Appears in: "Proceedings of the 1995 ACM SIGMETRICS Conference," May 1995.
Reprinted by permission of ACM.

Active Memory: A New Abstraction for Memory-System Simulation*

Alvin R. Lebeck and David A. Wood
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
{alvy,david}@cs.wisc.edu

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 simulations two or three times faster than a highly-tuned trace-driven simulator and only 2 to 7 times slower than the original program. 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%.

*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., Xerox Corp., and by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550

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. [5], estimate that *17 months* of processing time were used to obtain miss ratios for the SPEC92 benchmarks [20].

Fortunately, much of the inefficiency can be eliminated 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 which are consumed by the simulation. This abstraction hides the details of reference generation from the simulator, but introduces significant overhead 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 describes *active memory*, a new memory system simulation abstraction designed specifically for on-the-fly simulation.[1] Active memory 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. A predefined NULL handler is optimized for the no-action case

For example, a simple simulation that counts cache misses 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 refer-

[1] "Active memory" has also been used to describe the placement of processing logic next to memory. There is no connection between these terms.

ences 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 caches can be simulated by marking blocks *valid* if they are present in all caches. Since most references are to *valid* blocks, the NULL handler allows an active memory simulator to execute much faster than one using the traditional trace abstraction.

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 7 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 [21].

As described in Section 4, Fast-Cache efficiently implements this abstraction by inserting 9 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 only 3 cycles (assuming no cache misses) on a SuperSPARC processor.

In Section 5 we analyze 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 trap-driven simulation—which uses 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 40%). These results indicate that Fast-Cache is likely to be the fastest simulation technique over much of the 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 our Fast-Cache measurements on the SPARCstation 10—data cache pollution introduces at most a factor of four slowdown (over the original program). A simple model—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 eight times as many instruction misses as the original code. For most of our 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 at the expense of 3 more instructions for the "no action" case.

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; and
4. update the metrics, if necessary.

Traditionally, the first step was considered difficult and inefficient, usually requiring either expensive hardware monitors or slow instruction-level simulators [7]. The reference trace abstraction helps amortize this overhead by cleanly separating reference generation (step 1) from simulation (steps 2–4). Reference traces can be saved and reused for multiple simulations, with the added benefit of guaranteeing reproducible results.

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 [18] 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 [15] 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 [9]. On-the-fly simulation techniques—which combine steps 1–4—have become popular because they eliminate I/O overhead, context switches, and large storage requirements [4, 14, 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. 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 [4, 14], or (ii) buffer the reference in memory, incurring buffer management overhead and memory system delays caused by cache pollution [2, 22]. Furthermore, this overhead is almost always wasted, because in most simulations the common case, e.g., a cache hit, requires no action.

Clearly, optimizing the lookup (step 2) to quickly detect these "no action" cases can significantly improve simulation performance. MemSpy [11] 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 assembler, 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 [21]. However, ATOM still incurs unnecessary procedure linkage overhead in the no-action cases.

A recent alternative technique, *trap-driven simulation* [17, 25], 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. 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 does not simulate stack references [17], while Tapeworm II does not simulate any data references [25]. Furthermore, the overhead of memory exceptions 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 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 steps 3–4. Combining effective address generation and action lookup allows the simulation system to implement the no-action cases with no unnecessary overhead; only those references requiring action incur the procedure call overhead of invoking the simulator. The active memory abstraction hides the implementation of steps 1–2 from the simulator, allowing the simulator to be written in a high-level language.

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

Active Memory Run-Time System Provided	
`read_state(address)`	Return block state.
`write_state(address,state)`	Update block state.
User Written	
user_handler`(address)`	Invoked for action.
`sim_init()`	Simulator startup routine.
`sim_exit()`	Simulator exit routine.

Table 1: Active Memory Interface

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 fixed-size blocks, each with a user-defined state. Users—i.e., simulator writers—can specify, in a configuration file, 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.

Users can identify cases that require no simulator action by specifying the predefined NULL handler. Doing so allows the active memory system to implement this case as efficiently as possible, without breaking the abstraction: while this paper focusses on software implementations, active memory can also be supported using the same hardware required for trap-driven simulations.

The example in Figure 1 illustrates how we can use active memory to implement a simple data-cache simulation that counts cache misses. The user specifies the cache block size ($2^5 = 32$ bytes) and the functions to be invoked on each combination of reference and state; i.e., a `load` to an `invalid` block invokes the `miss_handler` routine. The function `noaction` is the predefined NULL handler. The simple miss handler increments the miss count, selects a victim block using a user-written routine (not shown), 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.

