Tools and Techniques for Memory System Design and Analysis

by

Alvin R. Lebeck

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1995

Abstract

As processor cycle times decrease, memory system performance becomes ever more critical to overall performance. Continually changing technology and workloads create a moving target for computer architects in their effort to design cost-effective memory systems. Meeting the demands of ever changing workloads and technology requires the following:

- Efficient techniques for evaluating memory system performance,
- Tuning programs to better use the memory system, and
- New memory system designs.

This thesis makes contributions in each of these areas.

Hardware and software developers rely on simulation to evaluate new ideas. In this thesis, I present a new interface for writing memory system simulators—the *active memory* abstraction—designed specifically for simulators that process memory references as the application executes and avoids storing them to tape or disk. Active memory allows simulators to optimize for the common case, e.g., cache hits, achieving simulation times only 2-6 times slower than the original un-instrumented application.

The efficiency of the active memory abstraction can be used by software designers to obtain information about their program's memory system behavior—called a cache profile. In this thesis, using the CProf cache profiling system, I show that cache profiling is an effective means of improving uniprocessor program performance by focusing a programmer's attention on problematic code sections and providing insight into the type of program transformation to apply. Execution time speedups for the programs studied range from 1.02 to 3.46, depending on the machine's memory system.

The third contribution of this thesis is *dynamic self-invalidation* (DSI), a new technique for reducing coherence overhead in shared-memory multiprocessors. The fundamental concept of DSI is for processors to automatically replace a block from their cache before another processor wants to access the block, allowing the directory to immediately respond with the data. My results show that, under sequential consistency DSI can reduce execution time by as much as 41% and under weak consistency by as much as 18%. Under weak consistency, DSI can also exploit tear-off blocks—which eliminate both invalidation and acknowledgment messages—for a total reduction in messages of up to 26%.

Acknowledgments

While pursuing my Ph.D. I have relied on the support, encouragement, friendship and guidance of many people. Now, I finally get to thank them.

First, and most important, is my wife, Mitali. Her support and encouragement throughout my graduate studies was never ending. Despite late nights, early mornings, and general “bad attitudes”, she continually did whatever it took to get me through.

The rest of my family provided the little things that make life meaningful and easier. My sons, Kiron and Niel, brought out the child in me for the last three and a half years. I’ve relied on my parents support and overwhelming confidence in me for more years than I can count. Mitali’s parents help and encouragement was always there when needed. Finally, my siblings, who provided sounding boards and family gatherings that allowed me to temporarily escape from the daily grind.

Within the Computer Sciences department, many people have influenced me, and I can not list all of them here.

My advisor, David Wood, provided essential leadership and helped me improve many of the skills necessary to perform high quality research. I hope I can provide the same level of mentoring to my future students.

Mark Hill has been a mentor since my undergraduate days. He created, and continues to create, opportunities for me to perform interesting work and expand my contacts in the research community. Most importantly, his skills at oral and written presentation have had a lasting impact on me.

Jim Larus and Guri Sohi provided advice and support throughout most of graduate studies. Jim’s work on executable editing (QPT and EEL) provided crucial infrastructure required to complete most of the work presented in this thesis. Guri encouraged me to write a master’s thesis, if I really wanted to be a professor; it was a very rewarding experience. He also suggested the work on self-invalidation.

Steve Reinhardt was the biggest sounding board for ideas I’ve ever had. On numerous occasions, mostly getting coffee, he helped me iron out many details.

Finally, the members of the Wisconsin Wind Tunnel project and the computer architecture group provided support and lively discussions/arguments that added to my understanding of many aspects of computer systems.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Technological advances have enabled the implementation of microprocessors with clock cycle times similar to supercomputers. As a current example, the Digital Equipment Corporation Alpha 21164 microprocessor [21] has a 3.3ns clock cycle time, while the Cray Research, Inc. T90 supercomputer has a 2ns cycle time [32]. The small difference in cycle times, together with the ability to produce multiple floating point results per cycle [31], implies that the peak performance of microprocessors is approaching that of supercomputers.

Unfortunately, the memory system performance of microprocessors is not comparable to that of supercomputers, creating a significant gap in sustained execution rates. Fast processors require fast memory accesses to achieve peak performance; supercomputers achieve this by using vast amounts of expensive static RAM (SRAM). Unfortunately, this approach is too expensive for most users, who require a more cost-effective solution.

Dynamic RAM (DRAM) is a cheaper, but slower alternative to SRAM. However, a memory system built entirely out of DRAM could not satisfy the access time requirements of fast processors. Instead, computer architects utilize cache memories to create a cost-effective alternative to supercomputer memory systems. Caches are a technique for using a small amount of fast SRAM in combination with large amounts of slow, inexpensive DRAM.

Caches sit between the (fast) processor and (slow, inexpensive) main memory, and hold regions of recently referenced main memory. References satisfied by the cache—called *hits*—proceed at processor speed; those unsatisfied—called *misses*—incur a cache miss penalty to fetch the corresponding data from main memory. Caches work because most programs exhibit significant locality. *Temporal* locality exists when a program references the same memory location multiple times in a short period. Caches exploit temporal locality by retaining recently referenced data. *Spatial* locality occurs when the program accesses memory locations close to those it has recently accessed. Caches exploit spatial locality by fetching multiple contiguous words—a *cache block*—whenever a miss occurs.

Since a cache satisfies most references, a cost-effective memory system can be designed by using slower, less expensive memory chips for main memory. This approach sacrifices performance for lower cost. To achieve supercomputer performance, multiple cost-effective microprocessors can be connected into a multiprocessor system [31]. Cache-coherent shared-memory multiprocessors exploit the fast access times of cache memories by allowing shared data to reside in a microprocessor's cache. Most of these systems implement a coherence policy to ensure that a processor does not reference stale data.

Although caches are a cost-effective alternative to supercomputer memory systems, no single memory system design satisfies all requirements. Each generation of microprocessors places new constraints (e.g., chip area, access time) on cache design, and architects continually search for the most cost-effective solution. Meeting the demands of ever changing technology and workloads, to achieve good memory system performance, requires the following three components. First, new techniques for *efficiently* evaluating memory system performance *must* be developed. Second, programmer's *must* tune their program's memory system behavior to better use the memory system. Third, architects *must* continually design new memory systems.

This thesis makes contributions in each of these three areas, with a focus on cache memories. *Active memory* is a new approach for rapidly simulating memory systems, designed specifically for on-the-fly memory system simulators that process memory references as the application executes. *CProf*, my second contribution, is a tool that uses simulation to provide insight on a program's memory system performance, allowing the programmer to restructure their code to improve locality. The current implementation of CProf uses active memory to efficiently simulate the desired memory system. The final contribution of this thesis is *dynamic self-invalidation*, a new memory system design for cache-coherent shared-memory multiprocessors that reduces the overhead of ensuring that a processor does not reference stale data. The following sections describe each of these contributions in more detail.

1.2 Active Memory

There are many alternatives for evaluating memory system performance, each having advantages and disadvantages. At one extreme, hardware prototyping permits the accurate evaluation of a particular design. Unfortunately, it is expensive, time consuming, and only allows evaluation of one design. At the other extreme is analytical modeling, which permits rapid evaluation of a large design space. However, it is difficult to accurately model memory system behavior. Instead, designers often use discrete event simulations that simulate the memory system behavior for each memory access in an application program.

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. Fortunately, simulation times can be reduced using a new simulation abstraction. This thesis examines *active memory* [47], a new memory system simulation abstraction designed specifically for on-the-fly simulation.

Conventional simulators rely on the *reference trace* abstraction: a reference generator produces a list of addresses that are processed by the simulator. These simulators incur significant overhead for each memory reference, even though the common case in many simulations—cache hits—require no action by the simulator. Active memory allows implementations to optimize for this common case, by tightly integrating reference generation and simulation. In this abstraction, memory is logically partitioned into fixed-size blocks, each with a user-defined state. Each memory reference logically invokes a user-specified function depending on the reference's type and current state of the accessed memory block. Simulators control which function gets invoked by manipulating the memory block states. The abstraction provides a predefined function that simulator writers can specify for the common, no-action case. Active memory implementations can optimize this no-action function depending on available system features (e.g., in-line software checks, or error correcting code (ECC) bits and fast traps.)

Fast-Cache, my all software implementation of the active memory abstraction for SPARC processors, eliminates unnecessary instructions for the no-action case. As few as 3 cycles are required for the critical no-action case on a SuperSPARC processor by using a simple in-line table lookup of the appropriate memory block state. Using Fast-Cache, simple data cache simulations (i.e., counting misses for a direct-mapped cache) execute only 2 to 6 times slower than the original program on a SPARCstation 10/51. This is two to three times faster than published numbers for highly optimized trace-driven simulators [71].

As the importance of memory hierarchy performance increases, hardware and software developers will increasingly rely on simulation to evaluate new ideas. The active memory abstraction provides the framework necessary to efficiently perform these simulations when more than 80% of the memory references do not require action by the simulator. Most interesting memory system designs fall into this range.

1.3 CProf

New cache memory designs can improve performance on future machines but do not help programmers that must use existing machines. Furthermore, caches only work well for programs that exhibit sufficient spatial and temporal locality. Programs with unruly access patterns spend much of their time transferring data to and from the cache. To fully exploit the performance potential of fast processors, cache behavior must be explicitly considered and codes restructured to increase locality.

Although compilers can perform this restructuring for some codes (e.g. regular scientific applications) [41,57] most programs are too complex for compilers to analyze. Therefore, programmers must be aware of their program's cache behavior and restructure their code to better use the memory hierarchy. To accomplish this, programmers must be given more than a simple indication of where their program suffers from poor cache behavior—*programmers must be given insight on how to fix cache performance problems*.

Traditional execution time profilers are insufficient for this purpose, since they generally ignore cache performance. Instead, the programmer needs a profile that focuses specifically on a program's cache behavior, identifying problematic code sections and data structures and providing insight for determining which program transformations will improve performance.

This thesis describes CProf, a cache profiler, and some of the techniques that programmers can use to improve cache performance [46]. CProf maps cache misses to individual source code lines and data structures. It also provides information on the cause of the cache misses by classifying them according to Hill's 3C model [30]. By knowing the cause of the cache misses, programmers gain insight for determining which program transformations are likely to improve cache performance. Most of the transformations are well known and include array merging, padding and aligning structures, structure and array packing, loop interchange, loop fusion, and blocking.

Using CProf and the set of simple transformations, I show how to tune the cache performance of six of the SPEC92 benchmarks. Restructuring their source code greatly improves cache behavior and achieves execution time speedups ranging from 1.02 to 3.46. The speedup depends on the machine's memory system, with greater speedups obtained in the Fortran programs which generally produced non-sequential access patterns when traversing arrays.

Cache performance will become more important as processor cycle times continue to decrease faster than main memory cycle times. CProf provides cache performance information at the source line and data structure level allowing programmer's to identify problematic code sections. The insight CProf provides helps programmers determine appropriate program transformations that improve a program's cache performance. This makes tools like CProf a valuable component of a programmer's tool box.

1.4 Dynamic Self-Invalidation

Microprocessors are designed for a high volume market and sacrifice performance for lower cost. Therefore, a single microprocessor is unlikely to achieve sustained performance equal to supercomputers. Instead, system designers attempt to achieve this performance by connecting multiple microprocessors and exploiting parallelism. Although there are many different approaches for designing multiprocessors (e.g., message passing, shared-memory, data parallel), in this thesis I focus on the memory system of cache coherent shared-memory multiprocessors.

Shared-memory multiprocessors simplify parallel programming by providing a single address space even when memory is physically distributed across many workstation-like processor nodes. Cache coherent shared-memory multiprocessors use caches to automatically replicate and migrate shared data and implement a coherence protocol to maintain a consistent view of the shared address space [9,28,40,49].

