Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors

Mark Horowitz
Computer Systems Laboratory
Stanford University
horowitz@ee.stanford.edu

Margaret Martonosi
Department of Electrical Engineering
Princeton University
martonosi@princeton.edu

Todd C. Mowry
Department of Electrical and Computer Engineering
University of Toronto
tcm@eecg.toronto.edu

Michael D. Smith
Division of Applied Sciences
Harvard University
smith@eecs.harvard.edu

Abstract

Memory latency is an important bottleneck in system performance that cannot be adequately solved by hardware alone. Several promising software techniques have been shown to address this problem successfully in specific situations. However, the generality of these software approaches has been limited because current architectures do not provide a fine-grained, low-overhead mechanism for observing and reacting to memory behavior directly. To fill this need, we propose 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. We describe two different implementations of informing memory operations—one based on a cache-outcome condition code and another based on low-overhead traps—and 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 look at cache coherence with fine-grained access control as a 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.

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 [Jou90]) 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 [AKL79,GJMS87,WL91] and prefetching [MLG92, Por89] 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 [CMM+88,DBKF90,GH93,LW94,MGA95]. Operating systems have used coarse-grained system information to reduce latencies by adjusting page coloring and migration strategies [BLRC94,CDV+94]. 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 is potentially shared by multiple processors [SFL+94].

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 system observation becoming more important, machine designers are providing at least limited hardware support for it. For example, hardware bus monitors have been used by some applications to gather statistics about references visible on an external bus [CDV+94, SG94, BLRC94, BM89]. 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

In: **Proceedings of the 23rd Annual International Symposium on Computer Architecture.** May, 1996.

cache miss counters, and the MIPS R10000 and Alpha 21064 and 21164 also include memory performance counters [JHei95, DEC92, ERB+95, Mat94]. Compared to bus monitors, on-chip counters allow more fine-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 could normally stall.

There are many alternative methods of achieving informing memory operations, and Section 2 describes three possible approaches: (i) a cache outcome condition code, (ii) a memory operation with a slot that is squashed if the reference hits, and (iii) 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 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*, 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 (evaluated more fully in an earlier study [HMMS95]) 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 (as well as space constraints) 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 simply becomes another condition code. 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 run-time 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 only applies 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 other 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).[1]

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: (i) the memory operation to the memory unit and (ii) 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 can even set the MHAR before each memory operation, invoking a separate handler for each miss, and incurring the one instruction of overhead per memory operation[2]. Instead, 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.

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.

All of the proposed methods have similar performance and hardware costs. The simplest approach adds an instruction to check the status of the previous memory operation (in program order) and incurs an overhead of one instruction in the case of a cache hit. Later in the paper 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 latter mechanism has the advantage that one can monitor whole system behavior without program instrumentation, simply by having the MHAR default to a general handler for all running processes. With cache outcome condition codes, programs must be compiled or instrumented with the new conditional branches to take advantage of the mechanism.

3 Implementation Issues

While the mechanisms proposed are architecturally different, the complexity of each hardware implementation is similar and

1.Alternately, we could define two sets of memory operations—those that trap on a cache miss and those that do not.

2. Actually doing this operation in one instruction is a little tricky. For example one could allow the MHAR to be the destination of an add-immediate instruction, and have a GPR act as the "base" register, but special registers generally cannot be used in ALU instructions. A more plausible solution would be to make the load MHAR instruction have the ability to add a small offset (or the contents of a register) to the PC to generate the desired address. While possible, this latter solution requires additional hardware.

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 to make these issues more clear. As previously mentioned, the low-overhead cache miss trap can be thought of as a load/store followed by an implicit conditional branch-and-link instruction. The destination of the conditional branch is the contents of the Miss Handler Address Register, and the link value is placed in the Miss Handler Return Register. What is encouraging about this proposal (and the other informing memory options) is that most of the necessary hardware mechanisms are the branch and exception mechanisms already present in current machines. To see this in more detail, we discuss two different implementations; we start with the simpler in-order-issue implementation (using the Alpha 21164 as an example) and then describe the implementation for an out-of-order machine (such as MIPS R10000).