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, many simulators that evaluate multiple cache configurations [6, 23, 12] use the property of inclusion [12] 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 optimizes these references with the NULL handler. This same technique can be used to efficiently simulate multiple cache configurations that *do not* maintain inclusion. When action is required, the simulator can use

```
/* Active Memory configuration */
/* for a simple cache simulation */

lg2blocksize 5 /* log base 2 of the block size */

LOADS
invalid miss_handler /* user handler to call */
valid   noaction     /* predefined NULL handler */

STORES
invalid miss_handler /* user handler to call */
valid   noaction     /* predefined NULL handler */


/* Simple Active Memory Handler (pseudo-code) */
miss_handler(void *address)
{
      miss_count++;
      victim_address = select_victim(address);
      write_state(address,valid);
      write_state(victim_address,invalid);
}

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

Figure 1: Simple Data-Cache Simulator Using Active Memory

the state to encode which caches contain a particular block and directly invoke a function specialized to update the appropriate caches. Simple simulations of a single cache benefit primarily from the efficiency of the predefined NULL handler.

4 Fast-Cache

In this section we present Fast-Cache, our implementation of active memory for SPARC processors. The active memory abstraction 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 the block's state by using the effective address as an index into this table, invoking an action only if necessary. 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 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 is required to shift the effective address to a table offset. By storing the base of the state table in an otherwise unused global register,[2] a third instruction is sufficient to load the state byte. 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.

The code inserted to test the state, and determine whether an action is required, depends on whether the condition codes are live. 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 from user mode. Thus, Fast-Cache generates two different test sequences depending upon whether the condition codes are live (i.e., will be used by a later branch instruction) or not.

In the common case, the condition codes are dead and Fast-Cache inserts a simple two instruction sequence that masks out the appropriate bits (loads and stores must check different state bits) and branches. We expect the common case to be no action, so the branch target is the next instruction in the original program. If action is required, the branch falls through into a four instruction "trampoline" which 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 [26]. This sequence could be further optimized on the SuperSPARC by scheduling independent instructions from the original program with the Fast-Cache inserted instructions.

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. No action cases invoke a NULL handler (literally a `return` and a `nop`), which requires 9 instructions, taking 7 cycles on the SuperSPARC.

[2] Registers `%g5, %g6, and %g7` are specified as reserved in the SPARC ABI.

When action is required, Fast-Cache invokes user handlers through a stub that saves processor state. 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.

The table lookup instructions could be inserted with any instrumentation methodology. Fast-Cache uses the EEL system [10], 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.

For trace-driven simulation, we consider an on-the-fly simulator that performs a procedure call to perform the lookup [21, 11]. To maintain a clean interface between the reference generator and the simulator, processor state is saved before invoking the simulator. Our implementation inserts two instructions before each memory reference that compute the effective address and jump 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. On a SuperSPARC processor, the lookup overhead is roughly 25 cycles when we can use the streamlined stub. Most of this overhead is the procedure call linkage; 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.