Conventional invalidation-based protocols send messages to explicitly invalidate outstanding copies of a cache block whenever a processor wants to modify it. The performance of these protocols could improve if we could eliminate the invalidation messages without changing the memory semantics. This can be accomplished by having processors replace the appropriate block from their cache just before another processor wants to modify it. This would allow the processor to immediately obtain the data, eliminating the need to send an invalidation message. Having processors *self-invalidate* blocks from their cache, instead of waiting for an invalidation message, would reduce latency and bandwidth requirements, potentially improving performance. However, if the block is self-invalidated too early, performance could decrease because of additional cache misses.

This thesis investigates *dynamic self-invalidation* (DSI) [48], a new technique for reducing cache coherence overhead in shared-memory multiprocessors. DSI eliminates invalidation messages by having a processor automatically invalidate its local copy of a cache block before a conflicting access by another processor. DSI is applicable to software, hardware, and hybrid coherence schemes. In this thesis I evaluate DSI in the context of hardware coherence protocols.

I investigate several practical techniques for dynamically identifying which blocks to self-invalidate and for the cache controller to self-invalidate the blocks. My results show that DSI reduces execution time of a sequentially consistent [42] full-map coherence protocol by as much as 41%. This performance impact is comparable to using an implementation of weak consistency [2,20,24] with a coalescing write-buffer to allow up to 16 outstanding requests for exclusive blocks. When used in conjunction with weak consistency, DSI can exploit tear-off blocks—which eliminate both invalidation and acknowledgment messages—for a total reduction in messages of up to 26%. Although my results indicate that DSI decreases execution time in a limited number of cases, reducing the number of messages in the system reduces contention in the network and at the directory controller which may reduce execution time on other systems.

DSI is a general technique, applicable to hardware [49], software [36,63], and hybrid [40,61] cache coherent shared-memory multiprocessors. Current trends in parallel architectures, e.g., faster processors and larger caches, can make coherence overhead a significant fraction of execution time. If this trend continues, DSI should be of increasing benefit.

1.5 Thesis Organization

This thesis is organized as follows. Chapter 2 describes the active memory abstraction and analyzes the performance of my implementation (Fast-Cache). Chapter 3 discusses cache profiling and shows how I used CProf to tune performance of six of the SPEC benchmarks. Dynamic self-invalidation is examined in Chapter 4. Chapters 2, 3 and 4 of this thesis are based on previous publications. Chapter 5 gives conclusions and suggests areas for future work. Appendix A provides details on Fast-Cache’s implementation. Appendix B describes the dynamic self-invalidation cache coherence protocols.

Chapter 2

Active Memory

2.1 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. [23], estimate that *17 months* of processing time were used to obtain miss ratios for the SPEC92 benchmarks [70].

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 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 chapter examines *active memory*,[1] 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 2.3, 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

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

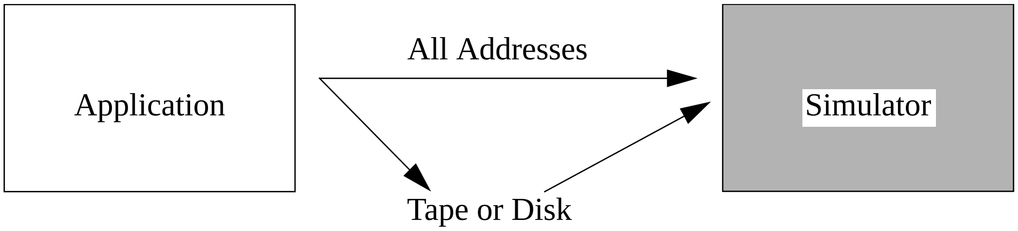


Figure 1: Trace-Driven Simulator

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 one using the traditional trace abstraction (10 cycles for the no-action case).

I 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 [71].

As described in Section 2.4, 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 2.5 analyzes the performance of Fast-Cache by modeling the effects of the additional lookup instructions. I 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, I 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 2.6 extends this model by incorporating the cache pollution caused by the additional instructions inserted by Fast-Cache. For data caches, I 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 model—that splits the difference between the two bounds—predicts the actual performance within 30%. For instruction caches, I 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, I present an alternative implementation, Fast-Cache-Indirect, that reduces code dilation to 2 static instructions at the expense of 3 more instructions for the “no action” case.

Section 2.7 discusses how to use the active memory abstraction for simulation more complex than simple miss counting.

2.2 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 [33]. 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.

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 [66] 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 [58] extended this work to set-associative memories by filtering references to a direct-mapped cache.

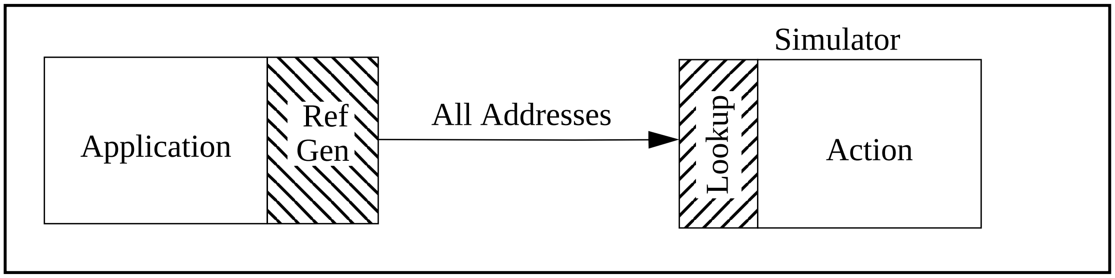


Figure 2: On-The-Fly Simulator

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 [43]. 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 [19,57,13,6].

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 [19,57], or (ii) buffer the reference in memory, incurring buffer management overhead and memory system delays caused by cache pollution [6,73]. 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 [51] 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 [71]. However, ATOM still incurs unnecessary procedure linkage overhead in the no-action cases.

A recent alternative technique, *trap-driven simulation* [60,78], optimizes "no action" cases to their logical extreme. Trap-driven simulators exploit the characteristics of the

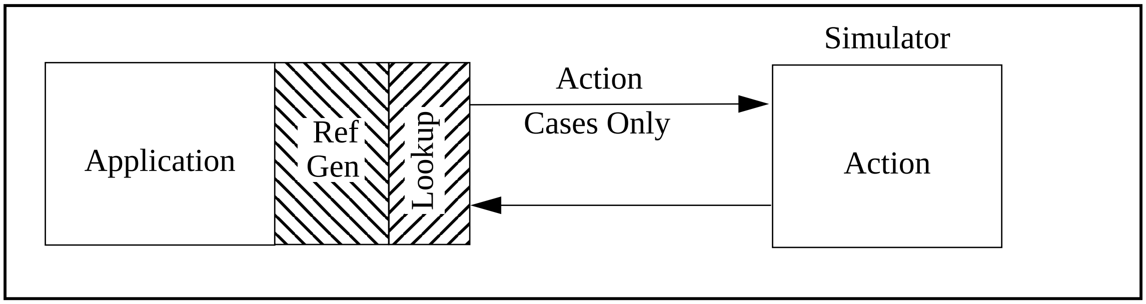


Figure 3: Active Memory Simulator

simulation platform to implement effective address calculation and lookup (steps 1 and 2) in hardware using error correcting code (ECC) bits [60] or valid bits in the TLB [78]. 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 [60] does not simulate stack references because of SPARC register windows, while Tapeworm II [78] does not simulate any data references because of write buffers on the DECstation. Furthermore, as I show in Section 2.5, the overhead of memory exceptions (roughly 250 cycles [78,77,59] 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 2.4 describes my 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 (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
sim_exit()	Simulator exit routine

Table 1: Active Memory Interface

2.3 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.

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 chapter focuses on software implementations, active memory can also be supported using the same hardware required for trap-driven simulations (see Figure 4b).

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 2.7). 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. The simple 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`

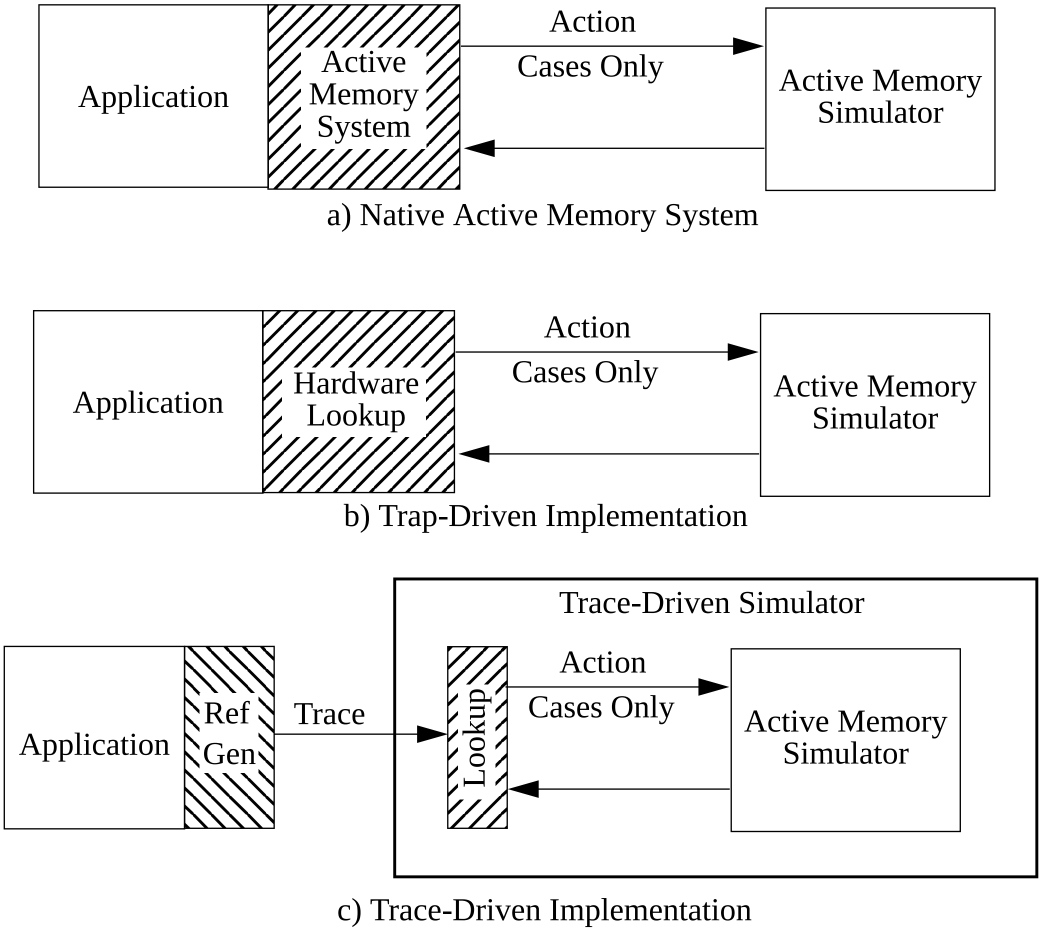


Figure 4: Active Memory Implementations

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.

