Active Memory: A New Abstraction for Memory System Simulation[1]

Alvin R. Lebeck
Computer Science Department
Duke University
Durham, NC 27708
alvy@cs.duke.edu
http://www.cs.duke.edu/~alvy

David A. Wood
Computer Sciences Department
University of Wisconsin—Madison
Madison, WI 53706
david@cs.wisc.edu
http://www.cs.wisc.edu/~david

Abstract

This paper describes the *active memory* abstraction for memory-system simulation. In this abstraction—designed specifically for on-the-fly simulation—memory references logically invoke a user-specified function depending upon the reference's type and accessed memory block state. Active memory allows simulator writers to specify the appropriate action on each reference, including "no action" for the common case of cache hits. Because the abstraction hides implementation details, implementations can be carefully tuned for particular platforms, permitting much more efficient on-the-fly simulation than the traditional trace-driven abstraction.

Our SPARC implementation, *Fast-Cache*, executes simple data cache simulation 2 to 6 times slower than the original, uninstrumented program on a SPARCstation 10; a procedure call based trace-driven simulator is 7 to 16 times slower than the original program, and a trace-driven simulator that buffers references in memory to amortize procedure call overhead is 3 to 8 times slower. Fast-Cache implements active memory by performing a fast table look up of the memory block state, taking as few as 3 cycles on a SuperSPARC for the no-action case. Modeling the effects of Fast-Cache's additional lookup instructions qualitatively shows that Fast-Cache is likely to be the most efficient simulator for miss ratios between 3% and 40%.

General Terms

Measurement, Performance

Subject Descriptors

B.3.3 [Memory Structures]; simulation, B.3.2 [Memory Structures]; Cache Memories, C.4 [Performance of Systems]; Measurement Techniques

Keywords

Cache, cache memory, memory hierarchy, on-the-fly simulation, trace-driven simulation, direct-execution simulation

1 Introduction

Simulation is the most-widely-used method to evaluate memory-system performance. However, current simulation techniques are discouragingly slow; simulation times can be as much as two or three orders of magnitude slower than the execution time of the original program. Gee, et al. [6], estimate that *17 months* of processing time were used to obtain miss ratios for the SPEC92 benchmarks [28].

Fortunately, simulation times can be reduced using a new simulation abstraction. The traditional approach—trace-driven simulation—employs a *reference trace* abstraction: a reference generator produces a list of memory addresses that the program references and is processed by the simulator (see Figure 1). This abstraction hides the details of reference generation

1. This paper is an extended version of a previous publication [14]. The extensions include additional analysis, and details about simulator implementation.

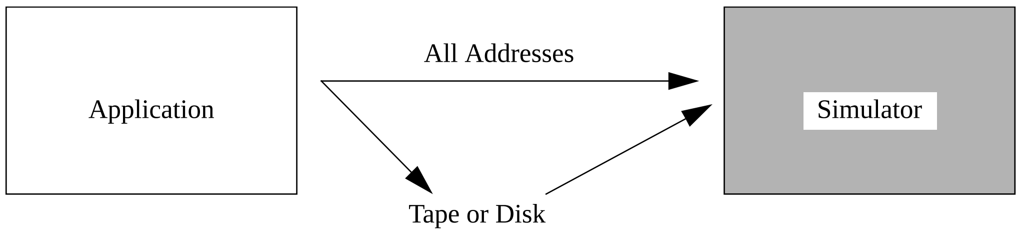


Figure 1: Trace-Driven Simulator

from the simulator, but introduces significant overhead (10-21 processor cycles on a SuperSPARC processor) that is wasted in the common case, e.g., a cache hit, in which the simulator takes no action on the reference. In the Gee, et al., study, 90% of the references required no simulator action for a 16 kilobyte cache.

This paper examines *active memory*, a new memory system simulation abstraction designed specifically for on-the-fly simulators that process memory references as the application executes. Active memory, described in Section 3, provides a clean interface that hides implementation details from the simulator writer, but allows a tight coupling between reference generation and simulation. In this abstraction, each memory reference logically invokes a user-specified function depending upon the reference's type and the current state of the accessed memory block. Simulators control which function is invoked by manipulating the states of the memory block. The abstraction provides a predefined function (NULL) that simulator writers can specify for the common, no-action case. Active memory implementations can optimize this NULL function depending on available system features (e.g., in-line software checks, or error correcting code (ECC) bits and fast traps.)

Consider an active memory simulator that counts cache misses. It can represent blocks that are present in the cache as *valid*, and all others as *invalid*. References to *valid* blocks invoke the predefined NULL handler, while references to *invalid* blocks invoke a user-written *miss* handler. The miss handler counts the miss, selects a victim, and updates the state of both the replaced and referenced blocks. Multiple alternative caches can be simulated by only marking blocks *valid* if they are present in all caches. Since most references are to *valid* blocks, an active memory implementation with an optimized NULL handler (3 cycles for the Fast-Cache system described below) could allow an active memory simulator to execute much faster than a highly-optimized implementation of the traditional trace abstraction (>= 10 cycles for the no-action case).

We have implemented active memory in the *Fast-Cache* simulation system, which eliminates unnecessary instructions in the common no-action case. Measurements on a SPARCstation 10/51 show that simple data-cache simulations run only 2 to 6 times slower than the original program. This is comparable to many execution-time profilers and two to three times faster than published numbers for highly optimized trace-driven simulators [29].

As described in Section 4, Fast-Cache efficiently implements this abstraction by inserting 9 SPARC instructions before each memory reference to look up a memory block's state and invoke the user-specified handler. If the lookup invokes the NULL handler, only 5 of these instructions actually execute, completing in as few as 3 cycles (assuming no cache misses) on a SuperSPARC processor.

Section 5 analyzes the performance of Fast-Cache by modeling the effects of the additional lookup instructions. We use this simple model to qualitatively show that Fast-Cache is more efficient than simulators that use hardware support to optimize no action cases—unless the simulated miss ratio is very small (e.g., less than 3%). Similarly, we show that Fast-Cache is more efficient than trace-driven simulation except when the miss ratio is very large (e.g., greater than 20%). These results indicate that Fast-Cache is likely to be the fastest simulation technique over much of the interesting cache memory design space.

Section 6 extends this model by incorporating the cache pollution caused by the additional instructions inserted by Fast-Cache. For data caches, we use an approximate bounds analysis to show that—for the Fast-Cache measurements on the SPARCstation 10—data cache pollution introduces at most a factor of 4 slowdown (over the original program). A simple estimator—that splits the difference between the two bounds—predicts the actual performance within 30%. For instruction caches, we show that the instrumented codes are likely to incur at least 8 times as many instruction misses as the original code. For most of the applications, the SuperSPARC first-level instruction cache miss ratios were so small, that this large increase had no appreciable effect on execution time. However, one program with a relatively large instruction cache miss ratio incurs noticeable additional slowdowns. To address this problem, we present an alternative implementation, Fast-Cache-Indirect, that reduces code dilation to 2 static instructions at the expense of 3 more executed instructions for the “no action” case.

Section 7 discusses how to use the active memory abstraction for simulations more complex than simple miss counting, and Section 8 concludes this paper.

2 Background

Memory-system simulation is conceptually simple. For each memory reference issued by the processor, the system must:

1. compute the effective address
2. look up the action required for that reference
3. simulate the action, if any.

Traditionally, the first step was considered difficult and inefficient, usually requiring either expensive hardware monitors or slow instruction-level simulators [8]. The reference trace abstraction helped amortize this overhead by cleanly separating reference generation (step 1) from simulation (steps 2–3). As illustrated in Figure 1, reference traces can be saved and reused for multiple simulations, with the added benefit of guaranteeing reproducible results [1,11].

Many techniques have been developed to improve trace-driven simulation time by reducing the size of reference traces. Some accomplish this by filtering out references that would hit in the simulated cache. Smith [26] proposed deleting references to the *n* most recently used blocks. The subsequent trace can be used to obtain approximate miss counts for fully associative memories that use LRU replacement with more than *n* blocks. Puzak [21] extended this work to set-associative memories by filtering references to a direct-mapped cache.

However, software reference generation techniques have improved to the point that regenerating the trace is nearly as efficient as reading it from disk or tape [11]. On-the-fly simulation techniques—which combine steps 1–3—have become popular because they eliminate I/O overhead, context switches, and large storage requirements [5,20,4,3,2].

Most on-the-fly simulation systems work by instrumenting a program to calculate each reference’s effective address and then invoke the simulator (see Figure 2). For typical RISC instruction sets, the effective address calculation is trivial, requiring at most one additional instruction per reference. Unfortunately, most on-the-fly simulation systems continue to use the reference trace abstraction. Although simple, this abstraction requires that the simulator either (i) perform a procedure call to process each reference, with the commensurate overhead to save and restore registers [5,20], or (ii) buffer the reference in

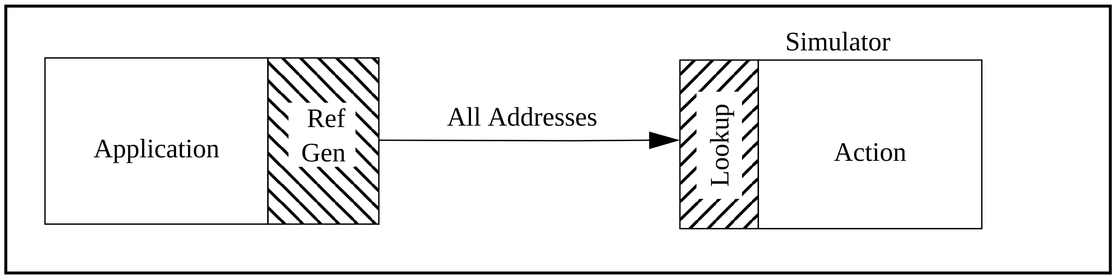


Figure 2: On-The-Fly Simulator

memory, incurring buffer management overhead and memory system delays caused by cache pollution [3,30]. Furthermore, this overhead is almost always wasted, because in most simulations the common case requires no action. For example, no action is required for cache hits in direct-mapped caches or many set-associative caches with random replacement. Similarly, no action is required for references to the most recently used (MRU) block in each set for set-associative caches with least recently used (LRU) replacement.

Clearly, optimizing the lookup (step 2) to quickly detect these “no action” cases can significantly improve simulation performance. MemSpy [16] builds on this observation by saving only the registers necessary to determine if a reference is a hit or a miss; hits branch around the remaining register saves and miss processing. MemSpy’s optimization improves performance but sacrifices trace-driven simulation’s clean abstraction. The action lookup code must be written in assembly language, so the appropriate registers may be saved, and must be modified for each different memory system. The ATOM cache simulator performs a similar optimization more cleanly, using the OM liveness analysis to detect, and save, caller-save registers used in the simulator routines [29]. However, ATOM still incurs unnecessary procedure linkage overhead in the no-action cases.

A recent alternative technique, *trap-driven simulation* [23,34], optimizes “no action” cases to their logical extreme. Trap-driven simulators exploit the characteristics of the simulation platform to implement effective address calculation and lookup (steps 1 and 2) in hardware using error correcting code (ECC) bits [23] or valid bits in the TLB [19]. References requiring no action run at full hardware speed; other references cause memory system exceptions that invoke simulation software. By executing most references without software intervention, these simulators potentially perform much better than other simulation systems.