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 [25, 24, 16]. Therefore, trap-driven simulation performance will be highly dependent on the target miss ratio.

Fast-Cache's lookup overhead is roughly 4 cycles (the mean lookup overhead for typical programs). However, the miss processing overhead, roughly 55 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.

We can obtain 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. We use a metric called *slowdown* to evaluate the different simulation techniques. Slowdown is 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:

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

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 $r \cdot I_{orig}$ 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, $r \cdot I_{orig}$, and the overhead of simulating a single target cache miss, C_{miss}.

We can simplify Equation 1 and express the slowdown

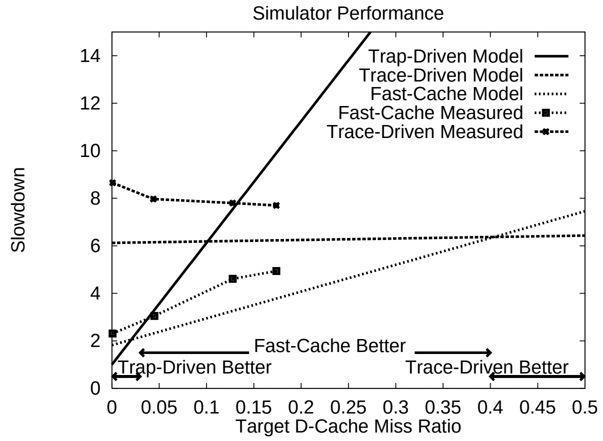

Figure 2: Qualitative Performance

The simulator parameters are miss processing overhead, C_{miss}, of 250 for trap-driven, 3 for trace-driven, and 55 for Fast-Cache, and lookup overhead, C_{lookup}, of 0 for trap-driven, 25 for trace-driven, 4 for Fast-Cache.

as a function of the target miss ratio m:

$$Slowdown = 1 + \frac{r}{CPI_{orig}}(C_{lookup} + m \cdot C_{miss}) \quad (2)$$

where CPI_{orig} is cycles-per-instruction $\frac{C_{orig}}{I_{orig}}$.

We can use Equation 2 to get a rough idea of the relative performance of the various simulation techniques. Figure 2 shows simulator slowdown versus target miss ratio, using a CPI_{orig} of 1.22 and reference ratio $r = 0.25$ (derived from the SPEC92 benchmark program `compress` [20]). The simulator parameters are miss processing overhead, C_{miss}, of 250 for trap-driven, 3 for trace-driven, and 55 for Fast-Cache, and lookup overhead, C_{lookup}, of 0 for trap-driven, 25 for trace-driven, 4 for Fast-Cache.

The results in Figure 2 confirm our expectations. Trace-driven simulation has very little dependence on target miss ratio since it incurs a high overhead for each memory reference. 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. Our model indicates that Fast-Cache performs better than trap-driven simulation for miss ratios greater than 2.5% and better than trace-driven simulation for miss ratios less than 40%. This model suggests that Fast-Cache is always superior to trace-driven simulation in practice, since caches rarely have such high miss ratios. 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 [8]. Furthermore, if the hardware is available, we can implement the active memory abstraction using the trap-driven technique as well. Thus the active memory abstraction gives the best performance over most of the design space.

To verify our simple model we measured the slowdowns of Fast-Cache and the fast trace-driven simulator described above. The results, shown in Figure 2, indicate that Fast-Cache is 1.5 to 3.4 times faster than the trace-driven simulator, over the range of target caches we simulated (16KB–1MB). More importantly, these measured slowdowns corroborate the general trends predicted by the model. However, the model clearly omits some important factors: the trace-driven simulator is at least a full-factor slower than predicted, while Fast-Cache is up-to a factor slower than predicted.

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: $C_{lookup} = f_{cc} \cdot C_{cc} + (1 - f_{cc}) \cdot C_{nocc}$. Substituting into Equation 2 yields a more accurate slowdown model:

$$\begin{aligned} Slowdown_{Inst} &= 1 + \frac{r}{CPI_{orig}}(f_{cc} \cdot C_{cc} + \\ &\quad (1 - f_{cc}) \cdot C_{nocc} + m \cdot C_{miss}) \end{aligned} \quad (3)$$

$Slowdown_{Inst}$ 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:

$$\begin{aligned} Slowdown &= Slowdown_{Inst} + Slowdown_{D-Cache} \\ &\quad + Slowdown_{I-Cache} \end{aligned} \quad (4)$$

