990. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing* (Amsterdam, The Netherlands, June 11–15, 1990). ACM Press, New York, NY, 1–6.

ANDERSON, J. M., BERC, L. M., DEAN, J., GHEMAWAT, S., HENZINGER, M. R., LEUNG, S.-T. A., SITES, R. L., VANDEVOORDE, M. T., WALDSPURGER, C. A., AND WEIHL, W. E. 1997. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst. 15*, 4 (Nov.), 357–390.

BERSHAD, B., LEE, D., ROMER, T. H., AND CHEN, J. B. 1994. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 4–7, 1994). ACM Press, New York, NY, 158–170.

BLUMRICH, M. A., LI, K., ALPERT, R., DUBNICKI, C., FELTEN, E. W., AND SANDBERG, J. 1994. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture* (Chicago, Ill., April 18–21, 1994). IEEE Computer Society Press, Los Alamitos, CA, 142–153.

BURKHART, H. AND MILLEN, R. 1989. Performance-measurement tools in a multiprocessor environment. *IEEE Trans. Comput. 38*, 5 (May), 725–737.

CHANDRA, R., DEVINE, S., VERGHESE, B., GUPTA, A., AND ROSENBLUM, M. 1994. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 4–7, 1994). ACM Press, New York, NY, 12–24.

COOPER, K., HALL, M., AND KENNEDY, K. 1993. A methodology for procedure cloning. *Comput. Lang. 19*, 2 (Apr.).

COVINGTON, R. C., MADALA, S., MEHTA, V., JUMP, J. R., AND SINCLAIR, J. B. 1988. The Rice parallel processing testbed. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Fe, New Mexico, May 24-27, 1988). ACM Press, New York, NY, 4–11.

DEAN, J., HICKS, J., WALDSPURGER, C. A., WEIHL, W., AND CHRYSOS, G. 1997. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of Micro-30*.

DIGITAL EQUIPMENT. 1992. DECChip 21064 RISC microprocessor preliminary data sheet. Tech. Rep., Digital Equipment Corp., Maynard, MA.

DIXIT, K. M. 1992. New CPU benchmark suites from SPEC. In *Proceedings of 37th International Conference on Computer Communications* (San Francisco, CA, Feb. 24–28, 1992). IEEE Computer Society Press, Los Alamitos, CA, 305–310.

DONGARRA, J. J., BREWER, O., KOHL, J. A., AND FINEBERG, S. 1990. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *J. Parallel Distrib. Comput. 9*, 2 (June), 185–202.

EDMONDSON, J. H., RUBINFELD, P. I., BANNON, P. J., BENSCHNEIDER, B. J., BERNSTEIN, D., CASTELINO, R. W., COOPER, E. M., DEVER, D. E., DONCHIN, D. R., FISCHER, T. C., JAIN, A. K., MEHTA, S., MEYER, J. E., PRESTON, R. P., RAJAGOPALAN, V., SOMANATHAN, C., TAYLOR, S. A., AND WOLRICH, G. M. 1995. Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor. *Digital Tech. J. 7*, 1 (Jan.), 119–135.

FARKAS, K. AND JOUPPI, N. 1994. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st International Symposium on Computer Architecture* (Chicago, Ill., April 18–21, 1994). IEEE Computer Society Press, Los Alamitos, CA, 211–222.

FISHER, J. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30*, 7 (July), 478–490.

GALLIVAN, K., JALBY, W., MEIER, U., AND SAMEH, A. 1987. The impact of hierarchical memory systems on linear algebra algorithm design. Tech. Rep. UIUCSRD 625, University of Illinois at Urbana-Champaign, Champaign, IL.

GOLDBERG, A. J. AND HENNESSY, J. L. 1993. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Trans. Parallel Distrib. Syst. 4*, 1 (Jan.), 28–40.

HEINRICH, J. 1995. MIPS R10000 microprocessor user's manual. MIPS Technologies, Inc.

HOROWITZ, M., MARTONOSI, M., MOWRY, T., AND SMITH, M. D. 1995. Informing loads: Enabling software to observe and react to memory behavior. Tech. Rep. CSL-TR-95-602, Computer Systems Laboratory, Stanford University, Stanford, CA.