2.4 Fast-Cache

This section describes Fast-Cache, my 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.

```
/* 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) */
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

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 Appendix A) 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, I do not currently implement this optimization. An additional instruction shifts the effective

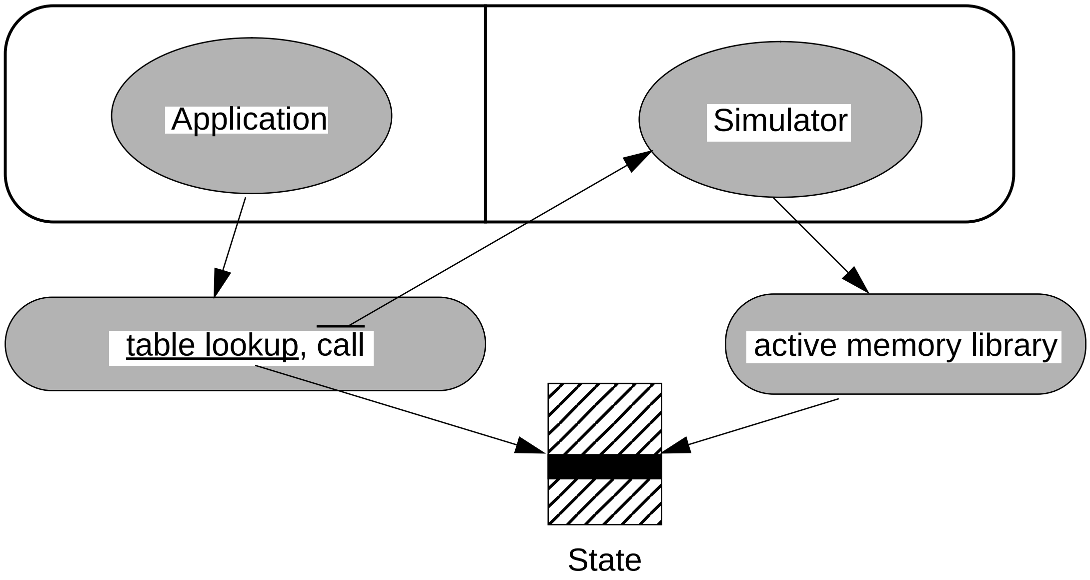


Figure 6: Fast-Cache Implementation

address to a table offset. By storing the base of the state table in an otherwise unused global register,[1] a third instruction suffices 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 2.2. I 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 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. 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.) I 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

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

branch falls through into a four instruction “trampoline” that jumps to the handler stub. Since I 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 [79]. 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, I cannot use a branch instruction. Instead, I 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.

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 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 [45], 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.

2.5 Qualitative Analysis

In this section I use a simple model to qualitatively compare the performance of Fast-Cache to trace-driven and trap-driven simulators. In Section 2.6, I 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, I focus on a simple miss-count simulation for direct-mapped data caches with 32-byte blocks—called the *target* cache. To simplify the discussion, I lump effective address calculation and action lookup into a single *lookup* term. Similarly, I lump action simulation and metric update into a single *miss processing* term.

For trace-driven simulation, I consider two on-the-fly simulators: one invokes the simulator for each memory reference (via procedure call) [71,51], 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 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, my 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 I can use the streamlined stub. Most of this overhead is the procedure call linkage, which could be reduced using techniques similar to ATOM’s. 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 (see Appendix A). 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. Like the procedure call implementation, I expect this technique 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 [78,77,59]. 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.

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

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.

I 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 2.6 extends the model to include these effects. I 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:

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

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 $r \cdot I_{\text{orig}}$ 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, I express it as a function of the target cache miss ratio, *m*, multiplied by the number of memory references, $r \cdot I_{\mathrm{orig}}$, and the overhead of simulating a single target cache miss, C_{miss}.

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

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

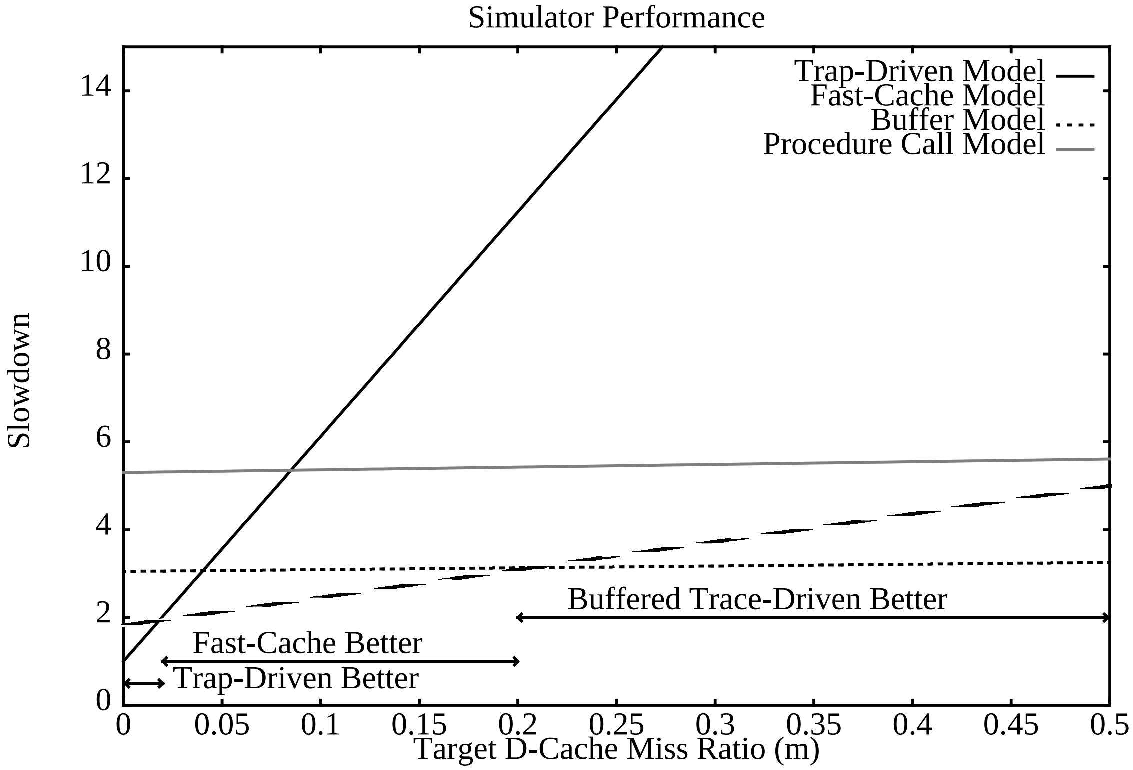
where $\mathrm{CPI}_{\mathrm{orig}}$ is cycles-per-instruction, $\frac{C_{\mathrm{orig}}}{I_{\mathrm{orig}}}$.

I can use Equation 2 to get a rough idea of the relative performance of the various simulation techniques. Figure 7 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` [70]). 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 7 confirm my 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



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 7: Qualitative Simulator Performance

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 requires the program counter of the memory reference. Buffering this additional information will inevitably slow down the simulation.

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

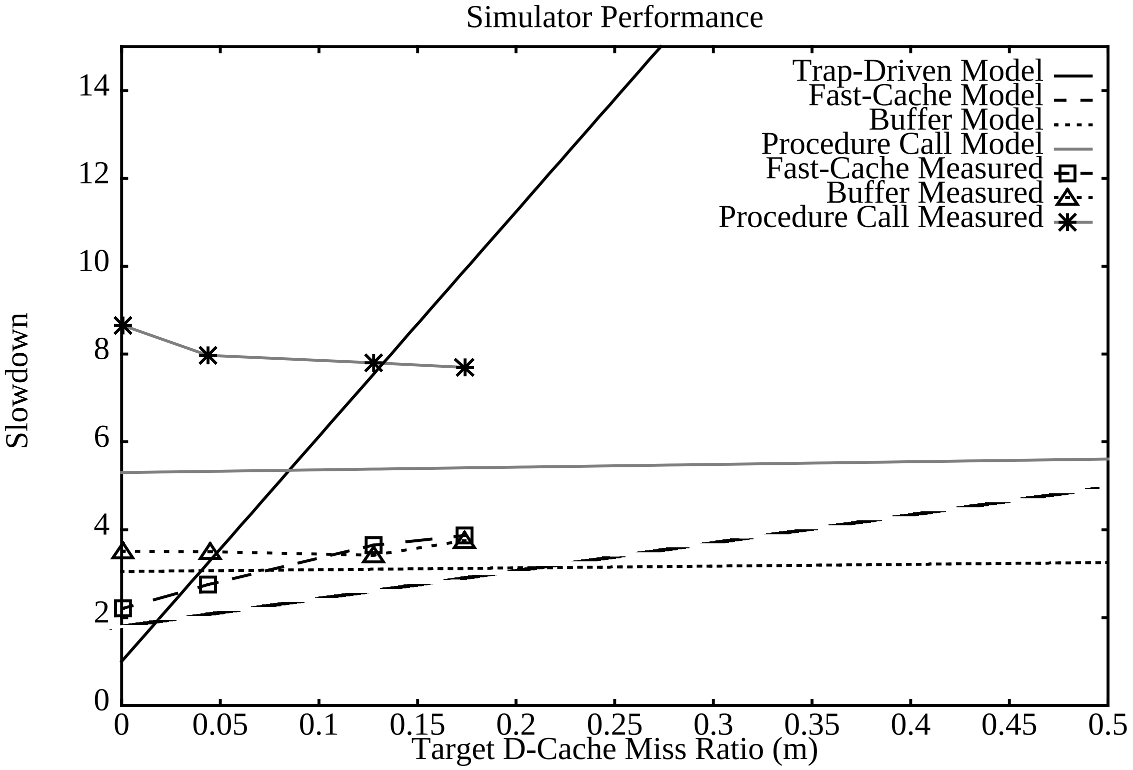


Figure 8: Measured Simulator Performance

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 [19,60] can not be implemented with the buffered simulator since the data may not be valid at the time of the reference. Therefore, I 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 [35]. 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

To verify the simple model, I measured the slowdowns of Fast-Cache and the two trace-driven simulators. The results, shown in Figure 8 and Figure 9, indicate that over the range of target caches I 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

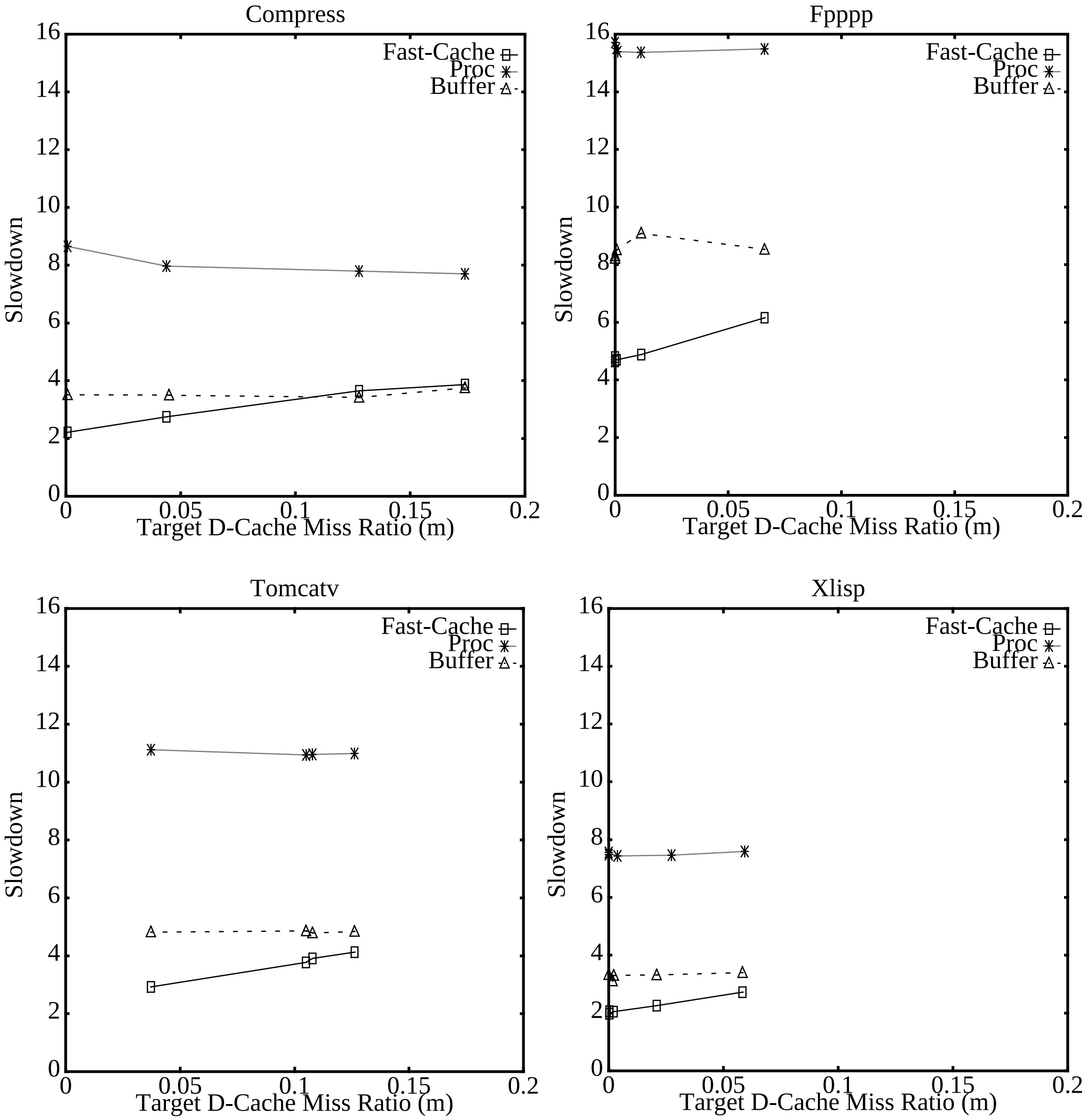


Figure 9: Measured Simulator Performance

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 a full-factor slower than predicted, and Fast-Cache is up-to a factor slower than predicted.

2.6 Detailed Analysis

The model derived in Section 2.5 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, I extend Equation 2 to incorporate the details of a Fast-Cache implementation executing on a SPARCstation 10/51.

First, I refine lookup overhead, which depends on whether or not Fast-Cache can use the SPARC condition codes. The lookup requires $C_{\mathrm{cc}} = 3$ cycles when the condition codes can be used and $C_{\mathrm{nocc}} = 7$ 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: $C_{\mathrm{lookup}} = f_{\mathrm{cc}} \cdot C_{cc} + (1 - f_{\mathrm{cc}}) \cdot C_{nocc}$. Substituting into Equation 2 yields a more accurate slowdown model:

$$\mathrm{Slowdown}_{\mathrm{Inst}} = 1 + \frac{r}{\mathrm{CPI}_{\mathrm{orig}}}\left(f_{\mathrm{cc}} \cdot C_{cc} + (1 - f_{\mathrm{cc}}) \cdot C_{nocc} + m \cdot C_{\mathrm{miss}}\right) \qquad \text{EQ 3.}$$

$\mathrm{Slowdown}_{\mathrm{Inst}}$ 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:

$$\mathrm{Slowdown} = \mathrm{Slowdown}_{\mathrm{Inst}} + \mathrm{Slowdown}_{\mathrm{D\text{-}Cache}} + \mathrm{Slowdown}_{\mathrm{I\text{-}Cache}} \qquad \text{EQ 4.}$$

Section 2.6.1 investigates Fast-Cache’s impact on the host data cache, and computes an estimate for its affect, $\mathrm{Slowdown}_{\mathrm{D\text{-}Cache}}$. Section 2.6.2 develops a model for Fast-Cache’s instruction cache behavior, and an estimate for $\mathrm{Slowdown}_{\mathrm{I\text{-}Cache}}$. 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 2.6.3 discusses the overall performance of Fast-Cache and Fast-Cache-Indirect.

2.6.1 Data Cache Effects

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

The lower bound, $\mathrm{Slowdown}_{\mathrm{D\text{-}Cache}}^{\mathrm{lower}}$, 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.

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

To be a true asymptotic bound, I 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, I assume C_{hostmiss} is the first-level cache miss penalty, or 5 cycles [76].

To validate this model, I use 4 programs from the SPEC92 benchmark suite [70]: `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 3.

To obtain a range of target miss ratios I varied the target cache size from 16 kilobytes to 1 megabyte, all direct-mapped with 32-byte blocks. I also simulated a 4-kilobyte cache for `fpppp` and `xlisp`, because of their low miss ratio on the other caches. I 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 10 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 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

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

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 [83]. To compute a single estimator of data cache performance, I calculate the mean of the upper and lower bounds:

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

As Figure 10 shows, this estimator—although simplistic—is quite accurate, predicting slowdowns within 30% of the measured values.