3.1 In-order-issue Machines

To be more concrete about the required hardware, this section describes how low-overhead cache miss traps could be added to the 21164 implementation of the Alpha architecture [ERB+95]. The 21164 is a superscalar machine that can execute up to 4 instructions per cycle. Its pipeline is shown in Figure 1; integer operations complete in 6 stages. The machine uses 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 (stage 3) 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 there is no way to stall the already-issued instruction. Instead, the machine takes a replay trap, flushes the pipeline, and then restarts the same instruction. The restarted instruction re-enters the issue stage during stage 11, one cycle before the data is available from the second-level cache.

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 re-issuing 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 non-blocking 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 complexity by describing the implementation of low-overhead traps for informing memory operations in the MIPS R10000. The key problem is keeping track of dependences and ordering, since the order of operations is not determined. 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 only lets instructions execute 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 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 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 only allows three predicted branches in execution at any time. If each reference becomes a potential branch, we will need about 3 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

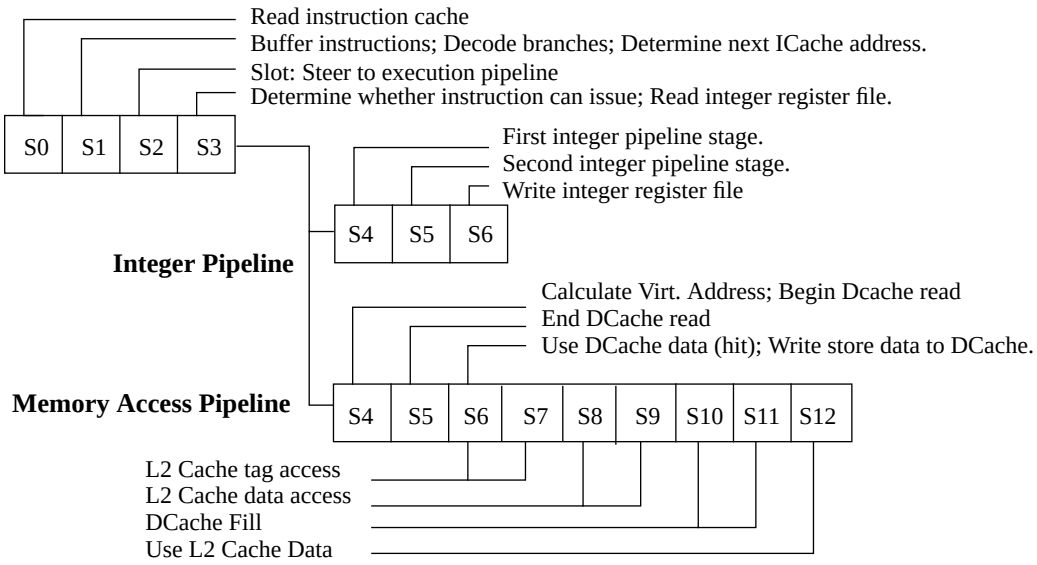

FIGURE 1. Alpha 21164 integer and memory access pipeline stages [ERB+95].

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 4.2, 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

Informing memory operations allow users get information about the state of the cache. This creates a new issue for hardware designers: what guarantees do they make about this state? In current machines, the cache state is not a deterministic function of the program since the cache is not saved and restored during a context switch. If software is only using informing memory operations for performance tuning, this non-determinism is not an issue—the user simply wants to know how the machine is performing at this instance. In contrast, there are other applications of informing memory operations (e.g., cache coherence, as discussed later in Section 4.3) which require that *every* time a new line is fetched into the cache, the system will detect the miss, thus allowing it to check "access rights" to this data. For such applications, the hardware must guarantee that these access checks cannot be bypassed. Unfortunately, today's out-of-order issue machines allow the first-level cache state to be updated speculatively. Since our architectural mechanisms do not execute if a speculative informing memory operation is squashed, 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. It is straightforward to extend our implementation for speculative store problems.

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 signalled 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 [FJ94]. This structure contains the address of the miss, the destination register, 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 registers are 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. In our simulation studies, extending the lifetime of these registers does not change the required number of registers—eight was sufficient in all cases.

4 Uses of Informing Memory Operations

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, and finally we take a detailed look at cache coherence as a case study.

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 (i) the amount of work to be performed in response to a miss and (ii) 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 their impact on performance in Section 4.2.