Unfortunately, trap-driven simulation lacks the portability and generality provided by trace-driven simulation. Portability suffers because these simulators require operating system and hardware support that is not readily available on most machines. Generality is lacking because current trap-driven simulators do not simulate arbitrary memory systems: the Wisconsin Wind Tunnel [23] does not simulate stack references because of SPARC register windows, while Tapeworm II [34] does not simulate any data references because of write buffers on the DECstation. Furthermore, as we show in Section 5, the overhead of

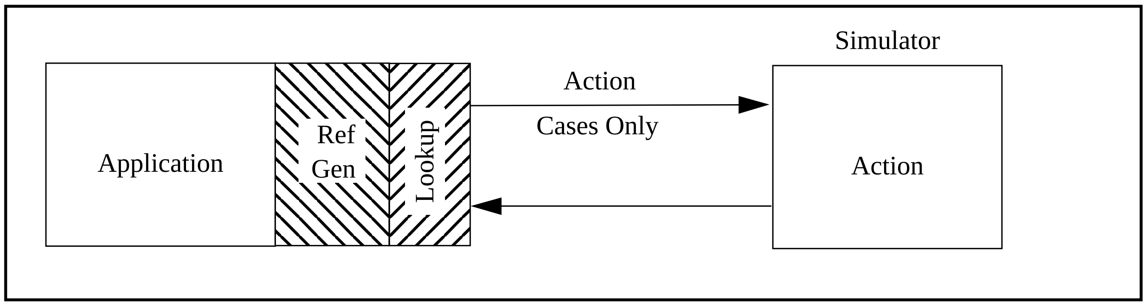


Figure 3: Active Memory Simulator

memory exceptions (roughly 250 cycles [34,33,22] on well tuned systems) can overwhelm the benefits of "free" lookups for simulations with non-negligible miss ratios.

The *active memory* abstraction—described in detail in the next section—combines some of the efficiency of trap-driven simulation with the generality and portability of trace-driven simulation. The central idea is to provide a clean abstraction between steps 1–2 and step 3. Combining effective address generation and action lookup allows the simulation system to implement the no-action cases without unnecessary overhead; only those references requiring action incur the procedure call overhead of invoking the simulator (see Figure 3.) The active memory abstraction hides the implementation of steps 1–2 from the simulator, allowing a variety of implementations for these two steps and allowing the simulator to be written in a high-level language.

The next section describes the active memory abstraction in detail. Section 4 describes our implementation for the SPARC architecture.

3 Active Memory

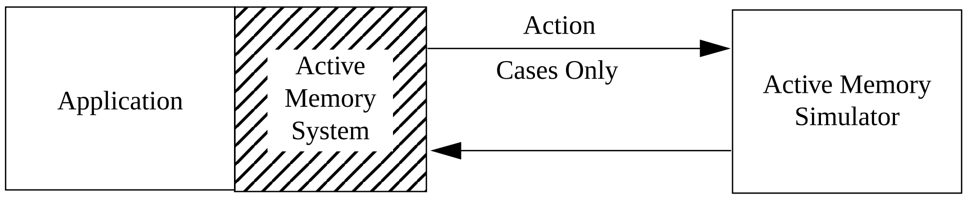
In the active memory abstraction, each memory reference conceptually invokes a user-specified function, called a *handler.* Memory is logically partitioned into aligned, fixed-size (power of two) blocks, each with a user-defined state. Users—i.e., simulator writers—specify which function gets invoked for each combination of reference type (`load` or `store`) and memory block state. A simulator is simply a set of handlers that control reference processing by manipulating memory block states, using the interface summarized in Table 1. This interface defines the active memory abstraction.

Active Memory Run-Time System Provided	
read_state(address)	Return block state.
write_state(address,state)	Update block state.
User Written (Simulator Functions)	
user_handler(address,pc)	Multiple handlers invoked for action. Separate handlers for loads and stores.
sim_init(argc,argv)	Simulator start-up routine

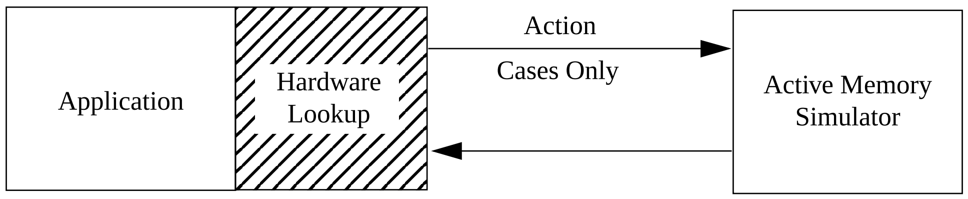
TABLE 1. Active Memory Interface

sim_exit()	Simulator exit routine

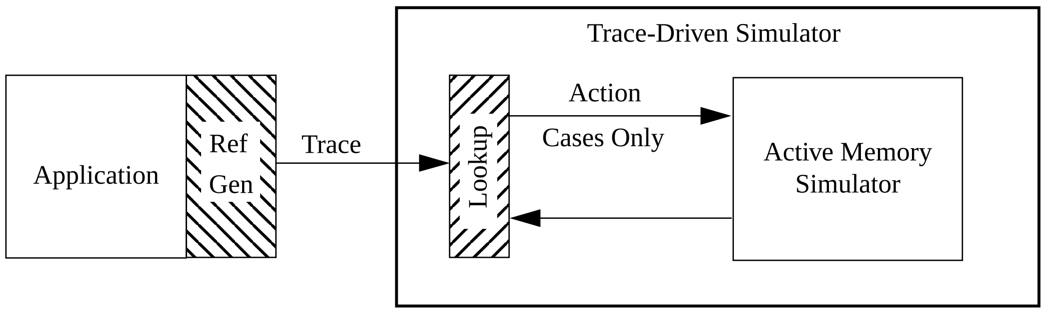
TABLE 1. Active Memory Interface



a) Native Active Memory System



b) Trap-Driven Implementation



c) Trace-Driven Implementation

Figure 4: Active Memory Implementations

From the simulator writer's perspective, a simulation begins with invocation of the sim_init routine, to set up the data structures needed by the handlers. After the sim_init routine returns the application begins execution, and memory references invoke handlers according to the memory block state and the type of memory reference. The `sim_exit` routine is invoked when the application terminates, allowing output of final simulation results.

Users can identify cases that do not require simulator action by specifying the predefined NULL handler. Making this case explicit allows the active memory system to implement this case as efficiently as possible, without breaking the abstraction. The active memory abstraction could be encapsulated in a trace-driven simulator (see Figure 4c). However, eliminating the reference trace abstraction and directly implementing active memory in on-the-fly simulators allows optimization of the NULL handler. While this paper focuses on software implementations, active memory can also be supported using the same hardware required for trap-driven simulations (see Figure 4b).

```
/* Active Memory configuration for a simple cache simulation */
lg2blocksize 5                          /* log base 2 of the blocksize */

LOADS                                   /* Indicates start of handlers for LOADs */
invalid    miss_handler                 /* user handler to call */
valid      noaction                     /* predefined NULL handler */

STORES                                  /* Indicates start of handlers for STOREs */
invalid    miss_handler                 /* user handler to call */
valid      noaction                     /* predefined NULL handler */

/* Simple Active Memory Handler (pseudo-code) */
sim_init()
{
        miss_count = 0;
        initalize_cache();
}

miss_handler (Addr address)
{
        miss_count++;
        victim_address = select_victim(address);
        write_state(address,valid);
        write_state(victim_address,invalid);
}

sim_exit()
{
        printf("miss count: %d\n",miss_count);
}
```

Figure 5: Simple Data-Cache Simulator Using Active Memory

The example in Figure 5 illustrates how to use active memory to implement a simple data-cache simulation that counts cache misses (more complex simulations are discussed in Section 7). The user specifies the cache block size (2^5 = 32 bytes) and the functions to be invoked on each combination of reference and state; e.g., a `load` to an `invalid` block invokes the `miss_handler` routine. The function `noaction` is the predefined NULL handler. At start up, the `sim_init` routine clears the cache miss counter and initializes the cache data structure (not shown). The simple handler, `miss_handler`, increments the miss count, selects a victim block using a user-written routine (not shown), and then marks the victim block state `invalid` and the referenced block state `valid`. The user-supplied termination routine `sim_exit` prints the number of misses at the end of the target program. Note that the simulator is written entirely in user-level code in a high-level language.

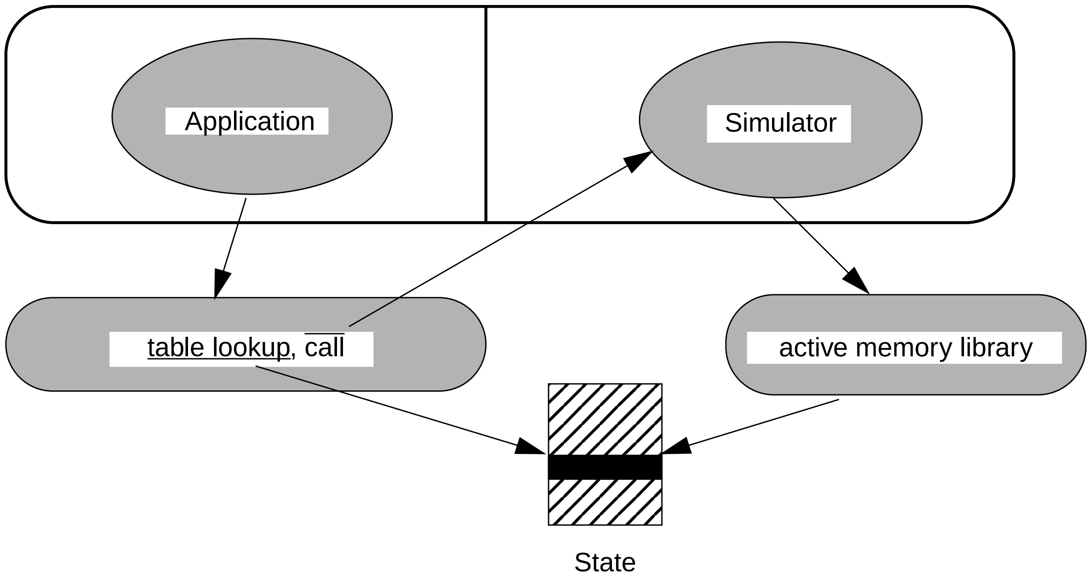


Figure 6: Fast-Cache Implementation

4 Fast-Cache

This section describes Fast-Cache, our implementation of the active memory abstraction for SPARC processors. Active memory allows Fast-Cache to provide an efficient, yet general simulation framework by: (i) optimizing cases that do not require simulator action, (ii) rapidly invoking specific simulator functions when action is required, (iii) isolating simulator writers from the details of reference generation, and (iv) providing simulator portability.

Conceptually, the active memory abstraction requires a large table to maintain the state of each block of memory. Before each reference, Fast-Cache checks a block's state by using the effective address as an index into this table and invokes an action only if necessary (see Figure 6). Fast-Cache allocates a byte of state per block, thus avoiding bit-shifting, and uses the UNIX *signal* and *mmap* facilities to dynamically allocate only the necessary portions of the state table.

Fast-Cache achieves its efficiency by inserting a fast, in-line table lookup before each memory reference. The inserted code (see Figure 7) computes the effective address, accesses the corresponding state, tests the state to determine if action is required, and invokes the user-written handler if necessary. The SPARC instruction set requires one instruction to compute the effective address: a single `add` instruction to compute base plus offset. This instruction could be eliminated in the case of a zero offset; however, we do not currently implement this optimization. An additional instruction shifts the effective address to a table offset. By storing the base of the state table in an otherwise unused global register, a third instruction suffices to load the state byte.[1] Since the memory block state indicates what, if any, action is required, these three instructions implement steps 1–2 in the taxonomy of Section 2. We can avoid using the reserved registers by scavenging temporarily unused registers and add a single instruction to set the table base. This additional instruction would not add any additional cycles to the lookup on

1. Register `%g5`, `%g6`, and `%g7` are specified as reserved in the SPARC v8 Application Binary Interface.

```
Cycle       Instruction                           Comments
0       add %g0, %g0, %g5                ! get the effective address %g0 is place holder
            ! split cascade into shift
1       sra %g5, LG2BLKSIZE, %g6         ! calculate block byte-index
            ! split ALUOP into LD
2       ldub [%g7 + %g6], %g6            ! load block state byte
            ! split load data use
3       andcc %g6, mask, %g0             ! check the right bit, mask set to correct
                                             mask
        bne 1f
        LD or ST                         ! the memory ref goes here
        sll %g6, LG2STUBSIZE, %g6        ! shift by stub size
        sethi HANDLER_TBL_BASE, %g7      ! set the stub base pointer
        jmpl %g7 + %g6, %g6              ! jump to handler stub
        sethi %hi(TBL_BASE), %g7         ! restore the state table pointer
    1:                                   ! next application instruction
```

The Fast-Cache lookup snippet requires 3 cycles when condition codes are not live, assuming the first instruction can be issued with previous instruction of application. If the first instruction can not be issued with the previous application instruction, then 4 cycles are required. In the sequence above `%g0` is a place holder for register specifier and/or immediate operands of the specific memory reference.