2.6.2 Instruction Cache Effects

The 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 [23]. 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

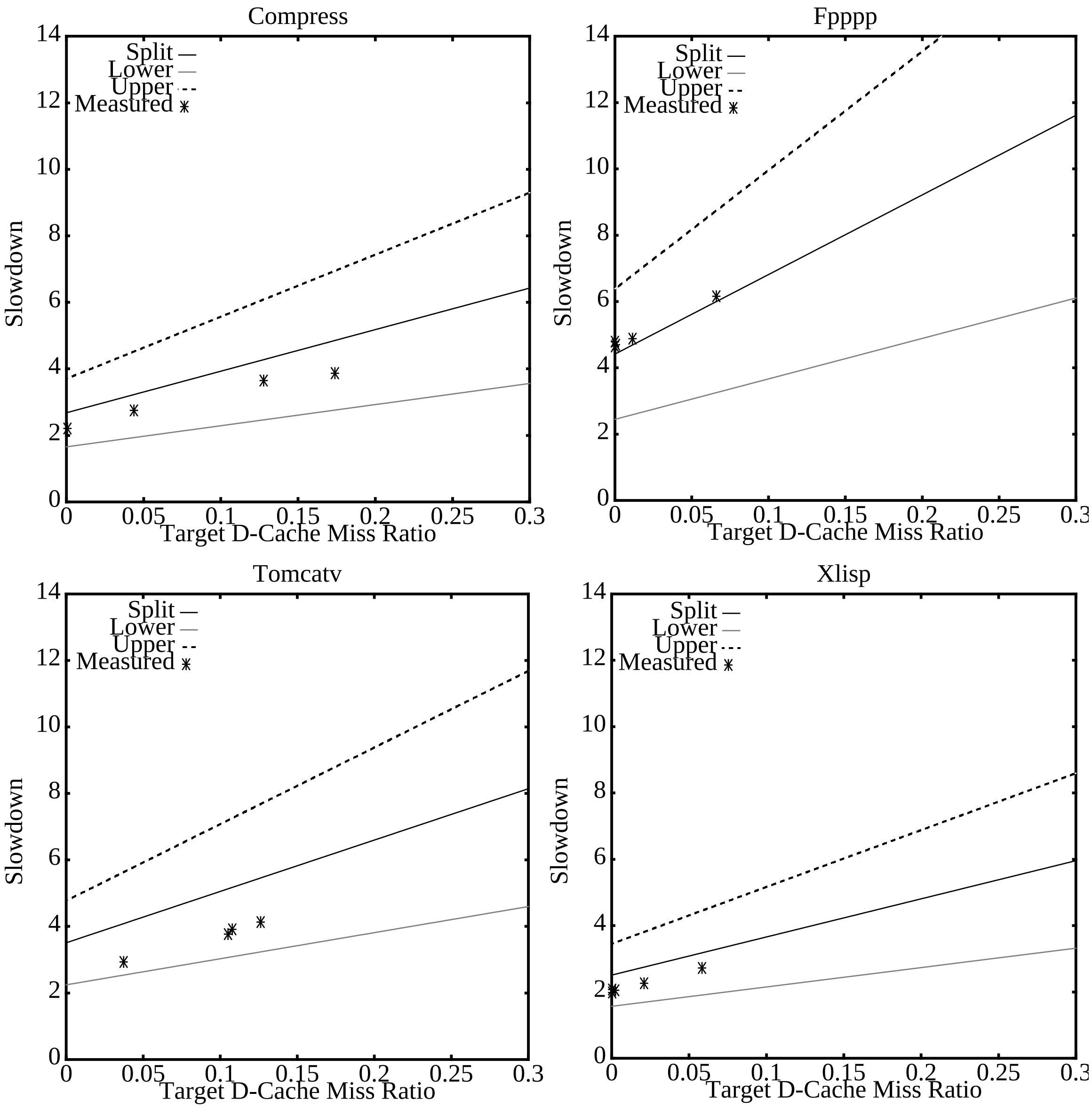


Figure 10: Data Cache Model

originally held 8 instructions; after instrumentation each holds an average of 2 original instructions. Intuitively, I 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 I 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. I call this the *scaled cache* model.

I can estimate the effect of a scaled cache using design target miss ratios [68] and other available data [23]. 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. [23] indicates that decreasing the instruction cache block size by *E* increases the number of instruction cache misses by *E*. Thus I expect that the number of instruction cache misses will be equal to $E\sqrt{E}$ times the original number of instruction cache misses. Since the original program incurs $I_{\mathrm{orig}} \cdot m_i$ misses, Fast-Cache incurs an additional slowdown of:

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

I 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 $I_{\mathrm{cc}} = 9$ instructions when it can use the condition codes and $I_{\mathrm{cc}} = 7$ instructions when it cannot, then the total code expansion is simply:

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

Since the total code expansion (see Table 4) is roughly a factor of 4, I 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.

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 [53]. To reduce instruction cache pollution, I 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 I 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 12.)

To validate the instruction cache models, I use *Shade* [13] to measure the instruction cache performance of the instrumented programs.[1] Because the code expansion is not

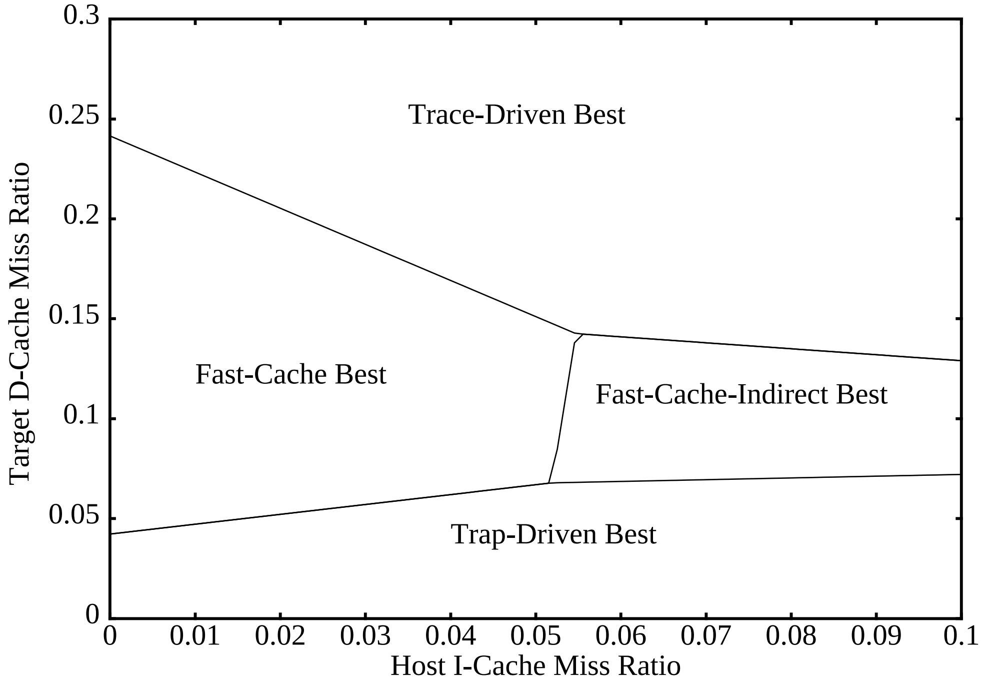
Benchmark	Original Misses (m_i)	Shade Misses Measured	Scaled Model (% of measured)		$E\sqrt{E}$ Model (% of measured)		Code Exp (E)
Fast-Cache							
Compress	329 (0.0%)	1,843	984	46%	1,848	0%	3.16
Fpppp	336,224 (3.7%)	4,629,793	4,361,246	6%	3,929,520	15%	5.15
Tomcatv	1,402 (0.0%)	27,143	33,680	24%	12,414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

exactly a power of two, I validate the scaled model by simulating caches of the next larger and smaller powers of two and interpolate. Table 4 shows 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.

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 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%).

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



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 11: Overall Simulator Performance

`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 I use the reference counter of the SuperSPARC second-level cache controller [76] 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.

2.6.3 Overall Performance

I now use the detailed model to revisit the comparison between Fast-Cache, trap-driven and trace-driven simulation. Figure 11 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 I 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 model, and 36% using $E\sqrt{E}$. For the programs I ran, instruction cache pollution has little effect on Fast-Cache simulation time (see Figure 12.) However, when simulating programs with larger instruction cache miss ratios, Fast-Cache-Indirect should be a better implementation.

2.7 Active Memory Applications and Extensions

2.7.1 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 [58]; the property of inclusion [52] indicates the number of references optimized is equal to the number of cache hits in a 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 13.

Many simulators that evaluate multiple cache configurations [30,74,52] use the property of inclusion [52] 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.

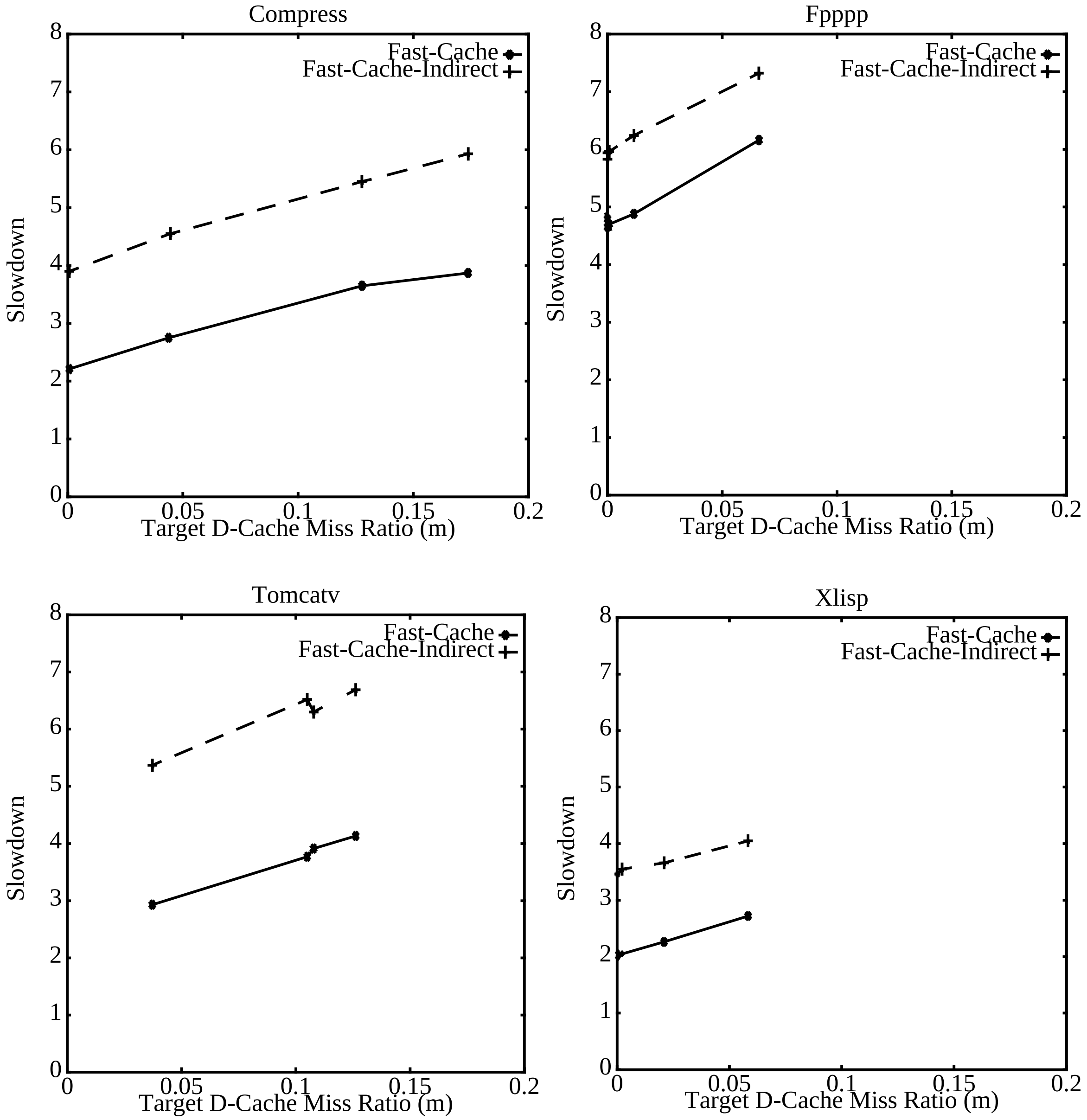


Figure 12: Fast-Cache-Indirect Performance

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 function specialized to update the appropriate caches. Simple simula-

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 13: Set-Associative Cache with LRU Replacement

tions 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 [61] and to provide low-cost portable fine-grain access control [63].

2.7.2 Extensions

A combination of table lookup and static analysis [81] 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 [35] or lockup-free caches [38], require accurate instruction cycle counts. Fast-Cache can easily add instruction cycle counts using techniques similar to QPT [43] or the Wisconsin Wind Tunnel [60]. 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 2.3, 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 completes before the application references the block, then when servicing a miss, the simulator 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 [38]. 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 [45] 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.

2.8 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 chapter 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. I 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 I 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 I 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. Chapter 3 discusses cache profiling—an important application that requires efficient cache simulation for its wide-spread use. 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.

Chapter 3

Cache Profiling

The active memory abstraction provides the mechanisms necessary to quickly execute simulations required by both hardware and software designers. Hardware designers rely on simulation to evaluate new memory systems. Software designers can use simulation to obtain information about their program's memory system behavior—called a cache profile. A cache profile allows programmers to restructure their code to improve spatial or temporal locality and improve overall performance.

The purpose of this chapter is to investigate cache profiling and tuning. I show that CProf—a cache profiling system—is an effective tool for improving program performance by focusing a programmer's attention on problematic code sections and providing *insight* into the type of program transformation to apply. Although many of the techniques explored in this chapter have been used sporadically in the supercomputer and multiprocessor communities, they also have broad applicability to programs running on fast uniprocessor workstations. Using CProf I obtained execution time speedups for the programs studied range from 1.02 to 3.46, depending on the machine's memory system.

3.1 Motivation

Cache memories help bridge the cycle-time gap between fast microprocessors and relatively slow main memories. By holding recently referenced regions of memory, caches can reduce the number of cycles the processor must stall waiting for data. As the disparity between processor and main memory cycle times increases—by 40% per year or more—cache performance becomes ever more critical.

Unfortunately, caches work well only for programs that exhibit sufficient locality. Other programs have reference patterns that caches cannot fully exploit, and spend much of their execution time transferring data between main memory and cache. For example, the SPEC92 [70] benchmark `tomcatv` spends 53% of its time waiting for memory on a DECstation 5000/125.

Fortunately, for many programs small changes in the source code can radically alter their memory reference pattern, greatly improving cache performance. Consider the well-known example of traversing a two-dimensional FORTRAN array. Since FORTRAN lays out arrays in column-major order, consecutive elements of a column are stored in consecutive memory locations. Traversing columns in the inner-loop (by incrementing the row index) produces a sequential reference pattern, and the spatial locality that most caches can exploit. If instead, the inner loop traverses rows, each inner-loop iteration references a different region of memory.