Section 6.1 investigates Fast-Cache's impact on the host data cache, and computes an estimate for its affect, $Slowdown_{D-Cache}$. Section 6.2 develops a model for Fast-Cache's instruction cache behavior, and an estimate for $Slowdown_{I-Cache}$. 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, $Slowdown_{D-Cache}$, 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, $Slowdown^{lower}_{D-Cache}$, 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 $2 \cdot r \cdot I_{orig} \cdot C_{hostmiss}$ 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 this paper, $B_h = 6$. 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 $2 \cdot r \cdot I_{orig} \cdot m \cdot B_h \cdot C_{hostmiss}$ cycles are added to the simulation time. Equation 5 shows the upper bound on the slowdown resulting only from data cache effects.

$$Slowdown^{upper}_{D-Cache} = \frac{2 \cdot r \cdot C_{hostmiss}}{CPI_{orig}} \left(1 + m \cdot B_h\right) \tag{5}$$

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 our host—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 [1].

Bench	**Insts**(10^9)	**Refs**(10^9)	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	5.82	1.53	0.26	0.98	1.38

Table 2: Benchmark Characteristics

To validate our model, we use 4 programs from the SPEC92 benchmark suite [20]: `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. Program characteristics are shown in Table 2.

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.

Figure 3 plots the measured and modeled slowdowns as a function of target miss ratio. The lowest line is $Slowdown_{Inst}$, the asymptotic lower bound. The upper 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 [27]. To compute a single estimator of data cache performance, we calculate the mean of the upper and lower bounds:

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

As Figure 3 shows, this estimator—although simplistic—is quite accurate, predicting slowdowns within 30 of the measured values%.

6.2 Instruction Cache Effects

The $Slowdown_{split}$ estimator is accurate despite ignoring instruction cache pollution. This is because most

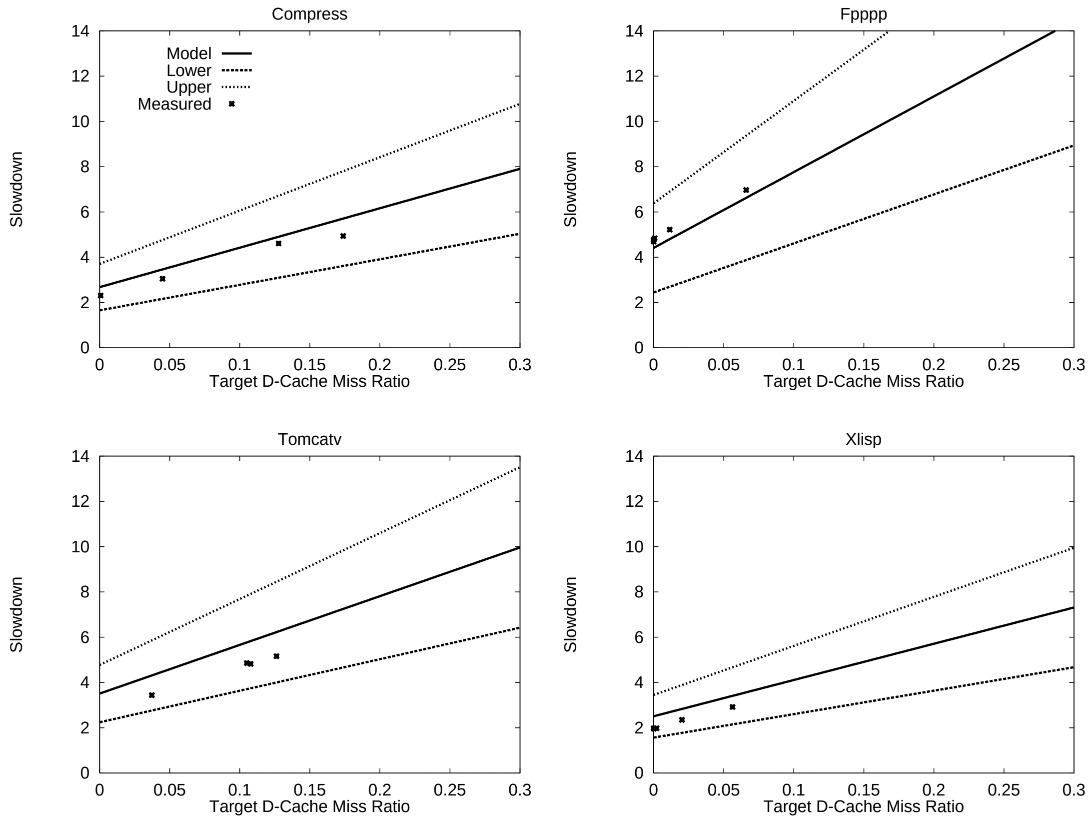

Figure 3: Fast-Cache Performance

of the SPEC benchmarks have extremely low instruction cache miss ratios on the SPARCstation 10/51 [5]. 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. Let's 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 only 1024 of the original instructions, on average. 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.

We can estimate the effect of a scaled cache using design target miss ratios [19] and other available data [5]. 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. [5] 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 equal to $E \cdot \sqrt{E}$ times the original number of instruction cache misses. Since the original program incurs $I_{orig} \cdot m_i$ misses, Fast-Cache incurs an additional slowdown of:

$$Slowdown_{I-Cache} = (E \cdot \sqrt{E} - 1) \cdot m_i \cdot C_{hostmiss} \quad (7)$$

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 $I_{nocc} = 7$ instructions when it cannot, then the total code expansion is simply:

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

Since the total code expansion, see Table 3, 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, and not fitting.

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 [13]. 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, our model predicts that the instrumented code will have only $1.6 \cdot \sqrt{1.6} = 2$ 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 our benchmarks, Fast-Cache-Indirect executes 3.3 to 8 times slower than the original program. This is 1.1 to 1.7 times slower than Fast-Cache.

To validate our instruction cache models we used *Shade* [3] to measure the instruction cache performance of the instrumented programs.[3] 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 3 shows how well our 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.

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, $E \cdot \sqrt{E}$ captures general trends, but is not a precise predictor. For example, the 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 our 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 $E \cdot \sqrt{E}$ predicts the number of instruction cache misses within 15% for Fast-Cache and 10% for Fast-Cache-Indirect. To further evaluate this model we used the reference counter of the SuperSPARC second-level cache controller [1] 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.) $E \cdot \sqrt{E}$ predicts the number of misses within 36% for Fast-Cache and 4% for Fast-Cache-Indirect.