Figure 7: Fast-Cache Lookup with Dead Condition Codes

the SuperSPARC processor, since it could be issued in the same cycle as the effective address computation (`add`) or state table index computation (`shift`).

The code inserted to test the state and determine whether an action is required, depends on whether the condition codes are live (i.e., contains a value that will be used by a subsequent branch instruction). The SPARC architecture has a single set of condition codes which are optionally set as a side-effect of most ALU instructions. Unfortunately, the SPARC v8 architecture does not provide a simple and efficient way to save and restore the condition codes in user mode.[1] Thus, Fast-Cache generates two different test sequences depending upon whether the condition codes are live or not.

In the common case (50%-98%), the condition codes are dead, and Fast-Cache uses a simple two instruction sequence that masks out the appropriate bits and branches (loads and stores must check different state bits.) We expect the common case to be no action, so the branch target is the next instruction in the original program. If an action is required, the branch falls through into a four instruction “trampoline” that jumps to the handler stub. Since we schedule the memory reference in the delay slot of the branch, the critical no-action path requires 5 instructions for a total of 3 cycles on the SuperSPARC (4 cycles if the effective address calculation cannot be issued with the preceding instruction). These numbers are approximate, of course, since inserting additional instructions may introduce or eliminate pipeline interlocks and affect the superscalar issue rate [35]. This sequence could be further optimized on the SuperSPARC by scheduling independent instructions from the original program with the Fast-Cache inserted instructions.

1. The SPARC v9 instruction set allows saving and restoring condition codes in user mode.

```
Cycle       Instruction                     Comments
0       add %g0, %g0, %g5               ! get the effective address, %g0 is place holder
            ! split cascade into shift
1       sra %g5, LG2BLKSIZE, %g6        ! calculate block byte-index
            ! split ALUOP into LD
2       ldub [%g7 + %g6], %g6           ! load block state byte
            ! split load data use
3       sll %g6, LG2STUBSIZE, %g6       ! shift by stub size
        sethi HANDLER_TBL_BASE, %g7     ! set the tbl ptr
            ! split before cascade into jmpl
4       jmpl %g7 + %g6, %g6             ! jump to handler jump table
            ! split after control transfer
5       sethi %hi(TBL_BASE), %g7        ! restore the state table ptr
```

The Fast-Cache lookup snippet requires 7 cycles when condition codes are live, assuming the first instruction can be issued with the previous instruction of application. The in-line sequence (shown above) takes 5 cycles , and an additional 2 cycles are required for the NULL handler (`ret` & `nop` instructions). In the sequence above `%g0` is a place holder for register specifier and/or immediate operands of the specific memory reference.

Figure 8: Fast-Cache Lookup with Live Condition Codes

If the condition codes are live, we cannot use a branch instruction. Instead, we use the block state to calculate the address of a handler stub and perform a procedure call (see Figure 8). No action cases invoke a NULL handler (literally a `return` and a `nop`), which requires 9 instructions, taking 7 cycles on the SuperSPARC.

When action is required, Fast-Cache invokes user handlers through a stub that saves processor state (see Appendix A). Most of the registers are saved in the normal way using the SPARC register windows. However the stub must save the condition codes, if live, and some of the global registers because the simulator handlers and the application are created through separate compilation.

The table lookup instructions could be inserted with any instrumentation methodology. Fast-Cache uses the EEL system [12], which takes an executable SPARC binary file, adds instrumentation code, and produces an executable that runs on the same machine. Fast-Cache minimizes perturbation by providing a separate data segment and library routines for the simulator.

5 Qualitative Analysis

In this section we use a simple model to qualitatively compare the performance of Fast-Cache to trace-driven and trap-driven simulators. In Section 6, we extend this model to incorporate cache interference effects and use it to analyze the performance of Fast-Cache in more detail.

For the comparison in this section, we focus on a simple miss-count simulation for direct-mapped data caches with 32-byte blocks—called the *target* cache. To simplify the discussion, we lump effective address calculation and action lookup into a single *lookup* term. Similarly, we lump action simulation and metric update into a single *miss processing* term.

```
Cycle      Instruction                      Comments
1       add %g6, 0x4, %g6           ! increment buf_ptr
        add %g0, %g0, %g5           ! get the effective address
            ! split--out of register write ports
2       cmp %g6, %g7                ! check if buffer full
        ble 1f                      ! branch if not full
            ! split after control transfer
3       st %g5, [%g6]               ! store it in the buffer
            ! split after delay slot instruction
        jmpl %g7+0x8, %g6           ! jump to handler jump table
        nop                         ! in case ref is in delay slot of call
      1:                            ! the memory ref goes here
```

Figure 9: Buffered Implementation With Dead Condition Codes

For trace-driven simulation, we consider two on-the-fly simulators: one invokes the simulator for each memory reference (via procedure call) [29,16], and one buffers effective addresses, invoking the simulator only when the buffer is full. To maintain a clean interface between the reference generator and the simulator, processor state is saved before invoking the simulator.

The procedure call implementation inserts two instructions before each memory reference that compute the effective address and jumps to a stub; the stub saves processor state, calls the simulator, then restores the state. The stub uses the SPARC register windows to save most of the state with a single instruction, but must explicitly save several global registers and the condition codes, if live. Since saving and restoring condition codes takes multiple instructions on SPARC, our implementation jumps to a separate streamlined stub when they are dead (see Appendix A). On a SuperSPARC processor, the lookup overhead is roughly 21 cycles when we can use the streamlined stub. Most of this overhead is the procedure call linkage, which could be reduced using techniques similar to ATOM's that saves and restores only the necessary registers. The actual lookup for a direct-mapped cache is little more than the shift-load-mask-compare sequence used by Fast-Cache. When a target miss does occur, the additional overhead for miss processing is very low, 3 cycles, because the lookup has already found the appropriate entry in the cache data structure. Because trace-driven simulation incurs a large lookup overhead, performance will depend primarily on the fraction of instructions that are memory references. Conversely, because the miss processing overhead is so low, it is almost independent from the target cache miss ratio.

The buffered implementation inserts 7 instructions before each memory reference, assuming condition codes are dead (see Figure 9, Appendix A shows the instructions used when condition codes are live). Only 5 of these instructions are required to store an entry in the buffer, and they execute in only 3 cycles on a SuperSPARC (assuming no cache misses). These five instructions compute the effective address, store it in the buffer, increment the buffer pointer, compare the buffer pointer to the end of the buffer, and branch if the buffer is not full. The fall through of the branch (the remaining two instructions) is a procedure call to the simulator routine that processes the entries in the buffer. Reading an entry from the buffer and checking the cache data structure requires 7 cycles with an additional 2 cycles for a target cache miss. The overhead of invoking the simulator is amortized over 1024 memory references, essentially eliminating it from the lookup overhead, resulting in a total of 10 cycles to perform the lookup.

An alternative implementation could use a signal handler that is invoked when the buffer is full. However, this approach would eliminate only one cycle from the lookup and incur significantly larger overhead when the buffer is full. Although the additional overhead could be amortized by storing more references, the larger buffer is likely to increase the amount of cache pollution, and reduce the potential benefits.

Like the procedure call implementation, we expect the buffered implementation to be mostly dependent on the fraction of instructions that are memory references with very little dependence on the miss ratio. However, this technique should be significantly faster, since it has one half the lookup overhead per reference.

Trap-driven simulators represent the other extreme, incurring no overhead for cache hits. Unfortunately, target cache misses cause memory system exceptions that invoke the kernel, resulting in miss processing overhead of approximately 250 cycles on highly tuned systems [34,33,22]. Therefore, trap-driven simulation performance will be highly dependent on the target miss ratio. It will exceed the performance of alternative simulators only for sufficiently low miss ratios.

For typical programs, only a small fraction of memory references occur when condition codes are live. Given this, and the uncertainty of the exact schedule for the lookup snippet (3 or 4 cycles), we assume that Fast-Cache's lookup overhead is 4 cycles. However, the miss processing overhead, roughly 31 cycles, is higher than a trace-driven simulator because the memory block states must be updated in addition to the regular cache data structures. Thus, Fast-Cache's simulation time depends on both the fraction of instructions that are memory references and the target miss ratio. Table 2 summarizes this comparison and the overhead for the various simulators.

Method	**Lookup**	**Miss processing**	**Dependence on fraction of references**	**Dependence on miss ratio**
Procedure	21	3	High	Low
Buffered	10	2	High	Low
Trap-Driven	0	250	Low	High
Fast-Cache	4	31	Moderate	Moderate

TABLE 2. Simulator Overhead

We can construct a simple model of simulation time by calculating the cycles required to execute the additional simulation instructions. This model ignores cache pollution on the host machine, which can be significant, but Section 6 extends the model to include these effects. Our model estimates *slowdown*, the simulation time divided by the execution time of the original, un-instrumented program. Ignoring cache effects, the slowdown is the number of cycles for the original program, plus the number of instruction cycles required to perform the lookups and miss processing, divided by the number of cycles for the original program:

$$\text{Slowdown} = 1 + \frac{(r \cdot I_{\text{orig}} \cdot C_{\text{lookup}})}{C_{\text{orig}}} + \frac{(r \cdot I_{\text{orig}} \cdot m \cdot C_{\text{miss}})}{C_{\text{orig}}} \quad (1)$$

The first term is simply the normalized execution time of the original program. The second term is the number of cycles to perform all lookups, where C_{lookup} is the overhead of a single lookup, divided by the number of cycles for the original program, C_{orig}. Since these are data-cache simulations, the lookup is performed only on the $r \cdot I_{\mathrm{orig}}$ data references, where r is the fraction of instructions that are memory references, and I_{orig} is the number of instructions in the original program.

The numerator of the last term is cycles to process all target cache misses. The number of misses for a given program is easily measured by running one of the simulators. Alternatively, we express it as a function of the target cache miss ratio, m, multiplied by the number of memory references, $r \cdot I_{\mathrm{orig}}$. Each target cache miss incurs a simulation overhead of C_{miss} cycles.