```
      DO 20 K = 1,100                     DO 20 K = 1,100
          DO 20 I = 1,5000                    DO 20 J = 1,100
              DO 20 J = 1,100                     DO 20 I = 1,5000
20                XA(I,J) = 2 * XA(I,J)   20          XA(I,J) = 2 * XA(I,J)
```

Row-Major Traversal **Column-Major Traversal**

By mentally applying the two different reference patterns to the underlying cache organization, I can predict the program’s cache performance. For example, for arrays that are much larger than the cache, the column-traversing version will have much better cache behavior than the row-traversing version. To verify my prediction, I executed the above code on a DECstation 5000/125 and the column-traversing version runs 1.69 times faster than the row-traversing version on an array of single-precision floating-point numbers.

I call the above analysis *mental* simulation of the cache behavior. This mental simulation is similar to asymptotic analysis of algorithms (e.g., worst-case behavior) that programmers commonly use to study the number of operations executed as a function of input size. When analyzing cache behavior, programmers perform a similar analysis, but must also have a basic understanding of cache operation (see Section 3.2).

Although asymptotic analysis is effective for certain algorithms, analyzing large complex programs is very difficult. Instead, programmers often rely on an execution-time profile to isolate problematic code sections, and then apply asymptotic analysis only on those sections. Unfortunately, traditional execution-time profiling tools, e.g., gprof [26] are generally insufficient to identify cache performance problems. For the FORTRAN array example above, an execution-time profile would identify the procedure or source lines as a bottleneck, but the programmer might erroneously conclude that the floating-point operations were responsible. Programmers would benefit from a profile that focuses specifically on a program’s cache behavior, identifying problematic code sections and data structures.

A cache profile can also provide insight into the cause of cache misses, which can help a programmer determine appropriate program transformations to improve performance.

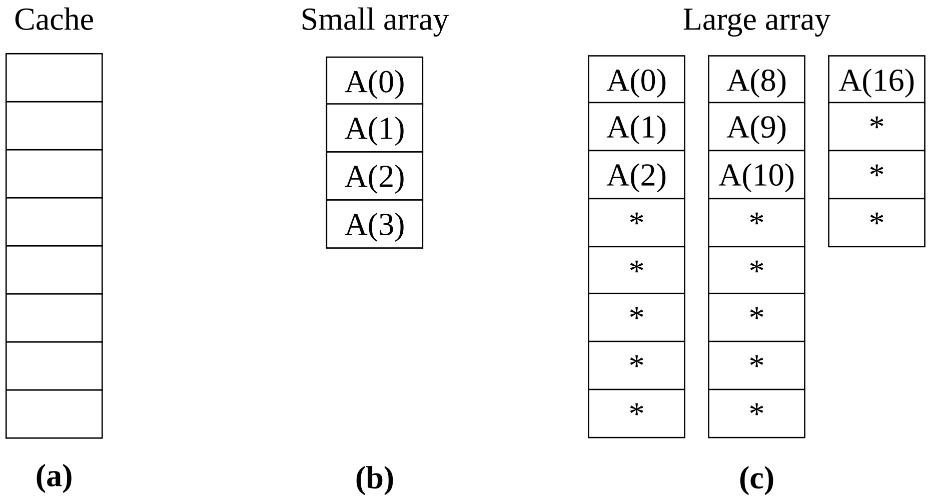
For cache profilers to become widely used, they must execute reasonably fast in addition to providing insight on how to remedy poor cache behavior. The active memory abstraction (described in Chapter 2) provides the framework necessary to quickly execute the simulations required to obtain a cache profile. A Fast-Cache implementation of CProf obtains detailed cache profiles of four programs with simulations executing only 3.4 to 15 times slower than the original *un-instrumented* program on a SPARCstation 10/51.

This chapter is organized as follows. Section 3.2 reviews how to reason about cache behavior and shows how knowing the cause of a cache miss helps provide insight into how to eliminate it. Section 3.3 presents a "cookbook" of simple program transformation techniques for improving program cache behavior, including array merging, padding and aligning structures, structure and array packing, loop interchange, loop fusion, and blocking. Section 3.4 briefly describes the CProf cache profiling system and its X-windows based user interface. Section 3.5 presents a case study where I used CProf to tune the cache performance of six programs from the SPEC92 benchmark suite: `compress`, `dnasa7`, `eqntott`, `spice`, `tomcatv`, and `xlisp`. I show how CProf identified the source lines and data structures that exhibit poor cache behavior, and how CProf helped provide the insight necessary to select the appropriate program transformation. Execution time speedups for these programs range from 1.02 to 3.46, depending on the machine's memory system. Others used CProf to tune the cache performance of relational database query processing, and improve execution time by 8% to 200% [64]. Section 3.6 concludes this chapter.

3.2 Understanding Cache Behavior: A Brief Review

To reason about a program's cache behavior, programmers must first recall the basic operation of cache memories. Caches are characterized by three major parameters: Capacity (**C**), Block Size (**B**), and Associativity (**A**). A cache's capacity (**C**) simply defines the total number of bytes it may contain. The block size (**B**) determines how many contiguous bytes are fetched on each cache miss. A cache may contain at most C/B blocks at any one time. Associativity (**A**) refers to the number of unique locations in the cache a particular block may reside in. If a block can reside in any location in the cache ($A=C/B$) the cache is called *fully-associative*; if a block can reside in exactly one location ($A=1$) it is called *direct-mapped*; if a block can reside in exactly A locations, it is called *A-way set-associative*. Smith's survey [67] provides a more detailed description of cache design.

With these three parameters, a programmer can analyze the first-order cache behavior for simple algorithms. Consider the simple example of nested loops where the outer-loop iterates L times and the inner-loop sequentially accesses an array of N 4-byte integers.



Sequentially accessing an array (b) that fits in cache (a) should produce M cache misses, where M is the number of cache blocks required to hold the array. Accessing an array that is much larger than the cache (c) should result ML cache misses, where L is the number of passes over the array.

Figure 14: Determining Expected Cache Behavior