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 [BM89, GH93, LW94, MGA95]. 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. A previous study demonstrated that informing memory operations can be used to collect precise per-reference miss rates with low runtime overheads (less than 25%) and tolerable data cache perturbations [HMMS95]. This tool uses a single miss handler containing roughly 10 instructions[3] 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

Software-controlled prefetching tolerates memory latency by moving data lines into the cache before they are needed [MLG92, Por89, CMCH91]. 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 [MLG92]. Informing memory operations can address this problem in two ways. The first option is to recompile for a subsequent run based on a detailed memory profile captured from earlier runs. The second option is to generate code capable of adapting its prefetching behavior "on the fly" based on dynamic cache miss information. This latter option can be accomplished either by generating multiple versions of a piece of code (e.g., a loop) with different prefetching strategies and using informing information to select which version to run, or else by placing prefetches directly in the miss handler itself so that prefetching overhead will only be induced when the application is actually suffering from cache misses (and hence prefetches should be beneficial). All of these approaches were evaluated in a previous study and were shown to be useful in improving prefetching performance [HMMS95]. The size of a miss handler for prefetching is likely to be small (less than 10 instructions) since it is either launching a handful of prefetches or else recording some simple statistics. Since we will want to tailor a prefetching response to its context within the program, the miss handler address is likely to change frequently.

4.1.3 Software-Controlled Multithreading

Multithreading tolerates memory latency by switching from one thread (or "context") to another at the start of a cache miss [Smi81, AKK+93, LGH94, TE94, ACC+90]. 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[4], 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 [Pau94]). 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.

4.1.4 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 4.3, but for now, the key parameters are the following: only a single miss handler is required, it is enabled for all potentially-shared references, the handler typically executes 20-30 instructions to check the coherence state, and the dependence chain through these instructions is roughly 10 cycles long.

3. The actual number of dynamic instructions can vary depending on whether the hash probe hits.

4. For example, observing the cache outcome condition code on the secondary cache.

TABLE 1. Simulation parameters for superscalar processors. (These parameters are roughly based on the MIPS R10000 and the 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 Memory	2 INT, 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	2-bit Counters	2-bit Counters

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

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 Table 1. We simulated fourteen SPEC92 benchmarks (five integer and nine floating-point) [Dix92], all of which were compiled with -O2 using the standard MIPS compilers 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 also 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.

4.2.2 Experimental Results

Our results with 1 and 10-instruction miss handlers are shown in Figures 2 and 3. (The su2cor benchmark is shown separately in Figure 3 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.[5] The bottom section is the number of slots where instructions actually graduate,[6] 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 2, we see that for twelve of these thirteen 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 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 instructions 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, cache

5. The number of graduation slots is the issue width (4 for both processors) multiplied by the number of cycles. We focus on graduation rather than issue slots to avoid counting speculative operations that are squashed.
6. This fraction multiplied by the issue width (4) gives us the IPC (instructions per clock) of the machine.