We can simplify Equation 1 and express the slowdown as a function of the target miss ratio m:

$$\text{Slowdown} = 1 + \frac{r}{\mathrm{CPI}_{\mathrm{orig}}}(C_{\mathrm{lookup}} + m \cdot C_{\mathrm{miss}}) \qquad (2)$$

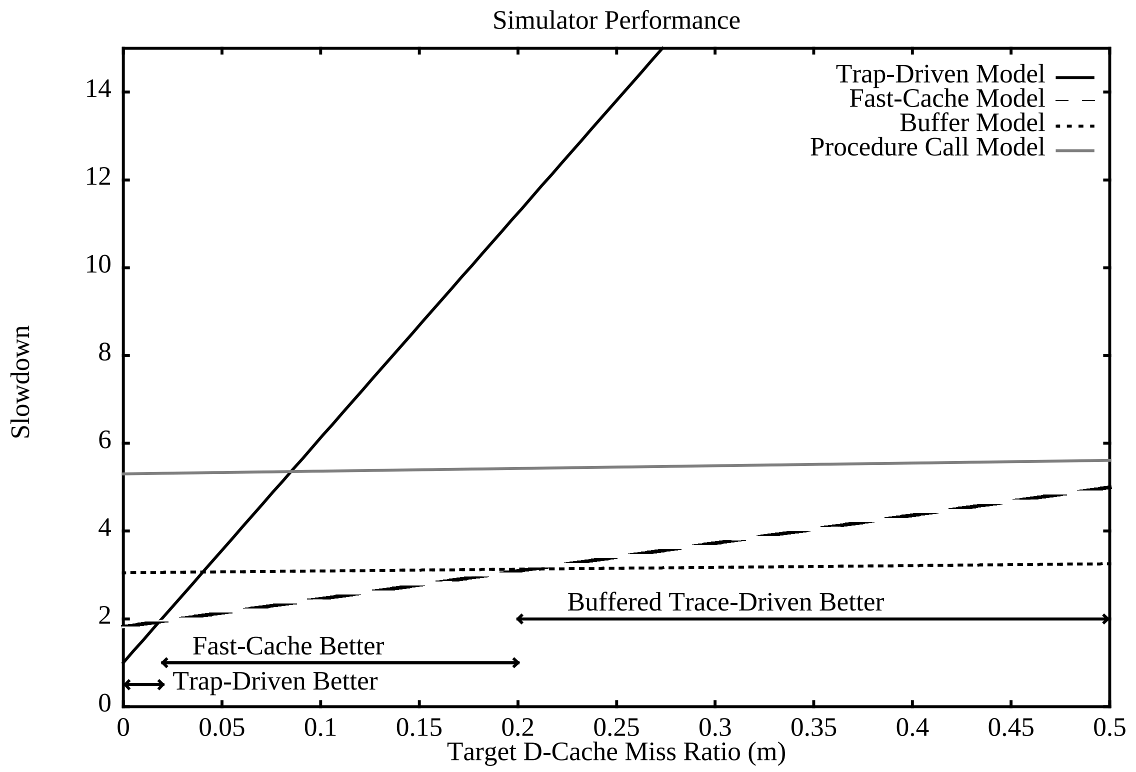
where $\mathrm{CPI}_{\mathrm{orig}}$ is cycles-per-instruction, $\frac{C_{\mathrm{orig}}}{I_{\mathrm{orig}}}$.

We can use Equation 2 to get a rough idea of the relative performance of the various simulation techniques. Figure 10 shows simulator slowdown versus target miss ratio, using a $\mathrm{CPI}_{\mathrm{orig}}$ of 1.22 and reference ratio $r = 0.25$ (derived from the SPEC92 benchmark program `compress` [28]). The simulator parameters are miss processing overhead, C_{miss}, of 250 for trap-driven, 3 for procedure call, 2 for buffered, and 31 for Fast-Cache, and lookup overhead, C_{lookup}, of 0 for trap-driven, 21 for procedure call, 10 for buffered, and 4 for Fast-Cache.

The results in Figure 10 confirm our expectations. Trace-driven simulation has very little dependence on target miss ratio since it incurs very little overhead for target cache misses. Conversely, trap-driven simulation has a very strong dependence on target miss ratio, performing well for very low miss ratios, but degrading quickly as miss processing overhead dominates simulation time. Fast-Cache has less dependence on target miss ratio because its miss processing overhead is much lower. Nonetheless, since Fast-Cache's miss processing overhead is much larger than its lookup overhead, its slowdown is dependent on the target miss ratio.

It is important to note that Fast-Cache outperforms the other simulation techniques over much of the relevant design space even for these very simple simulations. The model indicates that Fast-Cache performs better than trap-driven simulation for miss ratios greater than 2.5% and better than buffered trace-driven simulation for miss ratios less than 20% given the costs above. This model suggests that Fast-Cache is superior to trace-driven simulation for most practical simulations, since most caches do not require action for more than 20% of the references.

Although buffering references will outperform Fast-Cache for programs with large miss ratios, it is not as general purpose as either Fast-Cache or the procedure call simulator. The model assumes a very simple simulator that counts misses in a direct-mapped cache. This represents the best-case for the buffered simulator, since it requires only the effective address of the memory reference and since the simulator performs only a single compare to determine if a reference is a hit or miss. Many simulations require more information than just the effective address of a reference. For example, simulating modified bits requires the type of the memory reference (e.g., `load` vs. `store`), and cache profiling [13, 15] requires the program counter of the memory reference. Buffering this additional information will inevitably slow down the simulation.



The simulator parameters are miss processing overhead, C_{miss}, of 250 for trap-driven, 3 for procedure call, 2 for buffered, and 31 for Fast-Cache, and lookup overhead, C_{lookup}, of 0 for trap-driven, 21 for procedure call, 10 for buffered, 4 for Fast-Cache.

Figure 10: Qualitative Simulator Performance

For these more complex simulations, each reference will incur additional overhead to store this information in the buffer and to extract the information in the simulator. Assuming the additional store/load pair adds two cycles to the no-action case, simulator overhead increases by 20%. In contrast, Fast-Cache and the procedure call simulator incur no additional overhead, since they can use static analysis and directly invoke specific simulator functions for each reference type and pass the program counter as an argument along with the memory address. Simulating set-associative caches or multiple cache configurations also increases the lookup overhead for trace-driven simulators since they may have to perform multiple compares to determine that no-action is required. Finally, and perhaps most importantly, execution driven simulators [5,23] can not be implemented with the buffered simulator since the data may not be valid at the time of the reference. Therefore, we do not discuss the buffered simulator after this section.

Trap-driven simulation will be more efficient than Fast-Cache for some studies, such as large, second-level caches or TLBs. However, Fast-Cache will be better for complete memory hierarchy simulations, since first-level caches are unlikely to be much larger than 64 kilobytes [9]. Furthermore, if the hardware is available, the active memory abstraction can use the trap-driven technique as well. Thus the active memory abstraction gives the best performance over most of the design space

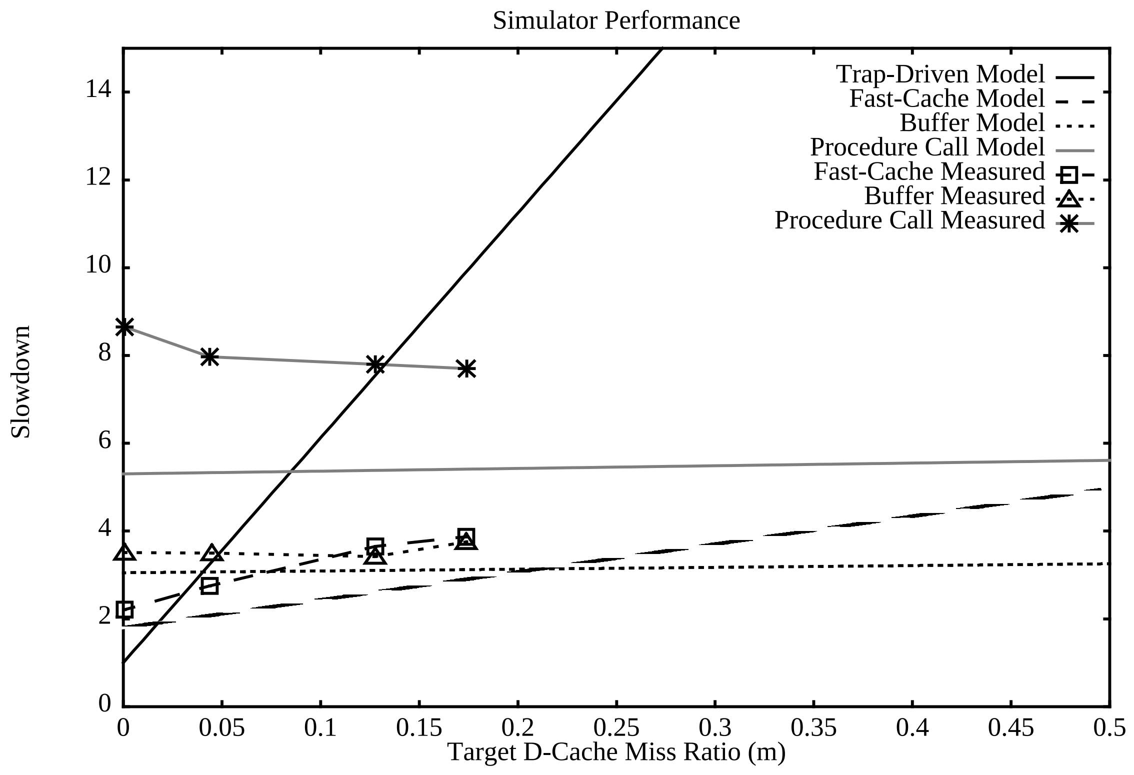


Figure 11: Measured Simulator Performance (Compress)

To verify the simple model, we use 4 programs from the SPEC92 benchmark suite [28]: `compress`, `fpppp`, `tomcatv`, and `xlisp`. All programs operate on the SPEC input files, and are compiled with `gcc` version 2.6.0 or `f77` version 1.4 at optimization level -O4.

We measured the slowdowns of Fast-Cache and the two trace-driven simulators. To obtain a range of target miss ratios we varied the target cache size from 16 kilobytes to 1 megabyte, all direct-mapped with 32-byte blocks. We also simulated a 4-kilobyte cache for `fpppp` and `xlisp`, because of their low miss ratio on the other caches. We measure execution time by taking the minimum of three runs on an otherwise idle machine, as measured with the UNIX `time` command. System time is included because the additional memory used by Fast-Cache may affect the virtual memory system.

The results, shown in Figure 11 and Figure 12, indicate that over the range of target caches we simulated (4KB–1MB), Fast-Cache is 0 to 1.5 times faster than the buffered simulator and 2 to 4 times faster than the procedure call simulator. More importantly, these measured slowdowns corroborate the general trends predicted by the model. The trace-driven simulators have very little dependence on the target miss ratio, and the higher lookup overhead of the procedure call implementation results in significantly larger slowdowns. The measured performance of Fast-Cache also exhibits the expected behavior; slowdowns increase as the target miss ratio increases. However, the model clearly omits some important factors (e.g., memory system performance): the procedure call simulator is at least 100% of the original program's execution time slower than predicted, and Fast-Cache is up-to 100% of the original program's execution time slower than predicted.