```
for (i = 0; i < L; ++i)
    for (j = 0; j < N; ++j)
        a[j] += 2;
```

If the size of the array (4N) is smaller than the cache capacity (see Figure 14b), the expected number of cache misses is equal to the size of the array divided by the cache block size, 4N/B (i.e., the number of cache blocks required to hold the entire array). If the size of the array is larger than the cache capacity (see Figure 14c), the expected number of cache misses is approximately equal to the number of cache blocks required to contain the array times the number of outer loop iterations (4NL/B).

Someday compilers may automate this analysis and transform the code to reduce the miss frequency; recent research has produced promising results for restricted problem domains [57,41]. However, for general codes using current commercial compilers, the programmer must manually analyze the programs and perform transformations by hand.

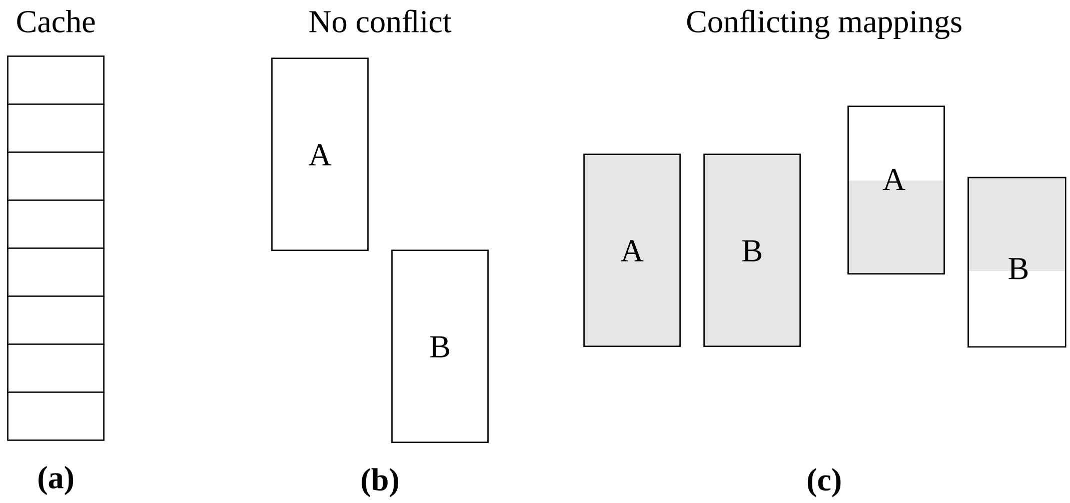
To select appropriate program transformations, a programmer must first understand the cause of poor cache behavior. One approach to understanding the cause of cache misses, is to classify each miss into one of three disjoint types: [30] *compulsory*, *capacity*, *conflict*.[1] A compulsory miss is caused by referencing a previously unreferenced cache block. In the small array example above (see Figure 14b), all misses are compulsory. Eliminating a compulsory miss requires prefetching the data, either by an explicit prefetch operation [57] or by placing more data items in a single cache block. For instance, changing the integers in my example to 2 bytes rather than 4, cuts the misses in half. However, since compulsory misses usually constitute a small fraction of all cache misses [30], I do not discuss them further.

A reference that is not a compulsory miss but misses in a fully-associative cache with LRU replacement is classified as a capacity miss. Capacity misses are caused by referencing more cache blocks than can fit in the cache. In the large array example above (see Figure 14c), I expect to see many capacity misses. Programmers can reduce capacity misses by restructuring the program to re-reference blocks while they are in cache. For example, it may be possible to modify the loop structure to perform the L outer-loop iterations on a portion of the array that fits in the cache and then move on to the next portion of the array. This technique, discussed further in the next section, is called *blocking* or *tiling*, and is similar to the techniques used to exploit the vector registers in some supercomputers.

A reference that hits in a fully-associative cache but misses in an A-way set-associative cache is classified as a conflict miss. A conflict miss to block X indicates that block X has been referenced in the recent past, since it is contained in the fully-associative cache, but at least A other cache blocks that map to the same cache set have been accessed since the last reference to block X. Consider the execution of a doubly-nested loop on a machine with a direct-mapped cache, where the inner loop sequentially accesses two arrays (e.g, a dot-product). If the combined size of the arrays is smaller than the cache, I might expect only compulsory misses. However, this ideal case only occurs if the two arrays map to different cache sets (Figure 15b). If they overlap, either partially or entirely (Figure 15c), they cause conflict misses as array elements compete for space in the set. Eliminating conflict misses requires a program transformation that changes either the memory allocation of the two arrays, so that contemporaneous accesses do not compete for the same sets, or that changes the manner in which the arrays are accessed. As discussed in the next section, one solution is to change the memory allocation by merging the two arrays into an array of structures.

The discussion thus far assumes a cache indexed using virtual addresses. Many systems index their caches with real or physical addresses, which makes cache behavior strongly dependent on page placement. However, many operating systems use page coloring to minimize this effect, thus reducing the performance difference between virtual-indexed and real-indexed caches [37].

[1] Hill defines compulsory, capacity, and conflict misses in terms of miss ratios. When generalizing this concept to individual cache misses, I must introduce *anti-conflict* misses which miss in a fully-associative cache with LRU replacement but hit in an A-way set-associative cache. Anti-conflict misses are generally only useful for understanding the rare cases when a set-associative cache performs better than a fully-associative cache of the same capacity.



The presence of conflict misses indicates a mapping problem. Figure 15b shows how two arrays that fit in cache with a mapping that will not produce any conflict misses, whereas Figure 15c shows two mappings that will result in conflict misses.

Figure 15: Conflicting Cache Mappings

3.3 Techniques for Improving Cache Behavior

The analysis techniques described in the previous section can help a programmer understand the cause of cache misses. One of the primary contributions of this chapter is the mapping of cache miss types to program transformations that reduce cache misses. In this section, I present a *cookbook* of simple program transformations that can help eliminate some of the misses.

Program transformations can be classified by the type of cache misses they eliminate. Conflict misses can be reduced by array merging, padding and aligning structures, structure and array packing, and loop interchange [57]. The first three techniques change the allocation of data structures, whereas loop interchange modifies the order that data structures are referenced. Capacity misses can be eliminated by program transformations that reuse data before it is displaced from the cache, such as loop fusion [57], blocking [41, 57], structure and array packing, and loop interchange. In the following sections, I present examples of each of these techniques, except loop interchange, which was discussed in Section 3.1.

Merging arrays. Some programs contemporaneously reference two (or more) arrays of the same dimension using the same indices. By merging multiple arrays into a single compound array, the programmer increases spatial locality and potentially reduces conflict misses. In the C programming language, this can be accomplished by declaring an array of

```
/* old declaration of two arrays */
int val[SIZE];
int key[SIZE];

/* new declaration of */
/* array of structures */
struct merge {
    int val; int key;
};

struct merge merged_array[SIZE];
```

(a)

```
C old declaration
      integer X(N,N)
      integer Y(N,N)

C new declaration
      integer XY(2*N,N)

C preprocessor macro
C definitions to perform addressing
#define X(i,j) XY((2*i)-1,N)
#define Y(i,j) XY((2*i),N)
```

(b)

Figure 16: Merging Arrays in C (a) and FORTRAN77 (b)

structures rather than two arrays (Figure 16a). This simple transformation can also be performed in FORTRAN90, which provides structures. Since FORTRAN77 does not have structures, the programmer can obtain the same effect using complex indexing (Figure 16b).

Padding and Aligning Structures. Referencing a data structure that spans two cache blocks may incur two misses, even if the structure itself is smaller than the block size. Padding structures to a multiple of the block size and aligning them on a block boundary can eliminate these “misalignment” misses, which generally show up as conflict misses. Padding is easily accomplished in C (Figure 17a) by declaring extra pad fields. Note that padding may introduce additional capacity misses. Alignment is a little more difficult, since the address of the structure must be a multiple of the cache block size. Statically-declared structures generally require compiler support. Dynamically allocated structures can be aligned by the programmer using simple pointer arithmetic (Figure 17b). Note that some dynamic memory allocators (e.g., some versions of *malloc()*) return cache-block aligned memory.

Packing. Packing is the opposite of padding; by packing an array into the smallest space possible, the programmer increases spatial locality, which can reduce both conflict and capacity misses. In the example in Figure 18, the programmer observes that the elements of array *value* are never greater than 255, and hence could fit in type *unsigned char*, which requires 8-bits, instead of *unsigned int*, which typically requires 32-bits. For a machine with 16-byte cache blocks, the code in Figure 18b permits 16 elements per block, rather than 4 and reduces the maximum number of cache misses by a factor of 4. Packing may introduce additional instructions to unpack the data.

```
/* old declaration of a twelve */
/* byte structure */
struct ex_struct {
    int val1,val2,val3;
};


/* new declaration of structure */
/* padded to 16-byte block size */
struct ex_struct {

    int val1,val2,val3;
    char pad[4];
};
```

(a)

```
/* original allocation does not */
/* guarantee alignment */
ar = (struct ex_struct *)
    malloc(sizeof(struct ex_struct)
                    * SIZE);

/* new code to guarantee alignment */
/* of structure. */
r = (struct ex_struct *)
    malloc(sizeof(struct ex_struct)
                    * (SIZE+1));

ar = ((int) ar + B-1)/B)*B
    /* B is the cache block size */
```

(b)

Figure 17: Padding (a) and aligning (b) structures in C.