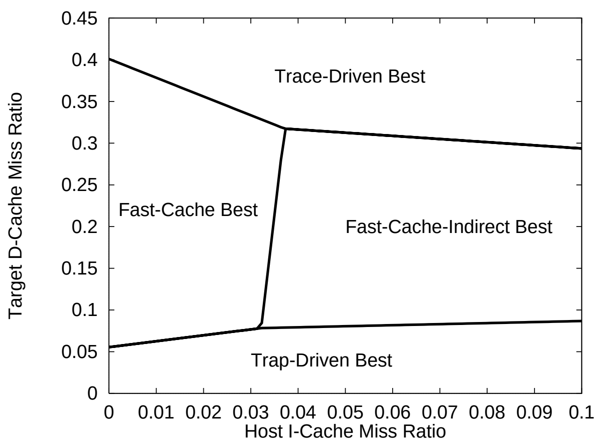

Figure 4: Simulator Performance

$r = 0.25, f_{cc} = 0.95, CPI = 1.22, B_h = 6$
Fast-Cache: $C_{cc} = 4, C_{nocc} = 7, C_{miss} = 55,$
$I_{cc} = 9, I_{nocc} = 7$
Fast-Cache-Indirect: $C_{cc} = 7, C_{nocc} = 9, C_{miss} = 57,$
$I_{cc} = I_{nocc} = 2$
Trace-Driven: $C_{lookup} = 25, C_{miss} = 3, I_{cc} = I_{nocc} = 2$
Trap-Driven: $C_{miss} = 250$

6.3 Overall Performance

We now use our detailed model to revisit the comparison between Fast-Cache, trap-driven and trace-driven simulation. Figure 4 compares the detailed performance model for Fast-Cache and Fast-Cache-Indirect against 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 data cache pollution. The comparison shows that either Fast-Cache or Fast-Cache-Indirect performs best over an important region of the design space.

Incorporating the cache pollution caused by Fast-Cache's additional instructions and data references allows us to predict Fast-Cache's performance on a SPARCstation 10/51 to within 32% of measured values, using our scaled cache model, and 36% using $E \cdot \sqrt{E}$. For three of the programs we ran, instruction cache pollution has little effect on Fast-Cache simulation time. For `fpppp`, Fast-Cache-Indirect performs nearly as well

[3] 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.

Benchmark	Original Misses (m_i)	Measured Misses	Scaled Model	$E \cdot \sqrt{E}$ Model	Code Exp
Fast-Cache					
Compress	329 (0.000%)	1,843	984 (46%)	1,848 (00%)	3.16
Fpppp	336,224 (3.731%)	4,629,793	4,361,246 (06%)	3,929,520 (15%)	5.15
Tomcatv	1,402 (0.007%)	27,143	33,680 (24%)	12,414 (54%)	4.28
Xlisp	1,538 (0.003%)	578,773	442,077 (24%)	9,984 (98%)	3.48
Fast-Cache-Indirect					
Compress	329 (0.000%)	1,221	458 (62%)	592 (50%)	1.48
Fpppp	336,224 (3.731%)	1,033,342	954,609 (08%)	922,598 (10%)	1.96
Tomcatv	1,402 (0.007%)	5,935	7,847 (32%)	3,385 (42%)	1.80
Xlisp	1,538 (0.003%)	13,670	14,890 (09%)	2,882 (78%)	1.52

Table 3: Instruction Cache Performance

as Fast-Cache, and when simulating programs with larger instruction cache miss ratios, Fast-Cache-Indirect should be a better implementation.

7 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 presents a new method for memory system simulation—the *active memory* abstraction—designed specifically for on-the-fly simulation. Active memory associates a state with each memory block, specifying 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 allows 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 7 times slower than the original, un-instrumented program on a SPARCstation 10; a fast trace-driven simulator is 7 to 16 times slower than the original program. The models show that Fast-Cache will perform better than trap-driven or trace-driven simulation for target miss ratios between 5% and 40%, *even* when we account for cache interference for Fast-Cache but not for the other simulators.

Our 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 misses as the original program. Although the instruction cache miss ratios for our applications 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 instructions 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.

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.

References

[1] *SuperSPARC User's Guide*, 1992. Alpha Edition.

[2] 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.

[3] 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.

[4] 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 II99–107, August 1991.

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

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

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

[8] 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.

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

[10] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, June 1995. To Appear.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] SPEC. SPEC Newsletter, Dec 1991.

[21] 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 (PLDI)*, pages 196–205, June 1994.

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.