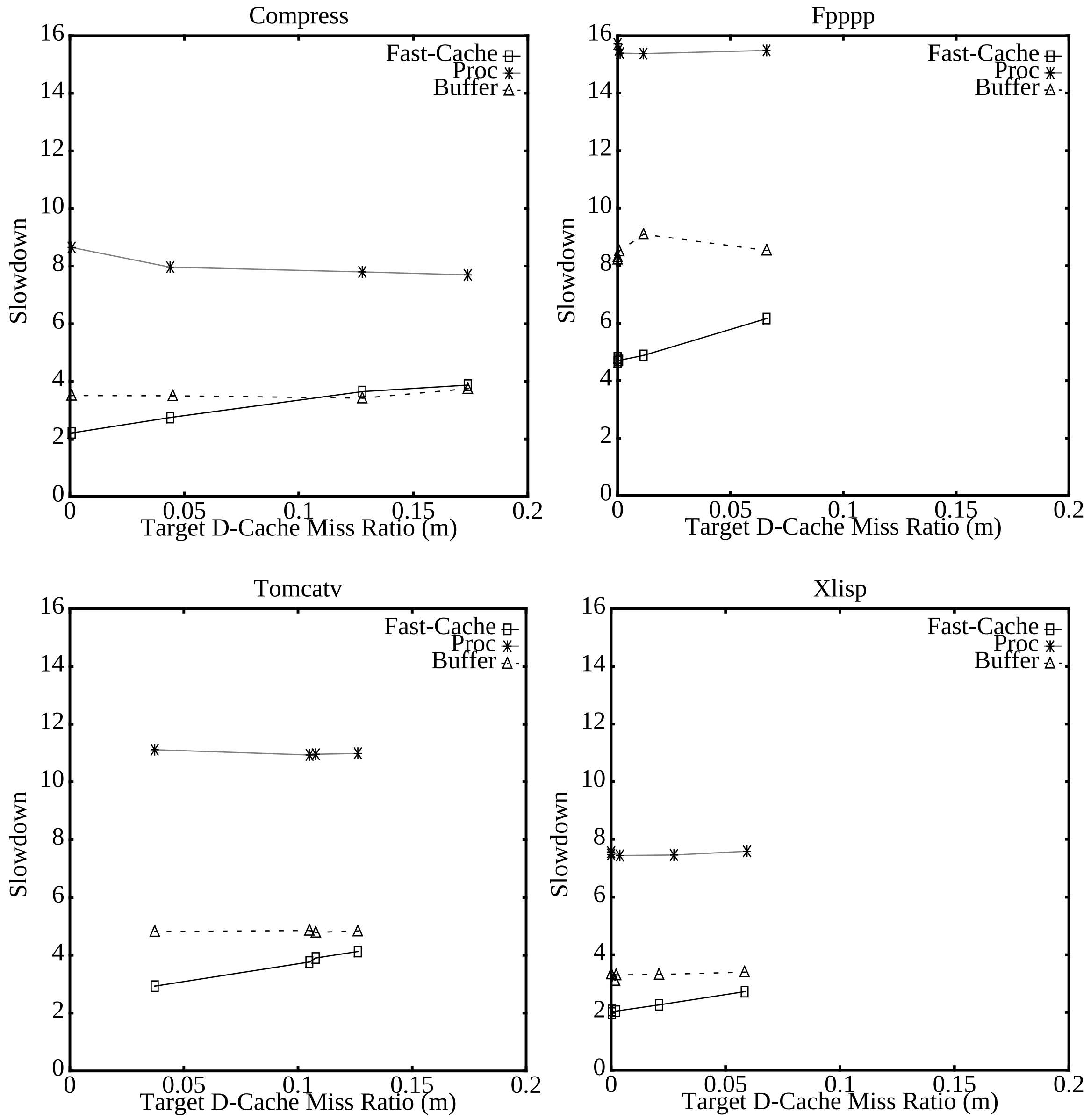


Figure 12: Measured Simulator Performance

6 Detailed Analysis

The model derived in Section 5 is useful for making qualitative comparisons between simulation techniques. However, actual simulator performance depends on details of the particular implementation and the specific host machine. In this section, we extend Equation 2 to incorporate the details of a Fast-Cache implementation executing on a SPARCstation 10/51.

First, we refine lookup overhead, which depends on whether or not Fast-Cache can use the SPARC condition codes. The lookup requires C_{cc} = 3 cycles when the condition codes can be used and C_{nocc} = 7 cycles when they cannot. If f_{cc} is the fraction of memory references where the lookup can use the condition codes, then the number of lookup cycles is: $C_{\text{lookup}} = f_{\text{cc}} \cdot C_{cc} + (1 - f_{\text{cc}}) \cdot C_{nocc}$. Substituting into Equation 2 yields a more accurate slowdown model:

$$\text{Slowdown}_{\text{Inst}} = 1 + \frac{r}{\text{CPI}_{\text{orig}}}(f_{\text{cc}} \cdot C_{cc} + (1 - f_{\text{cc}}) \cdot C_{nocc} + m \cdot C_{\text{miss}}) \tag{3}$$

$\text{Slowdown}_{\text{Inst}}$ is still an optimistic estimate because it assumes no adverse effects on the host cache. Including terms for the additional host instruction and data cache misses caused by Fast-Cache provides a more accurate model:

$$\text{Slowdown} = \text{Slowdown}_{\text{Inst}} + \text{Slowdown}_{\text{D-Cache}} + \text{Slowdown}_{\text{I-Cache}} \tag{4}$$

Section 6.1 investigates Fast-Cache's impact on the host data cache, and computes an estimate for its affect, $\text{Slowdown}_{\text{D-Cache}}$. Section 6.2 develops a model for Fast-Cache's instruction cache behavior, and an estimate for $\text{Slowdown}_{\text{I-Cache}}$. It also presents an alternative implementation, called Fast-Cache-Indirect, that trades off more instructions in the common case for better instruction cache performance. Section 6.3 discusses the overall performance of Fast-Cache and Fast-Cache-Indirect.

6.1 Data Cache Effects

The slowdown due to data cache interference, $\text{Slowdown}_{\text{D-Cache}}$, is simply the number of additional host data cache misses multiplied by the host data cache miss penalty C_{hostmiss}. We use asymptotic analysis to bound the number of misses, since modeling the interference exactly is difficult.

The lower bound, $\text{Slowdown}_{\text{D-Cache}}^{\text{lower}}$, is simply 0, obtained by assuming there are no additional misses. The upper bound is determined by assuming that each data cache block Fast-Cache touches results in a miss. Furthermore, each of these blocks displaces a "live" block, causing an additional miss later for the application.

Fast-Cache introduces data references in two places: action lookup and target miss processing. Recall that action lookup, performed for each memory reference in the application, loads a single byte from the state table. Thus in the worst case, Fast-Cache causes two additional misses for each memory reference in the application. This results in an additional $2 \cdot r \cdot I_{\text{orig}} \cdot C_{\text{hostmiss}}$ cycles for simulation.

Processing a target cache miss requires that the simulator touch B_{h} unique blocks. These blocks include target cache tag storage, the state of the replaced block, and storage for metrics. For the direct-mapped simulator used in these experiments, $B_{\text{h}} = 5$. In the worst case, each target miss causes the simulator to incur B_{h} host cache misses and displace B_{h} live blocks. If each displaced block results in a later application miss, then $2 \cdot r \cdot I_{\text{orig}} \cdot m \cdot B_{\text{h}} \cdot C_{\text{hostmiss}}$ cycles are added to the simulation time. Equation 5 shows the upper bound on the slowdown resulting only from data cache effects.

$$\text{Slowdown}_{\text{D-Cache}}^{\text{upper}} = \frac{2 \cdot r \cdot C_{\text{hostmiss}}}{\text{CPI}_{\text{orig}}}(1 + m \cdot B_{\text{h}}) \tag{5}$$

To be a true asymptotic bound, we must assume that the additional misses miss in *all* levels of the host cache hierarchy. This seems excessively pessimistic given that the host machine—a SPARCstation 10/51—has a unified 1-megabyte direct-mapped second-level cache backing up the 16-kilobyte 4-way-associative first-level data cache. Instead, we assume C_{hostmiss} is the first-level cache miss penalty, or 5 cycles [32].

The characteristics of the programs used to validate this model are shown in Table 3. Figure 13 plots the measured and modeled slowdowns as a function of target miss ratio. The lowest line is $\text{Slowdown}_{\text{Inst}}$, the asymptotic lower bound. The upper

Program	Instructions (billions)	References (billions)	r	f_{cc}	CPI
Compress	0.08	0.02	0.25	0.95	1.22
Fpppp	5.41	2.58	0.48	0.83	1.22
Tomcatv	1.65	0.67	0.41	0.52	1.61
Xlisp	5.85	1.53	0.26	0.98	1.38

TABLE 3. Benchmark characteristics

line is the approximate upper bound, assuming a perfect instruction cache and second-level data cache. The measured slowdowns are plotted as individual data points. The results show two things. First, the upper bound approximations are acceptable because all measured slowdowns are well within the bounds. Second, the upper bound is conservative, significantly overestimating the slowdown due to data cache pollution.

The upper bound is overly pessimistic because (i) not all Fast-Cache data references will actually miss, and (ii) when they do miss, the probability of replacing a live block is approximately one-third, not one [37]. To compute a single estimator of data cache performance, we calculate the mean of the upper and lower bounds:

$$\text{Slowdown}_{\text{split}} = \text{Slowdown}_{\text{Inst}} + \frac{\text{Slowdown}_{D-Cache}^{upper}}{2} \tag{6}$$

As Figure 13 shows, this estimator—although simplistic—is quite accurate, predicting slowdowns within 30% of the measured values.

6.2 Instruction Cache Effects

The $\text{Slowdown}_{\text{split}}$ estimator is accurate despite ignoring instruction cache pollution. This is because most of the SPEC benchmarks have extremely low instruction cache miss ratios on the SPARCstation 10/51 [6]. Thus, Fast-Cache's code expansion has very little effect on their performance. In contrast, for codes with more significant instruction cache miss ratios, such as `fpppp`, instruction cache behavior has a noticeable impact.

To understand the effect of code dilation on instruction cache pollution, consider a 16-kilobyte instruction cache with 32-byte blocks. Assume that the Fast-Cache instrumentation expands the application's dynamic code size by a factor of 4. Normally, this cache would hold 4096 of the application's instructions; but with code dilation, the cache will contain, on average only 1024 of the original instructions. Similarly, each cache block originally held 8 instructions; after instrumentation each holds an average of 2 original instructions. Intuitively, we should be able to estimate the cache performance of the instrumented code by simulating a cache one-fourth as large, with cache blocks one-fourth as big.

This observation suggests that we can approximate instruction cache performance by assuming that each instruction in the original program is E times bigger, where E is the average dynamic code dilation. In other words, the cache performance of the *instrumented* application on the *original* instruction cache should be roughly the same as the performance of the *un-instrumented* application on a cache that has $1/E$ times the capacity and $1/E$ times the cache block size as the original instruction cache. We call this the *scaled cache* model, and compute cache miss ratios by simulating the appropriately scaled cache con-