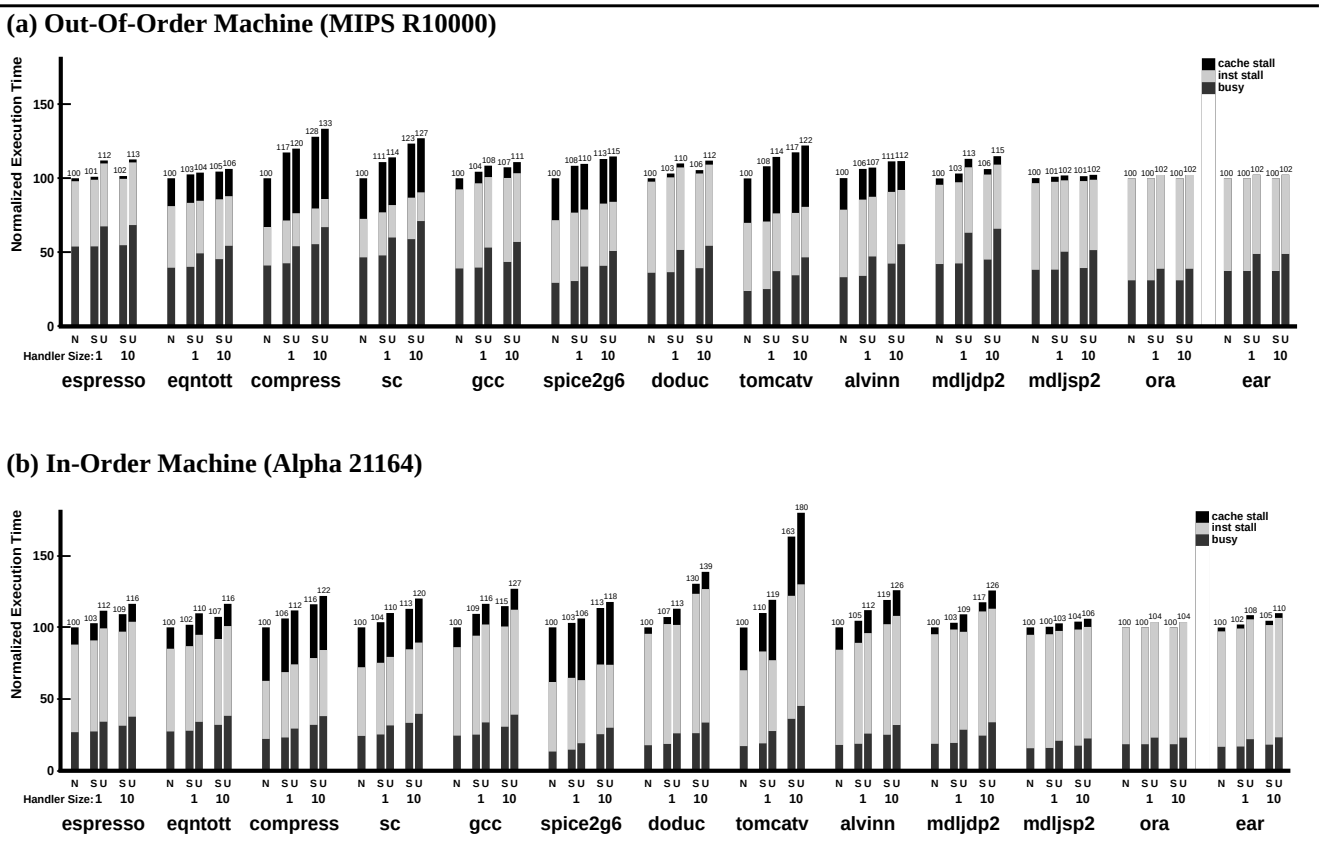

FIGURE 2. Performance of generic miss handlers containing 1 and 10 instructions. (N = no miss handler, S = single miss handler, U = unique miss handler per static reference.)

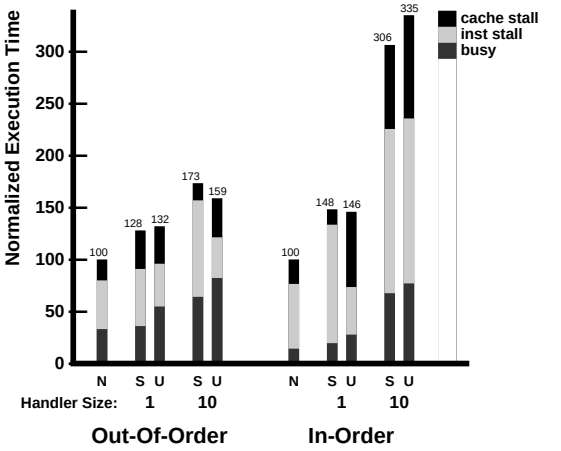

FIGURE 3. Performance of the SU2COR benchmark with generic miss handlers containing 1 and 10 instructions. (N = no miss handler, S = single miss handler, U = unique miss handler per static reference.)

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 vs. 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 makes sense because the high branch prediction accuracies in the floating point benchmarks give the out-of-order machine ample time to overlap miss handler processing.

In contrast to the other thirteen benchmarks, the su2cor benchmark shown in Figure 3 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 3.2). 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.

4.3 Case Study: Cache Coherence with Fine-Grained Access Control

We now take a detailed look at one of the applications mentioned previously: 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 writing these handlers to respond to these memory references with a corresponding action (such as changing a block's state or sending out invalidations). As discussed by Schoinas *et al.* [SFL+94], there are many implementation tradeoffs. In some machines, access control is implemented using bus-snooping hardware, auxiliary protocol processors, or specialized cache or memory controllers [NAB+94, ABC+95]. Using software-based techniques, one can instrument individual memory references with extra code to check protocol state and update it accordingly. Noting the close match between the requirements of fine-grained access control and the features of informing memory operations, this section discusses implementing fine-grained access control using informing memory operations.