HOROWITZ, M., MARTONOSI, M., MOWRY, T., AND SMITH, M. D. 1996. Informing memory operations: Providing performance feedback in modern processors. In *Proceedings of the 23rd International Symposium on Computer Architecture* (Philadelphia, PA, May). ACM Press, New York, NY.

JOUPPI, N. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*.

LAM, M., ROTHBERG, E., AND WOLF, M. 1991. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA, Apr. 8–11). ACM Press, New York, NY, 63–74.

LAUDON, J., GUPTA, A., AND HOROWITZ, M. 1994. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 4–7, 1994). ACM Press, New York, NY, 308–318.

LEBECK, A. R. AND WOOD, D. A. 1994. Cache profiling and the SPEC benchmarks: A case study. *Computer 27*, 10 (Oct.), 15–26.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W.-D., GUPTA, A., HENNESSY, J., HOROWITZ, M., AND LAM, M. S. 1992. The Stanford Dash multiprocessor. *Computer 25*, 3 (Mar.), 63–79.

LUK, C.-K. AND MOWRY, T. C. 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 222–233.

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. 1995. Tuning memory performance in sequential and parallel programs. *Computer 28*, 4 (Apr.), 32–40.

MATHISEN, T. 1994. Pentium secrets. *BYTE 19*, 7, 191–192.

MOWRY, T. C. 1995. Tolerating latency through software-controlled data prefetching. Tech. Rep. CSL-TR-94-626, Stanford University, Stanford, CA.

MOWRY, T. C. AND LUK, C.-K. 1997. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of Micro-30*.

MOWRY, T. C., LAM, M. S., AND GUPTA, A. 1992. 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* (Boston, MA, Oct. 12–15). ACM Press, New York, NY, 62–73.

NOWATZYK, A., AYBAY, G., AND BROWNE, M. 1994. The S3.mp scalable shared memory multiprocessor. In *Proceedings of the 27th Hawaiian International Conference on System Sciences.* Vol. 1, *Architecture*. IEEE Computer Society Press, Los Alamitos, CA, 144–153.

PAUL, R. P. 1994. *SPARC Architecture, Assembly Language Programming, and C*. Prentice-Hall, Inc., Upper Saddle River, NJ.

PORTERFIELD, A. K. 1989. Software methods for improvement of cache performance on supercomputer applications. Ph.D thesis, Rice University, Houston, TX.

REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. 1994. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st International Symposium on Computer Architecture* (Chicago, Ill., April 18–21, 1994). IEEE Computer Society Press, Los Alamitos, CA, 325–337.

SCHOINAS, I., FALSAFI, B., LEBECK, A. R., REINHARDT, S. K., LARUS, J. R., AND WOOD, D. A. 1994. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 4–7, 1994). ACM Press, New York, NY, 297–306.

SINGH, J. P., WEBER, W.-D., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News 20*, 1 (Mar.), 5–44.

SINGHAL, A. AND GOLDBERG, A. J. 1994. Architectural support for performance tuning: A case study on the SPARCcenter 2000. In *Proceedings of the 21st International Symposium on Computer Architecture* (Chicago, Ill., April 18–21, 1994). IEEE Computer Society Press, Los Alamitos, CA, 48–59.

SMITH, B. J. 1981. Architecture and applications of the HEP Multiprocessor Computer System. In *SPIE Real-Time Signal Processing IV*. SPIE Press, Bellingham, WA.

SMITH, M. D. 1992. Support for speculative execution in high-performance processors. Ph.D. thesis, Stanford University, Stanford, CA.

THEKKATH, R. AND EGGERS, S. J. 1994. The effectiveness of multiple hardware contexts. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 4–7, 1994). ACM Press, New York, NY, 328–337.

TJIANG, S. W. K. AND HENNESSY, J. L. 1992. Sharlit: A tool for building optimizers. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Francisco, CA, June 17–19), R. L. Wexelblat, Ed. ACM Press, New York, NY.

WOLF, M. E. AND LAM, M. S. 1991. A data locality optimization algorithm. In *Proceedings of the 4th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada, June 26–28). ACM Press, New York, NY, 30–44.

YEAGER, K. C. 1996. The MIPS R10000 superscalar microprocessor. *IEEE Micro 16*, 2 (Apr.), 28–40.

Received: January 1997; revised: February 1998; accepted: February 1998