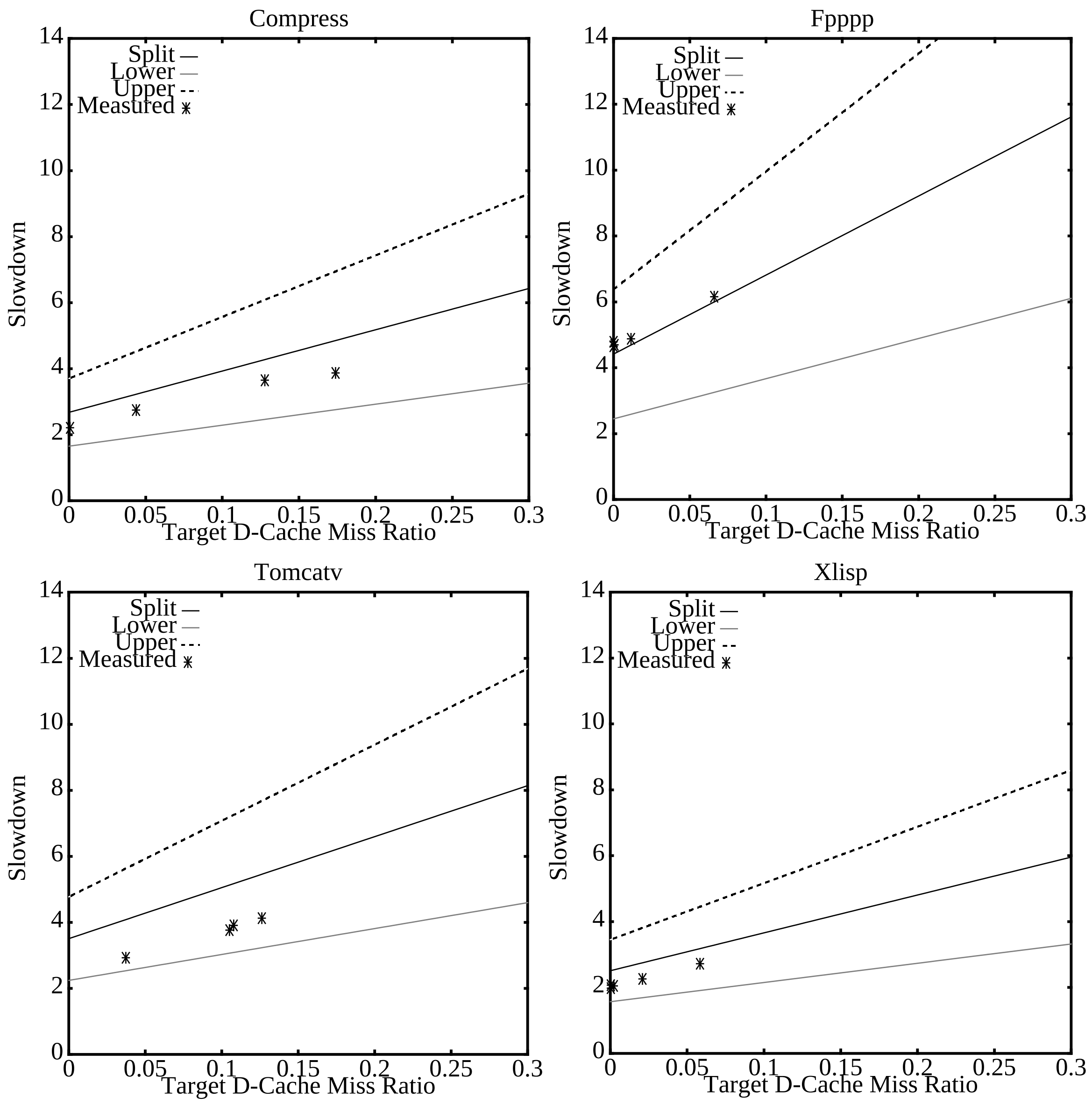


Figure 13: Data Cache Model

We can avoid simulating the scaled caches, and estimate their effect using design target miss ratios [27] and other available data [6]. Design target miss ratios predict that decreasing the cache size by a factor of E increases the number of misses by $\sqrt{E}$. Data gathered by Gee, et al. [6] indicates that decreasing the instruction cache block size by E increases the number of instruction cache misses by E. Thus we expect that the number of instruction cache misses will be approximately $E\sqrt{E}$ times the original number of instruction cache misses. Since the original program incurs $I_{\text{orig}} \cdot m_i$ misses, Fast-Cache incurs an additional slowdown of:

$$\text{Slowdown}_{\text{I-Cache}} = \frac{(E\sqrt{E} - 1) \cdot \text{m}_i \cdot C_{hostmiss}}{\text{CPI}_{\text{orig}}} \qquad (7)$$

We compute Fast-Cache's code expansion by multiplying the number of instructions inserted for the table lookup by the number of times the lookup is executed. If Fast-Cache inserts I_{cc} = 9 instructions when it can use the condition codes and I_{cc} = 7 instructions when it cannot, then the total code expansion is simply:

$$E = 1 + r \cdot (f_{cc} \cdot I_{cc} + (1 - f_{cc}) \cdot I_{nocc}) \qquad (8)$$

Since the total code expansion (see Table 5) is roughly a factor of 4, we expect the instrumented code to incur roughly 8 times as many instruction cache misses. Of course, these are general trends, and any given increment in code size can make the difference between the code fitting in the cache or not fitting. In this case, the miss ratio can increase by a much larger amount.

This analysis indicates that Fast-Cache is likely to perform poorly for applications with high instruction cache miss ratios, such as the operating system or large commercial codes [18]. To reduce instruction cache pollution, we present an alternative implementation, *Fast-Cache-Indirect*, which inserts only two instructions—a jump-and-link plus effective address calculation—per memory reference. This reduces the code expansion from a factor of 4 to 1.6, for typical codes. Consequently, the model predicts that the instrumented code will have only $1.6\sqrt{1.6} \approx 2$ times as many instruction cache misses. The drawback of this approach is an additional 3 instructions on the critical no-action lookup path, however it will be faster for some ill-behaved codes. For the benchmarks we studied, Fast-Cache-Indirect executes 3.4 to 7 times slower than the original program. This is 1.2 to 1.8 times slower than Fast-Cache (Figure 15.)

To validate the instruction cache models, we use *Shade* [4] to measure the instruction cache performance of the instrumented programs.[1] Because the code expansion is not exactly a power of two, we validate the scaled model by simulating caches of the next larger and smaller powers of two and interpolate. Table 4 and Table 5 show how well the two models match the measured values. For `fpppp`, `tomcatv` and `xlisp`, the scaled model is within 32% of the measured instruction cache performance. The relative difference is larger for `compress`, but it has so few misses that a relative difference is meaningless.

Bench-mark	**Original Misses (m_i)**	**Shade**	**Scaled Model (% error)**	**$E\sqrt{E}$ Model (% error)**	**Code Exp**
Compress	329 (0.0%)	1,843	984 (46%)	1,848 (0%)	3.16
Fpppp	336,224 (3.7%)	4,629,793	4,361,246 (6%)	3,929,520 (15%)	5.15
Tomcatv	1,402 (0.0%)	27,143	33,680 (24%)	12,414 (54%)	4.28
Xlisp	1,538 (0.0%)	578,773	442,077 (24%)	9,984 (98%)	3.48

TABLE 4. Fast-Cache Instruction Cache Performance

1. Due to Shade's large slowdowns, we used smaller input data sets for fpppp, tomcatv, and xlisp. This should have little impact on the instruction cache performance.

Bench-mark	Original Misses (m_i)	Shade	Scaled Model (% error)	$E\sqrt{E}$ Model (% error)	Code Exp
Compress	329 (0.0%)	1,221	458 (62%)	592 (50%)	1.48
Fpppp	336,224 (3.7%)	1,033,342	954,609 (8%)	922,598 (10%)	1.96
Tomcatv	1,402 (0.0%)	5,935	7,847 (32%)	3,385 (42%)	1.80
Xlisp	1,538 (0.0%)	13,670	14,890 (9%)	2,882 (78%)	1.52

TABLE 5. Fast-Cache-Indirect Instruction Cache Performance

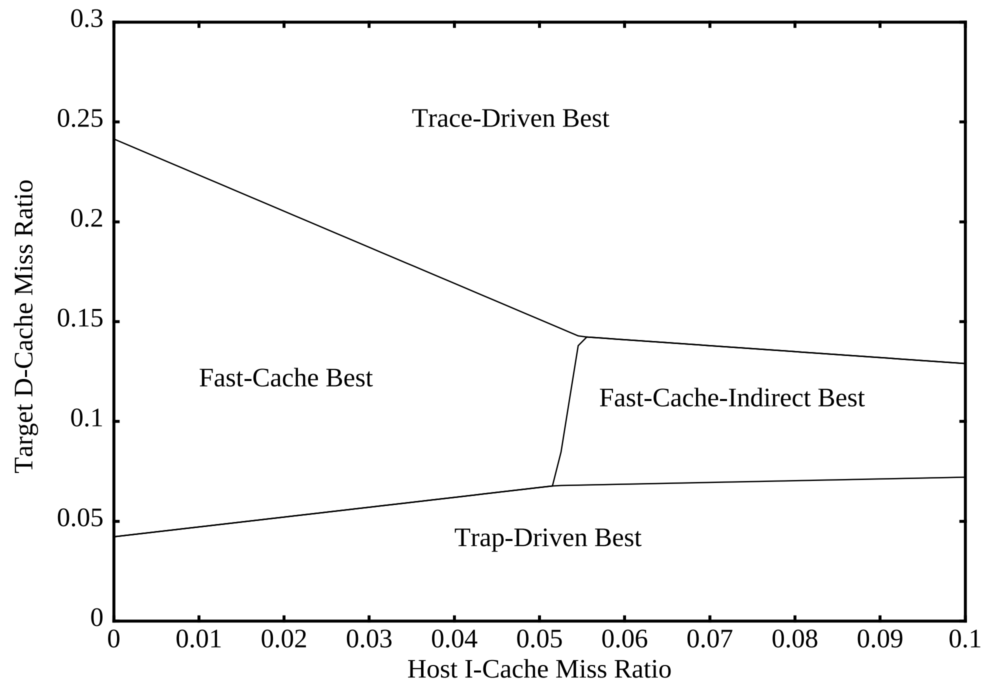
The scaled model captures the general trend in instruction cache misses caused by code dilation. However, it assumes the dilation is uniform, hence it is not a precise predictor. Similarly, $E\sqrt{E}$ captures general trends, but is not a precise predictor. For example, the measured instruction cache miss ratio for `tomcatv` increases by a factor of 20 rather than the predicted factor of 9. This occurs because the instrumentation enlarges the instruction working set beyond the SuperSPARC cache size. However, for three of the benchmarks the impact on performance is negligible because the applications have such low miss ratios (i.e., less than 0.007%).

`Fpppp` is the only benchmark with a non-negligible instruction cache miss ratio (3.7%) and $E\sqrt{E}$ predicts the number of instruction cache misses within 15% for Fast-Cache and 10% for Fast-Cache-Indirect. To further evaluate this model we use the reference counter of the SuperSPARC second-level cache controller [32] to measure the number of level-one misses for the original data set. The count includes both data cache read misses and instruction cache misses, but `fpppp` is dominated by instruction cache misses. $E\sqrt{E}$ predicts the number of misses within 36% for Fast-Cache and 4% for Fast-Cache-Indirect.

6.3 Overall Performance

We now use the detailed model to revisit the comparison between Fast-Cache, trap-driven and trace-driven simulation. Figure 14 compares the detailed performance model for Fast-Cache and Fast-Cache-Indirect against the qualitative model (Equation 2) for both trap-driven and trace-driven simulation; the graph plots the regions of best performance as a function of the original program's host instruction cache miss ratio and the target data cache miss ratio. Note that this comparison is biased against Fast-Cache, since we assume that neither trap-driven nor trace-driven simulation incur any cache pollution. The comparison shows that either Fast-Cache or Fast-Cache-Indirect performs best over an important region of the design space. Although trap-driven simulation performs best for low data cache miss ratios, recall that it is not always an option. Therefore, with respect to trace-driven simulation, Fast-Cache covers an even larger area of the design space.

Incorporating the cache pollution caused by Fast-Cache's additional instructions and data references shows that Fast-Cache's performance can degrade for programs with large instruction cache miss ratios. Nonetheless, even for simple data cache simulations, the model indicates that Fast-Cache covers most of the relevant design space. The model predicts Fast-Cache's instruction cache performance on a SPARCstation 10/51 to within 32% of measured values, using the scaled cache



r = 0.25, f_{cc} = 0.95, CPI = 1.22, B_h = 5
Fast-Cache: C_{cc} = 3, C_{nocc} = 7, C_{miss} = 31, I_{cc} = 9, I_{nocc} = 7
Fast-Cache-Indirect: C_{cc} = 7, C_{nocc} = 9, C_{miss} = 34, I_{cc} = I_{nocc} = 2
Procedure Call Trace-Driven: C_{lookup} = 21, C_{miss} = 3
Trap-Driven: C_{miss} = 250

Figure 14: Overall Simulator Performance

model, and 36% using $E\sqrt{E}$. For the programs we ran, instruction cache pollution has little effect on Fast-Cache simulation time (see Figure 15.) However, when simulating programs with larger instruction cache miss ratios, Fast-Cache-Indirect should be a better implementation.