4.3.1 Overview of Approach

Our approach is similar to the Blizzard E scheme described by Schoinas *et al.* [SFL+94]. 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 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.)

TABLE 2. 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 cycle 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 + 9 handler cycles to determine if load or store) • 25 cycle state change time

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 (READWRITE for a store, READONLY or READWRITE for a load). If so, the reference continues, with no state changes and no further delay. If the current level is inadequate (INVALID on loads, INVALID or READONLY on stores) protocol operations are needed. 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 [BLA+94].[7]

4.3.2 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 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 2. To give a feel for the relative performance of our access control method, we have also com-

7. User level operations cause DMAs at another processor, and these induce invalidations of data cached at the remote node. Similar functionality could be implemented using a message passing machine and active messages [ECGS92].

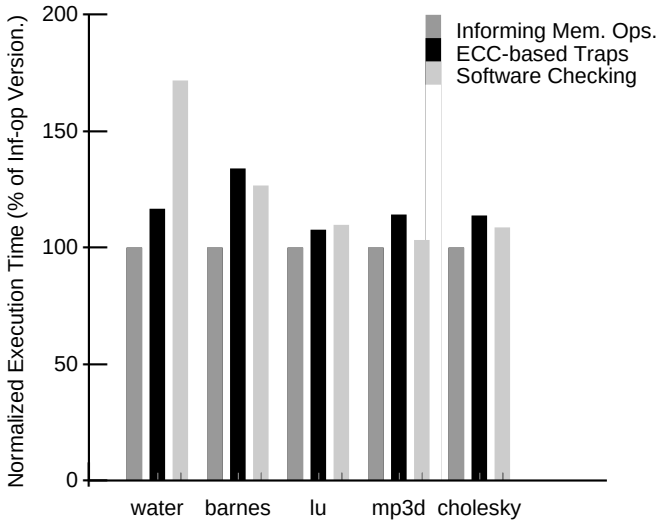

FIGURE 4. Normalized execution times for three access control methods.

pared our approach to two other access control systems: (i) based on per-*reference*-checking (protection lookup on each potentially shared reference) and (ii) based on ECC faults[8]. 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 [SFL+94] and are also listed in Table 2.

We simulated all three access control methods using identical machine assumptions and parameters. Figure 4 shows the performance of an informing-memory-based coherence scheme compared to software-based or trap-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 vs. writes), the informing-op-based approach always out-performs 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 tradeoffs; space constraints prevent us from discussing them in detail. 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.

8. Our informing memory approach does not rely on hardware *specialized* for multi-process access control, and for this reason, we compare ourselves to two published approaches that similarly do not rely on special access control hardware. Systems with hardware support for cache coherence [KOH+94] or access control [RLW94], offer better performance, but at the cost of more (and more specialized) hardware.

5 Conclusions

Informing memory operations are a general primitive for allowing software to observe and react to its own memory referencing behavior. For both of the implementations discussed in detail—cache outcome condition codes and low-overhead cache miss traps—modern processors already contain the bulk of the necessary hardware support; it is used to support branches and exceptions. 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 non-architected miss counters, since conditionally branching on this code need not serialize the pipeline. Even for the more general low-overhead cache miss trap, 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 speculative data from entering the primary cache.

These two mechanisms also offer different tradeoffs on the software side as well. Cache outcome condition codes have simpler hardware requirements, but require executables to be compiled or edited with explicit branch-on-cache-miss instructions to invoke cache miss handlers. Low-overhead cache miss traps, on the other hand, allow for monitoring without special compilers or executable editors; 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.

6 Acknowledgments

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 supported by a Research Grant from the Natural Sciences and Engineering Research Council of Canada. Michael D. Smith is supported by a National Science Foundation Young Investigator award (CCR-9457779).

7 References

[ABC+95] A. Agarwal, R. Bianchini, D. Chaiken, et al. The MIT Alewife Machine: Architecture and Performance. *Proc. 22nd Annual Int'l. Symp. on Computer Architecture.* Jun, 1995.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, et al. The Tera Computer System. *Intl. Conference Supercomputing.* pp 1-6. June, 1990.