```
/* old declaration of an array */
/* of unsigned integers. */
unsigned int values[10000];

/* loop sequencing through values */
for (i=0; i<10000; i++)
    values[i] = i % 256;
```

(a)

```
/* new declaration of an array */
/* of unsigned characters. */
/* Valid iff 0 <= value <= 255 */
unsigned char values[10000];

/* loop sequencing through values */
for (i=0; i<10000; i++)
    values[i] = i % 256;
```

(b)

Figure 18: Unpacked (a) and packed (b) array structures in C.

Loop Fusion. Numeric programs often consist of several operations on the same data, coded as multiple loops over the same arrays. By combining these loops, a programmer increases the program’s temporal locality and frequently reduces the number of capacity misses. The examples in Figure 19 combine two doubly-nested loops so that all operations are performed on an entire row before moving on to the next.

Loop fusion is exactly the opposite of *loop fission*, a program transformation that splits independent portions of a loop body into separate loops. Loop fission helps an optimizing compiler detect loops that exploit vector hardware on some supercomputers, or separates

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        a[i][j] = 1/b[i][j]*c[i][j];

for (i=0; i < N; i++)
    for (j=0; j < N; j++)
        d[i][j] = a[i][j]+c[i][j];
```

(a)

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
        a[i][j] = 1/b[i][j]*c[i][j];
        d[i][j] = a[i][j]+c[i][j];
    }
```

(b)

Figure 19: Separate (a) and fused (b) loops.

```
      DO 110 J = 1, M
          DO 110 K = 1, N
              DO 110 I = 1, L
              C(I,K) = C(I,K) +
              A(I,J) * B(J,K)
110   CONTINUE
```

(a)

```
DO 110 J = 1, M, 4
    DO 110 K = 1, N
        DO 110 I = 1, L
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
                    + A(I,J+1) * B(J+1,K)
                    + A(I,J+2) *B(J+2,K)
                    + A(I,J+3) * B(J+3,K)

110 CONTINUE
```

(b)

Figure 20: Naive (a) and SPEC column-blocked (b) matrix multiply.

Blocking. Blocking is a general technique for restructuring a program to reuse chunks of data that fit in the cache, and hence reduce capacity misses. The SPEC matrix multiply (part of `dnasa7`, a FORTRAN77 program) implements a column-blocked algorithm (Figure 20b) that achieves a 2.04 speedup over a naive implementation (Figure 20a) on a DECstation 5000/125. The algorithm tries to keep 4 columns of the A matrix in cache for the duration of the outermost loop, ideally getting N-1 hits for each miss. If the matrix is so large that 4 columns do not fit in the cache, one could use a two-dimensional (row and column) blocked algorithm instead.

3.4 CProf: A Cache Profiling System

The analysis and transformation techniques, described in the previous section, can help a programmer develop algorithms that minimize cache misses. However, cache misses result from the complex interaction between algorithm, memory allocation, and cache configuration; when the program is executed, the programmer’s expectations may not match reality. I have developed a cache profiling system, CProf, that addresses this prob-

Program	Time (Sec)	Slowdown
compress	44.5	15.0
fpppp	989.9	5.2
tomcatv	915.8	9.2
xlisp	911.2	3.4

Table 5: Slowdowns for an Active Memory Implementation of CProf

lem by mapping cache misses to source lines and data structures and classifying the misses as compulsory, capacity, and conflict. This provides the insight necessary for programmers to select program transformations that improve cache behavior.

Cache and memory system profilers differ from the better-known execution-time profilers by focussing specifically on memory system performance. Memory system profilers do not obviate execution-time profilers; instead, they provide supplementary information necessary to quickly identify memory system bottlenecks and tune memory system performance.

There are a number of cache and memory system profilers that differ in the level of detail they present to a programmer. High-level tools, such as MTOOL [25], identify procedures or basic blocks that incur large memory overheads. Other cache profilers, such as PFC-Sim [57] and CProf, identify cache misses at the source line level, allowing much more detailed analysis. Of course this extra detail does not come for free; MTOOL runs much faster than profilers requiring address tracing and full cache simulation. However, full simulation permits a profiler to identify which data structures are responsible for cache misses and to determine the type of miss, features provided by both Memspy [50] and CProf. Furthermore, the active memory abstraction can be used to significantly reduce simulation times. Table 5 shows that execution times for a CProf simulator implemented using Fast-Cache, are between 3.5 and 15 times slower than the original program. However, the results presented in this chapter were obtained using an implementation of CProf that processes traces generated by QPT [43].

Memspy is very similar to CProf, the difference being the granularity at which source code is annotated and the miss type classification. Memspy annotates source code at the procedure level and provides only two miss types for uniprocessors: compulsory and replacement. Determining if a replacement miss is a result of referencing more data than will fit into the cache—a capacity miss—or a mapping problem—a conflict miss—is left to the user. Memspy provides some insight into the cause of replacement misses by identifying the data structures competing for space in the cache. CProf is an artifact of my contributions to cache profiling: providing fine-grain source identification, data structure support, and classifying cache misses as compulsory, capacity, or conflict, thus providing insight for fixing memory system bottlenecks.

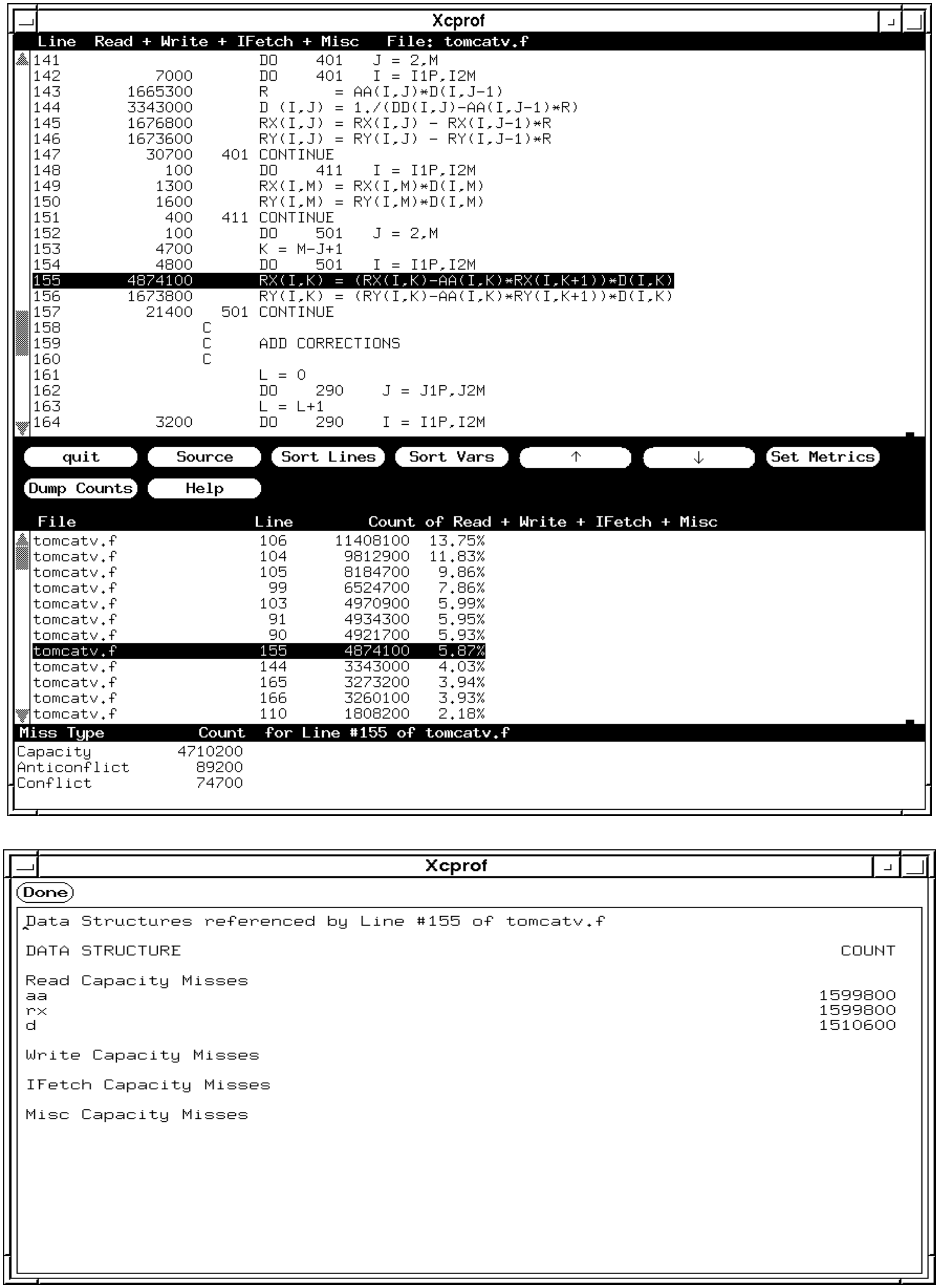
3.4.1 CProf User Interface

CProf uses a flexible X-windows interface (see Figure 21) to present the cache profile in a way that helps the programmer determine the cache performance bottlenecks. The user can list either source lines or data structures, sorted in descending order of importance, allowing quick identification of poor cache behavior. Misses are cross-referenced, so a programmer can quickly determine which of several data structures on a source line is responsible for most cache misses.

CProf's user interface is divided into three sections for data presentation and one section for command buttons. The top section is the text window, the middle section is the data window, and the bottom section is the detail window. A particular window's use depends on the selected command button.

The *source* button opens a pull-down menu with an entry for each source file and an additional entry that allows a display of a list of source files sorted by the number of cache misses. Selecting one of the files displays the source code in the text window. Each source line is labeled with the number of cache misses generated by that line. I highlight the line with the most cache misses. The up-arrow and down-arrow buttons allow movement within the source file to the line with the next higher or next lower number of misses, respectively. The detail window refines the cache misses for the highlighted line into the miss type. Selecting a miss type opens a window that displays the data structures referenced by this source line and the corresponding number of cache misses for the miss type selected (Figure 21). The *sort lines* button displays a list of source lines in the data window, sorted according to the number of cache misses. Each entry contains the file name, the line number, the number of cache misses, and the percent of the total misses. A sorted list of data structures is displayed by the *sort vars* button. Each entry in this list contains the variable name, the count of the number of misses and the percentage of total misses. Selecting a miss type causes a window to open that displays the source lines that reference this data structure and the corresponding number of cache misses for the miss type selected. The user selects which reference types (*READ*, *WRITE*, *IFETCH*) to display with the *set metrics* button. Finally, the counts displayed in the data window can be written to a file with the *dump counts* button.

CProf annotates both static and dynamic data structures. Dynamically allocated structures are labeled by concatenating the procedure names on the call stack at the point of allocation [84]. An end counter value allows unique identification of all dynamically allocated structures.