7 Active Memory Applications and Extensions

7.1 Applications

The active memory abstraction enables efficient simulation of a broad range of memory systems. Complex simulations can benefit from both the NULL handler and direct invocation of simulator functions. For example, active memory can be used to simulate set-associative caches as well. A particular simulator depends on the policy for replacing a block within a set. Random replacement can use an implementation similar to the direct-mapped cache, calling a handler only when a block is not resident in the cache. An active memory implementation of least recently used (LRU) replacement can optimize references to the most recently used (MRU) block since the LRU state does not change. References to MRU blocks would invoke the NULL handler, while all other references invoke the simulator. This is similar to Puzak's trace filtering for set-associative caches [21]; the property of inclusion [17] indicates the number of references optimized is equal to the number of cache hits in a

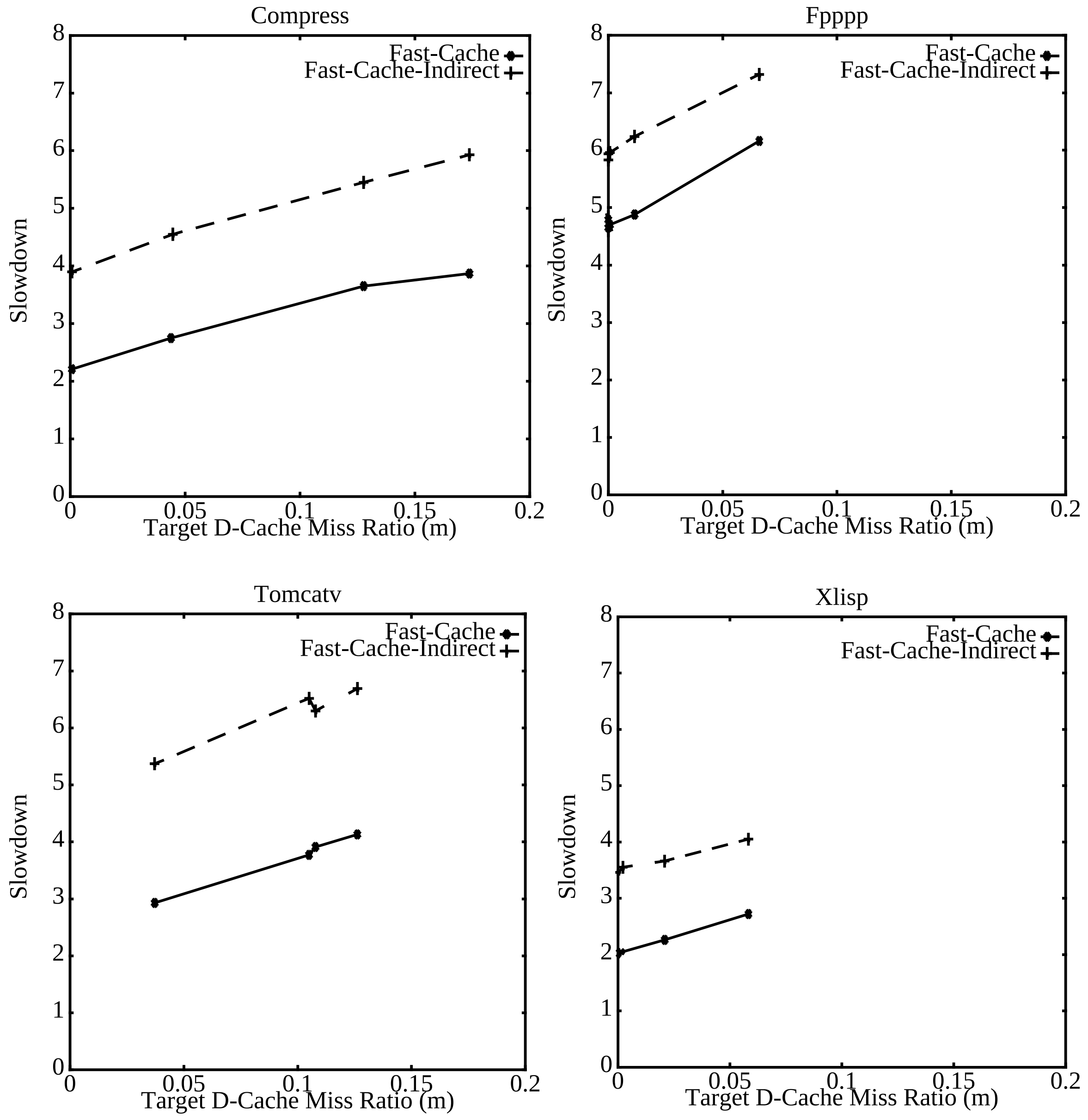


Figure 15: Fast-Cache-Indirect Performance

direct-mapped cache, with the same number of sets as the set-associative cache. A further optimization distinguishes misses from hits to non-MRU blocks by using more than two states per cache block. An example configuration is shown in Figure 16.

Many simulators that evaluate multiple cache configurations [7,31,17] use the property of inclusion [17] to limit the search for caches that contain a given block. No action is required for blocks that are contained in *all* simulated caches. An active memory implementation can optimize these references with the NULL handler.

This same technique can be used to efficiently simulate multiple cache configurations that *do not* maintain inclusion. The NULL handler is invoked only if no action is required for any of the alternative caches (e.g, MRU block in all caches). When action is required, the simulator can use the state to encode which caches contain a particular block and directly invoke a func-

STATE	Handler	Comment
0	miss_handler	/* called for blocks not in the cache */
1	non_mru_hit	/* cache hits to non-mru blocks */
2	noaction	/* cache hits to mru blocks */
3	noaction	/* unused */

Figure 16: Set-Associative Cache with LRU Replacement

tion specialized to update the appropriate caches. Simple simulations of a single cache benefit primarily from the efficiency of the predefined NULL handler.

Finally, the active memory abstraction has been used to simulate the Typhoon multiprocessor [24] and to provide low-cost portable fine-grain access control [25].

7.2 Extensions

A combination of table lookup and static analysis [36] can be used to efficiently simulate instruction fetches. The program counter for each instruction is easily obtained when adding instrumentation. For split instruction and data caches, at the beginning of each basic block a table lookup is performed for only the instructions that occupy unique cache blocks. For unified caches, exact simulation requires checking at a finer grain, however the added accuracy is probably not worth the extra overhead.

Timing dependent simulations, such as prefetching, write buffers [9] or lockup-free caches [10], require accurate instruction cycle counts. Fast-Cache can easily add instruction cycle counts using techniques similar to QPT [11] or the Wisconsin Wind Tunnel [23]. Although simulator overhead will increase to update the cycle count, the active memory abstraction still permits efficient simulation of these complex memory systems. For example, to simulate hardware initiated prefetches, a simulator similar to the one shown in Figure 5 of Section 3, can be used. The miss handler would initiate the prefetch according to some policy (e.g., next block), and mark the state of the prefetched block `prefetch`. If the application references a block in the state `prefetch` a separate prefetch handler is invoked to increment time by the amount required for the prefetch to complete and to mark the state of the block `valid`. This eliminates the need to check the prefetch buffer on every miss. If the prefetch should have completed before the application references the block, then the prefetch handler can simply mark the state of the block `valid`.

A similar approach can be used to simulate write buffers. However, action is required for each `store` instruction to update the write buffer. If writes can be merged, the state of the block can be used to indicate that a merge may be required. This eliminates the need to examine the write buffer on each `store`. Similarly, the state of a block can indicate what action is necessary for a `load` instruction. For example, the `load` may be required to stall until the buffer drains.

Accurate cycle counts also permits the active-memory abstraction to support efficient simulation of lockup-free caches [10]. For static pipelines, the abstraction is extended to support a limited form of “busy bits”—a bit associated with each register indicating its contents are not available as an operand. The user controls the value of each register’s busy bit, marking a register busy when it is the destination of an outstanding `load`. The bit is checked only at the first use of the register after the corresponding load; if it is busy a simulator function is invoked to process outstanding requests until the register is no longer busy. Pipelines that can issue instructions out of order present a more challenging problem to any memory system simulator, since it is difficult to determine which instructions can be issued. One possible solution is to use static analysis and executable editing [12] to determine and create groups of instructions—called *tasks*—that can be issued independently. If a task experiences a cache miss, it is suspended until the load completes and another task is selected to execute.

Currently, the active memory abstraction provides a single predefined function—the NULL handler. The abstraction can be extended to support other predefined functions. For example, it could provide a set of counters and predefined functions for incrementing particular counters.

Finally, to support simulation of unaligned memory accesses, implementations of the abstraction may have to dynamically detect when a cache block boundary is crossed and invoke the appropriate handlers. This may increase the lookup overhead for active memory, but a trace-driven simulator would also incur this additional overhead. For some architectures, it may be possible to statically determine that some memory instructions are aligned and eliminate the need for an alignment check.

8 Conclusion

The performance of conventional simulation systems is limited by the simple interface—the reference trace abstraction—between the reference generator and the simulator. This paper examines a new interface for memory system simulators—the *active memory* abstraction—designed specifically for on-the-fly simulation. Active memory associates a state with each memory block, and simulators specify a function to be invoked when the block is referenced. A simulator using this abstraction manipulates memory block states to control which references it processes. A predefined NULL function can be optimized in active memory implementations, allowing expedient processing of references that do not require simulator action. Active memory isolates simulator writers from the details of reference generation—providing simulator portability—yet permits efficient implementation on stock hardware.

Fast-Cache implements the abstraction by inserting 9 instructions before each memory reference, to quickly determine whether a simulator action is required. We both measured and modeled the performance of Fast-Cache. Measured Fast-Cache simulation times are 2 to 6 times slower than the original, un-instrumented program on a SPARCstation 10; a procedure call based trace-driven simulator is 7 to 16 times slower than the original program, and a buffered trace-driven simulator is 3 to 8 times slower. The models show that Fast-Cache will perform better than trap-driven or trace-driven simulation for target miss ratios between 5% and 20%, *even* when we account for cache interference for Fast-Cache but not for the other simulators. Furthermore, the system features required for trap-driven simulation are not always available, increasing the range of miss ratios where Fast-Cache is superior.

The detailed model captures the general trend in cache interference caused by Fast-Cache's instrumentation code. The model indicates that code dilation may cause eight times as many instruction cache misses as the original program. Although the instruction cache miss ratios for the applications we studied were so low that this increase was insignificant, larger codes may incur significant slowdowns. Fast-Cache-Indirect significantly reduces code dilation at the expense of 3 extra cycles for the table lookup.

As the impact of memory hierarchy performance on total system performance increases, hardware and software developers will increasingly rely on simulation to evaluate new ideas. Fast-Cache provides the mechanisms necessary for efficient memory system simulation by using the active memory abstraction to optimize for the common case. In the future, as the ability of processors to issue multiple instructions in a single cycle increases, the impact of executing the instrumentation that implements the active memory abstraction will decrease, resulting in even better simulator performance.

9 Acknowledgments

We would like to thank James Larus for providing EEL. Mark Hill, James Larus, Babak Falsafi, Steve Reinhardt, and Thea Sklenar provided comments on early drafts of this paper. We also appreciate the support provided by the members of the Wisconsin Wind Tunnel project. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government. This work is supported in part by NSF PYI Award CCR-9157366, NSF Grants CDA-9024618 and MIP-9225097, donations from Thinking Machines Corp., Digital Equipment Corp., SUN Microsystems, Xerox Corp., and by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550

10 References

[1] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 119–127, June 1986.

[2] Robert C. Bedichek. Talisman: Fast and Accurate Multicomputer Simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 14–24, May 1995.

[3] Anita Borg, R. E. Kessler, and David W. Wall. Generation and Analysis of Very Long Address Traces. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 270–281, May 1990.

[4] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[5] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software)*, pages II99–II107, August 1991.

[6] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan J. Smith. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro*, 13(4):17–27, August 1993.

[7] Mark D. Hill and Alan J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[8] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[9] Norman P. Jouppi and Steven J. E. Wilton. Tradeoffs in Two-Level On-Chip Caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, April 1994.