[AKK+93] A. Agarwal, J. Kubiatowicz, D. Kranz, et al. Sparcle:

An Evolutionary Processor Design for Large-Scale Multiprocessors. *IEEE Micro*, pp 48-61, June 1993.

[AKL79] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. Automatic Program Transformations for Virtual Memory Computers. *Proc. 1979 National Computer Conf.* pp 969-974, June 1979.

[BLA+94] M. A. Blumrich, K. Li, R. Alpert, et al. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. *Proc. 21st Int'l. Symp. on Computer Architecture.* pp 141-153. Apr., 1994.

[BLRC94] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. *Proc. 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp 158–170, Oct. 1994.

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

[CDV+94] R. Chandra, S. Devine, B. Verghese, et al. Scheduling and page migration for multiprocessor compute servers. *Proc. Sixth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp 12–24, October 1994.

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

[CMM+88] R. C. Covington, S. Madala, V. Mehta, et al. The Rice Parallel Processing Testbed. *Proc. 1988 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems.* pp 4-11. May, 1988.

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

[DBKF90] J. Dongarra, O. Brewer, J. A. Kohl and S. Fineberg. A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors. *Jour. of Parallel and Distributed Computing*, pp 185-202. Jun, 1990.

[DEC92] Digital Equipment Corp. DECChip 21064 RISC Microprocessor Preliminary Data Sheet. Technical report, 1992.

[ECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. *Proc. 19th Annual Int'l. Symp. on Computer Architecture*, pp 256-266, May 1992.

[ERB+95] J. H. Edmonson, P. I. Rubenfeld, P. J. Bannon, et al. Internal Organization of the Alpha 21164, a 300 MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Tech. Journal*, 7(1): 119-135, 1995.

[FJ94] K. Farkas and N. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads, *Proc. 21st Annual Int'l. Symp. on Computer Architecture*. pp 211-222. April, 1994.

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

[GJMS87] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. Technical Report UIUCSRD 625, Univ. of Illinois, 1987.

[HMMS95] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing Loads: Enabling Software to Observe and React to Memory Behavior. Stanford CSL Technical Report CSL-TR-95-673. Stanford University. July 1995.

[JHei95] Joe Heinrich. MIPS R10000 Microprocessor User's Manual. 1995.

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

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, et al. The Stanford FLASH Multiprocessor. *Proc. 21st Annual Int'l. Symposium on Computer Architecture*, pp 302–313, April 1994.

[LGH94] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. *Sixth Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp 308–318, Oct. 1994.

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

[Mat94] Terje Mathison. Pentium Secrets. *Byte*, pp 191-192, July 1994.

[MGA95] M. Martonosi, A. Gupta, and T. E. Anderson. Tuning Memory Performance of Sequential and Parallel Programs. *IEEE Computer*, pp 32-40. April 1995.

[MLG92] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *Proc. 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pp 62–73, Oct. 1992.

[NAB+94] A. Nowatzyk, G. Aybay, M. Browne, et al. The S3.mp Scalable Shared Memory Multiprocessor. *Proc. 27th Hawaii Intl. Conf. on System Sciences Vol. I: Architecture.* pp 144-53. Jan, 1994.

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

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

[RLW94] S. K. Reinhardt, J. R. Larus and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. *Proc. 21st Int'l. Symp. on Computer Architecture.* pp 325-337. Apr., 1994.

[SFL+94] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-Grain Access Control for Distributed Shared Memory. *Proc 6th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*. pp 297-306. Oct. 1994.

[SG94] Ashok Singhal and Aaron J. Goldberg. Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000. *Proc. 21st Annual Int'l. Symp. on Computer Architecture*. pp 325-337. April, 1994.

[Smi81] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE Real-Time Signal Processing IV*, Vol. 298, 1981.

[TE94] R. Thekkath and S. J. Eggers. The Effectiveness of Multiple Hardware Contexts. *6th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp 328-337, Oct.1994.

[TL94] C. A. Thekkath and H. M. Levy. Hardware and Software Support for Efficient Exception Handling. *Proc 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. pp 110-119. Oct. 1994.

[WL91] M. E. Wolf and M. S. Lam. A Data Locality Optimization Algorithm. Proc. SIGPLAN '91 Conf. on Programming Language Design and Implementation, pp 30-44, June 1991.