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

[11] James R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[12] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[13] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE COMPUTER*, 27(10):15–26, October 1994.

[14] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory System Simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 220–230, May 1995.

[15] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[16] M. Martonosi, A. Gupta, and T. Anderson. Effectiveness of Trace Sampling for Performance Debugging Tools. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 248–259, May 1993.

[17] R. L. Mattson, J. Gecsei, D. R. Schultz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[18] Ann Marie Grizzaffi Maynard, Colette M. Donnely, and Bret R. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 145–156, October 1994.

[19] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 27–38, May 1993.

[20] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989. Also available as Rice COMP TR 89-93.

[21] T. R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachusetts, February 1985. Dept. of Electrical and Computer Engineering.

[22] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[23] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[24] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–336, April 1994.

[25] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–306, 1994.

[26] Alan. J. Smith. Two Methods for Efficient Analysis of Memory Address Trace Data. *IEEE Transactions on Software Engineering*, 3(12), January 1977.

[27] Alan J. Smith. Line (block) size choice for CPU cache memories. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[28] SPEC. SPEC Newsletter, December 1991.

[29] Amitabh Srivastava and Alan Eustace. ATOM A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[30] Craig B. Stunkel and W. Kent Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.

[31] R. A. Sugumar and S. G. Abraham. Efficient Simulation of Multiple Cache Configurations using Binomial Trees. *Technical Report CSE-TR-111-91*, 1991.

[32] Texas Instruments. *SuperSPARC User's Guide*, 1992. Alpha Edition.

[33] Chandramohan A. Thekkath and Henry M. Levy. Hardware and Software Support for Efficient Exception Handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 110–119, October 1994.

[34] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Trap-Driven Simulation with TapewormII. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 132–144, October 1994.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP)*, pages 203–216, December 1993.

[36] David B. Whalley. Fast Instruction Cache Performance Evaluation Using Compiler-Time Analysis. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–22, May 1992.

[37] David A. Wood, Mark D. Hill, and R. E. Kessler. A Model for Estimating Trace-Sample Miss Ratios. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 79–89, May 1991.

Appendix A

This appendix describes the code sequences—*called snippets*—that are used by Fast-Cache and the buffered trace-driven simulator. Each simulator has two different snippets, depending on whether condition codes are live or not. The instructions on the critical path are in bold face, and the number before the instruction indicates the cycle the instruction is issued. The comments between instructions explain why two instructions are not co-issued on the SuperSPARC processor. Different schedules are possible on other processors.

In each snippet, the register `%g0` is a place holder. When Fast-Cache inserts the snippets, this register specifier is set to the appropriate value according to the instrumented memory reference. Immediate fields are also set when the snippet is inserted, we use 0x0 as a place holder for immediates. Instructions for computing the effective address are shown using register + register addressing, they may change to register + immediate addressing when the snippet is inserted.

We have also included the code sequences—*called handler stubs*—used when invoking the simulator (e.g., action cases in Fast-Cache or buffer full for buffered trace-driven simulation). Again, different stubs are used when condition codes are live, since they must be saved, versus when they are dead.

Fast-Cache: Dead Condition Codes

The Fast-Cache lookup snippet requires 3 cycles when condition codes are not live, assuming the first instruction can be issued with previous instruction of application. If the first instruction can not be issued with the previous application instruction, then 4 cycles are required.

```
0       add %g0, %g0, %g5              ! get the effective address
           ! split cascade into shift
1       sra %g5, 0x0, %g6              ! calculate block byte-index, 0x0 is set to
                                          block size
           ! split ALUOP into LD
2       ldub [%g7 + %g6], %g6          ! load block state byte
           ! split load data use
3       andcc %g6, 0x0, %g0            ! check the right bit, 0x0 set to correct
                                          mask
        bne 1f
        LD or ST                       ! the memory ref goes here
        sll %g6, 0x0, %g6              ! shift by stub size
        sethi 0x0, %g7                 ! set the stub base pointer
        jmpl %g7 + %g6, %g6            ! jump to handler stub
        sethi %hi(TBL_BASE), %g7       ! restore the state table pointer
     1:                                ! next application instruction
```

Fast-Cache: Live Condition Codes

The Fast-Cache lookup snippet requires 7 cycles when condition codes are live, assuming the first instruction can be issued with the previous instruction of application. The in-line sequence (shown below) takes 5 cycles, and an additional 2 cycles are required for the NULL handler (`ret` & `nop` instructions).

```
0       add %g0, %g0, %g5            ! get the effective address
            ! split cascade into shift
1       sra %g5, 0x0, %g6            ! calculate block byte-index
            ! split ALUOP into LD
2       ldub [%g7 + %g6], %g6        ! load block state byte
            ! split load data use
3       sll %g6, 0x0, %g6            ! shift by stub size
        sethi 0x0, %g7               ! set the tbl ptr
            ! split before cascade into jmpl
4       jmpl %g7 + %g6, %g6          ! jump to handler jump table
            ! split after control transfer
5       sethi %hi(TBL_BASE), %g7     ! restore the bit tbl ptr
```

Fast-Cache-Indirect: In-line Snippet

Fast-Cache-Indirect inserts only two instructions before each memory reference. These two instructions require 1 cycle to execute.

```
0    jmpl %g7 + 0x0, %g6          ! jump to handler jump table
            ! split after control transfer
1    add %g0, %g0, %g5            ! get the effective address
```

Fast-Cache-Indirect: Dead Condition Codes

The out-of-line snippet for Fast-Cache-Indirect is nearly identical to the in-line snippet used by Fast-Cache. However, the effective address is already computed by the time control reaches this snippet so the first instruction is the shift to calculate the byte index. If no action is required, this snippet executes in 4 cycles, completing the no action case in a total of 6 cycles. Note that we have started time at cycle 2 for this snippet because of the one cycle required to transfer control.

```
2       sra %g5, 0x0, %g5            ! calculate block byte-index
            ! split ALUOP into LD
3       ldub [%g7 + %g5], %g7        ! load block state byte
            ! split load data use
4       andcc %g7, 0x0, %g0          ! check the right bit
        bne 1f
            ! split after control transfer
        sll %g5, 0x0, %g5            ! shift the effective address back
        save %sp, -96, %sp           ! get some registers
        sll %g7, 0x0, %l0            ! shift by stub size
        sethi 0x0, %l1               ! set the jmp tbl ptr
        sethi %hi(TBL_BASE), %g7     ! set bit tbl ptr
        jmpl %l0 + %l1, %g0          ! jump to handler jump table
        restore                      ! restore the regs

        !! these two instructions are never executed if action is
        !! required and the above jmpl is taken

5    1: jmpl %g6 + 8, %g0            ! return to application
6       sethi %hi(TBL_BASE), %g7     ! restore the bit tbl ptr
```

Fast-Cache-Indirect: Live Condition Codes

When condition codes are live, Fast-Cache-Indirect requires 9 cycles to complete the no action case (two additional cycles are required for the `return` and `nop`). Again, we have started time at 2 to account for the 1 cycle to transfer control to this snippet. This snippet is slightly different than the in-line snippet, since we cannot destroy the value in `%g6`, since it holds the return address.

```
2    save %sp, -96, %sp        ! get some registers
            ! split after serial instruction
3    sra %g5, 0x0, %l0         ! calculate block byte-index
            ! split ALUOP into LD
4    ldub [%g7 + %l0], %l1     ! load block state byte
            ! split load data use
5    sll %l1, 0x0, %l1         ! shift by stub size
     sethi 0x0, %l2            ! set the tbl ptr
            ! split before cascade into jmpl
6    jmpl %l2 + %l1, %g0       ! jump to handler jump table
            ! split after control transfer
7    restore
```

Buffer: Dead Condition Codes

The in-line buffer snippet writes the memory address to the buffer in 3 cycles if condition codes are not live. This is independent of whether the first instruction is issued with the previous instruction from the application.

```
1        add %g6, 0x4, %g6         ! increment buf_ptr
         add %g0, %g0, %g5         ! get the effective address
            ! split--out of register write ports
2        cmp %g6, %g7              ! check if buffer full
         ble 1f                    ! branch if not full
            ! split after control transfer
3        st %g5, [%g6]             ! store it in the buffer
            ! split after delay slot instruction
         jmpl %g7+0x8, %g6         ! jump to handler jump table
         nop                       ! in case ref is in delay slot of call
     1:                            ! the memory ref goes here
```

Buffer: Live Condition Codes

When condition codes are live, the buffered simulator requires 8 cycles to store an entry in the buffer. This is independent of whether the first instruction can be issued with the applications preceding instruction, since the first five instructions will always execute in 2 cycles.

```
0        add %g6, 0x4, %g6         ! increment buf_ptr
            ! split out of register write ports
1        add %g0, %g0, %g5         ! get the effective address
         st %g5, [%g6]             ! store the address
         save %sp, -96, %sp        ! Hide modifications of %o7
            ! split out of register write ports
                                   ! Check if buffer overflowed:
2        sub %g7, %g6, %g5         ! %g5 = buf_ptr - end_buf
            ! split cascade into shift
```

```
3       sra %g5, 31, %g5           ! %g5 = -1 if overflow, 0 otherwise
           ! split cascade into shift
4       sll %g5, 2, %g5            ! %g5 = -4 if overflow, 0 otherwise
        add %g5, 16, %g5           ! %g5 = 12 if overflow, 16 otherwise
           ! split out of register write ports
5       call L70                   ! %o7 = PC
           ! split after control transfer
6       jmpl %o7+%g5, %g0
           ! split after control transfer
     L70:
7       nop
           ! split after delay slot instruction
        jmpl %g7+0x8, %g6          ! Here if overflow, empty the buffer
           ! split after control transfer
8       restore                    ! Here if no overflow
```

Handler Stub: Dead Condition Codes

The handler stub that does not save or restore the condition codes.

```
        save %sp, -96, %sp
        mov %g1, %l1               ! save globals
        mov %g2, %l2
        mov %g3, %l3
        mov %g4, %l4
        mov %g5, %l6
        mov %g6, %o1               ! save ret_pc
        mov %g5, %o0
        sethi 0x0, %g5
        jmpl %g5 + 0x0, %o7        ! call a handler
        rd %y, %l0                 ! save Y register (in delay slot)
        mov %l1, %g1
        mov %l2, %g2
        mov %l3, %g3
        mov %l4, %g4
        mov %l6, %g5               ! restore eff addr for ifetch sim
        wr %l0, %g0, %y            ! restore Y reg
        jmpl %g6 + 0x8, %g0        ! return to code
        restore
```

Handler Stub: Live Condition Codes

The handler stub that saves and restores the condition codes. setcc is the base of a jump table for snippets that restore the condition codes.

```
        save %sp, -96, %sp
        mov %g1, %l1                  ! save globals
        mov %g2, %l2
        mov %g3, %l3
        mov %g4, %l4
        mov %g5, %l6
        sethi 0x1, %l5                ! set %l5 to %hi(setcc)
        bneg,a  1f                    ! these branches save the CCR
```

```
   or %l5, 0x80, %l5
1: be,a 2f
   or %l5, 0x40, %l5
2: bvs,a   3f
   or %l5, 0x20, %l5
3: bcs,a   4f
   or %l5, 0x10, %l5
4: mov %g6, %o1                    ! save ret_pc
   mov %g5, %o0
   sethi 0x0, %g5
   jmpl %g5 + 0x0, %o7             ! call a handler
   rd %y, %l0                      ! save Y register (in delay slot)
   mov %l1, %g1
   mov %l2, %g2
   mov %l3, %g3
   mov %l4, %g4
   mov %l6, %g5                    ! restore eff addr for ifetch sim
   wr %l0, %g0, %y                 ! restore Y reg
   jmpl %l5 + 0x0, %g0             ! invoke setcc restore
   restore
```