```
 Line  Read + Write + IFetch + Misc    File: tomcatv.f
141                         DO    401   J = 2,M
142         7000            DO    401   I = I1P,I2M
143      1665300            R        = AA(I,J)*D(I,J-1)
144      3343000            D (I,J) = 1./(DD(I,J)-AA(I,J-1)*R)
145      1676800            RX(I,J) = RX(I,J) - RX(I,J-1)*R
146      1673600            RY(I,J) = RY(I,J) - RY(I,J-1)*R
147        30700        401 CONTINUE
148          100            DO    411   I = I1P,I2M
149         1300            RX(I,M) = RX(I,M)*D(I,M)
150         1600            RY(I,M) = RY(I,M)*D(I,M)
151          400        411 CONTINUE
152          100            DO    501   J = 2,M
153         4700            K = M-J+1
154         4800            DO    501   I = I1P,I2M
155      4874100            RX(I,K) = (RX(I,K)-AA(I,K)*RX(I,K+1))*D(I,K)
156      1673800            RY(I,K) = (RY(I,K)-AA(I,K)*RY(I,K+1))*D(I,K)
157        21400        501 CONTINUE
158                   C
159                   C     ADD CORRECTIONS
160                   C
161                         L = 0
162                         DO    290    J = J1P,J2M
163                         L = L+1
164         3200            DO    290    I = I1P,I2M
```

```
 File                     Line        Count  of Read + Write + IFetch + Misc
tomcatv.f                  106     11408100  13.75%
tomcatv.f                  104      9812900  11.83%
tomcatv.f                  105      8184700   9.86%
tomcatv.f                   99      6524700   7.86%
tomcatv.f                  103      4970900   5.99%
tomcatv.f                   91      4934300   5.95%
tomcatv.f                   90      4921700   5.93%
tomcatv.f                  155      4874100   5.87%
tomcatv.f                  144      3343000   4.03%
tomcatv.f                  165      3273200   3.94%
tomcatv.f                  166      3260100   3.93%
tomcatv.f                  110      1808200   2.18%
```

```
Miss Type              Count   for Line #155 of tomcatv.f
Capacity             4710200
Anticonflict           89200
Conflict               74700
```

```
Data Structures referenced by Line #155 of tomcatv.f

DATA STRUCTURE                                                          COUNT

Read Capacity Misses
aa                                                                    1599800
rx                                                                    1599800
d                                                                     1510600

Write Capacity Misses

IFetch Capacity Misses

Misc Capacity Misses
```

Figure 21: CProf User Interface

The *text window* is used to view individual source files, where each line is annotated with the corresponding number of cache misses. The X-windows user interface allows the user to browse within the source file, moving to the line with the next higher or next lower number of cache misses. The *detail window* displays the number of each miss type for the currently selected source line or data structure.

CProf is very effective at identifying where a program exhibits poor cache behavior, and the cache miss types help a programmer select what type of program transformation to apply. In the next section, I describe more about how to use CProf to tune program performance.

3.5 Case Study: The SPEC Benchmarks

In this section, I describe a study where I use CProf and the cookbook of transformations to tune the cache performance of six programs from the SPEC92 benchmark suite: `compress`, `dnasa7`, `eqntott`, `spice`, `tomcatv`, and `xlisp`. The purpose of this section is two-fold. First, I show that I can obtain significant speedups using cache profiling, even for codes that have been extensively tuned using execution-time profilers. Second, I show how to use CProf to gain insight into the cache behavior, and determine which transformations were likely to improve performance.

I present performance results in terms of speedup in user execution time[1] on three models of the DECstation 5000, the 5000/240, 5000/125, and 5000/200. Each of these machines have separate 64-kilobyte direct-mapped instruction and data caches, 16-byte blocks, and a write buffer. The 5000/125 and 5000/200 use a 25 MHz MIPS R3000 processor chip. The major difference between the memory systems of these two machines is the cache miss penalty—16 processor cycles on the DECstation 5000/200 and 34 cycles on the DECstation 5000/125—which helps illustrate the importance of cache profiling as cache miss penalty increases. The 5000/240 uses a 40 MHZ MIPS R3000 processor chip and has a 28 cycle miss penalty.

These machines also have secondary differences with significant performance impact. For example, the 5000/2xx have 4-deep write buffers, while the 5000/125 has only a 2-deep write buffer. In addition, the 5000/240 performs sequential prefetch on cache misses, reducing the effective miss penalty for long sequential accesses. While these secondary factors can significantly affect execution time, I have not found it necessary to model these factors in CProf's cache simulation.

[1] System time accounts for very little of the total execution time for most of the programs. `compress` is the exception where system time is relatively high because of the large amount of I/O. In this case excluding the system time eliminates the bias introduced by the different I/O systems.

Program	Restructuring Technique					
	Merging Arrays	Loop Fusion	Loop Interchange	Pad and Align	Packing	Blocking
btrix†	•	•	•			
cholesky†			•			
compress	•					•
eqntott					•	
gmtry†			•			
mxm†						•
spice	•					
tomcatv	•	•				
vpenta†	•	•	•			
xlisp				•		

† dnasa7

Table 6: Restructuring Techniques for Improved Cache Performance

To reduce experimental error, I averaged the execution time over five runs. The programs were compiled at optimization level -O3 using the MIPS Version 2.1 C and F77 compilers. spice was the one exception, which I compiled at optimization level -O2, per the SPEC make file. Note that while run-times are all reported with full optimization, I profiled most of the programs at optimization level -O1, with full symbolic debugging (-g). Cache profiling at high optimization levels suffers from the same difficulties as debugging (i.e., incorrect line numbers), since CProf uses the same symbol table information.

Table 6 shows the applications that benefited from the various restructuring techniques. The benchmark dnasa7 consists of seven numerical kernels; I broke out five kernels with poor cache performance and analyzed them separately. Both the original and tuned times for dnasa7 include the SPEC version of matrix multiply (mxm).

Table 7 and Figure 22 present execution time results for the six benchmarks. The full programs execute as much as 90% faster when modified to improve cache behavior. Breaking out the kernels in dnasa7 shows even more striking results, with speedups as much as 3.46 for vpenta on the DECstation 5000/240, 2.53 on the DECstation 5000/125, and 2.14 on the DECstation 5000/200.

Program	DEC5000/125		DEC5000/200		DEC5000/240		Modification
	Sec.	S	Sec.	S	Sec.	S	
compress	7.70		5.98		5.56		original
	7.34	1.05	5.84	1.02	5.22	1.07	array merge
	4.94	1.56	4.60	1.30	2.90	1.92	reduced hash table
dnasa7	1228.22		904.60		796.60		original
	945.18	1.30	727.84	1.24	527.24	1.51	tuned kernels
btrix	144.06		114.50		82.52		original
	109.50	1.32	89.92	1.27	55.94	1.48	loop interchange &
cholesky	188.90		141.14		97.14		original
	162.16	1.16	124.94	1.13	73.66	1.32	loop interchange
gmtry	177.06		141.98		128.42		original
	119.78	1.48	95.82	1.48	50.92	2.52	loop interchange
mxm	248.44		184.56		91.36		naive
	122.06	2.04	106.02	1.74	66.08	1.38	SPEC blocked
vpenta	264.78		169.86		203.80		original
	126.38	2.10	91.80	1.85	69.60	2.93	array merge
	104.54	2.53	79.42	2.14	58.88	3.46	+loop fusion
eqntott	67.56		58.70		39.96		original
	60.98	1.11	55.40	1.06	38.92	1.03	changed short to char
spice	2242.10		1762.34		1557.90		original
	1781.72	1.26	1406.04	1.25	1163.42	1.34	array merge
tomcatv	221.20		161.20		137.30		original
	167.24	1.32	134.38	1.20	91.40	1.50	merged arrays X & Y
	150.88	1.47	126.36	1.28	86.08	1.60	+loop fusion
xlisp	385.24		286.56		205.72		original
	361.96	1.06	277.18	1.03	190.30	1.08	pad node to 16 bytes

Table 7: Execution Time Speedups (S)

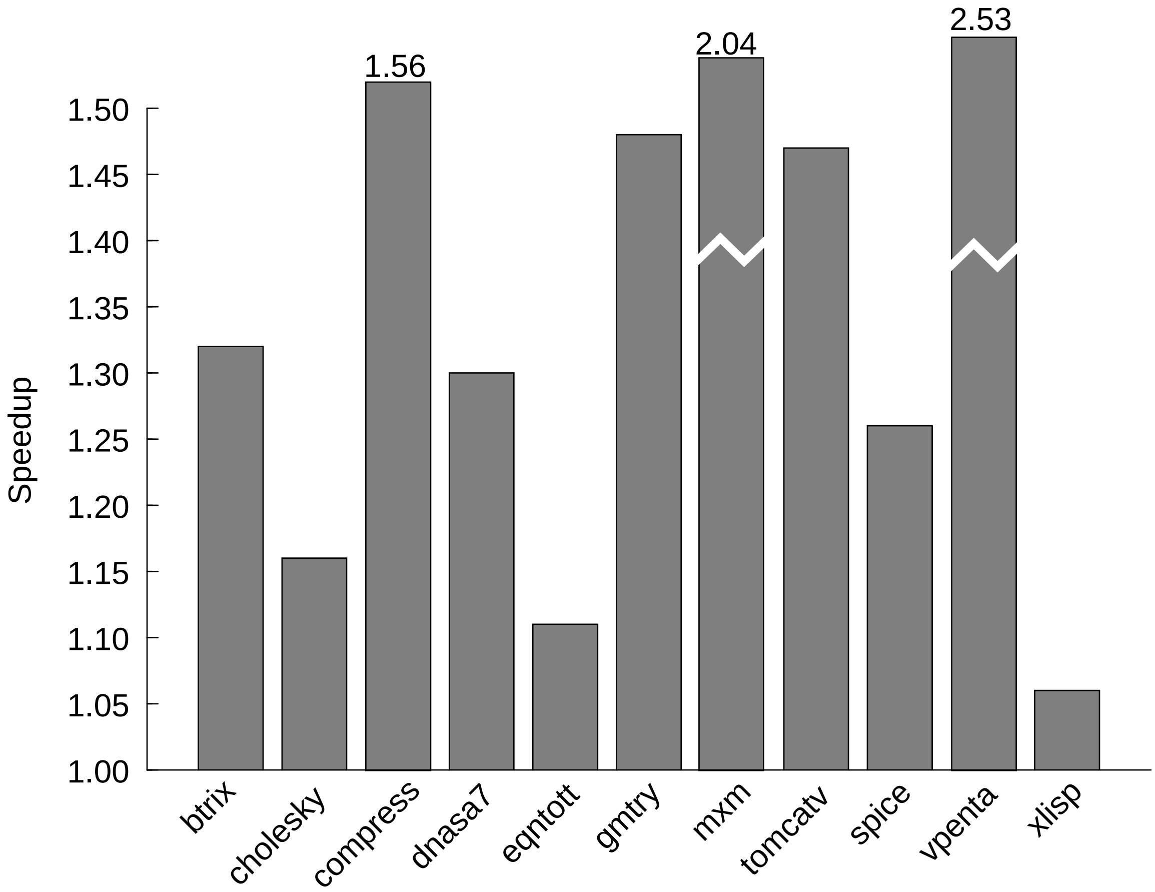


Figure 22: Speedups on a DECstation 5000/125

The remainder of this section discusses my experience cache profiling and modifying each program. I provide a very brief description of the program, followed by the results of the initial CProf execution. I then discuss the modifications I performed and the resulting speedups.

3.5.1 compress

`compress` is a UNIX utility that implements the well-known Lempel-Ziv data compression algorithm. For each input character, `compress` searches a hash table for a prefix key. When the key matches, another array is accessed to obtain the appropriate value. The hash table is quite large (69001 entries), to reduce the probability of collisions. When a collision does occur, a secondary probe is initiated.

CProf indicates two source lines that reference the data structure that stores the keys are responsible for 71% of the cache misses. One source line is the initial probe into the hash table, which accounts for 21% of the cache misses. The other source line performs the secondary probe operation when there is a collision and accounts for 50% of the misses. CProf also shows that most of the misses are capacity misses.

Recall that I can eliminate capacity misses by processing data in portions that fit in the cache. Applying this insight to `compress`, I reduced the hash table size from 69001 to 5003, which is small enough to fit in the data cache.[1] This change results in speedups of 1.92 on a DECstation 5000/240, 1.56 on a 5000/125, and 1.30 on a 5000/200. However, this modification actually changes the program output, since the compression ratio (original file size / compressed file size) is related to the size of the hash table. The output is still a compatible compressed file, but it does not match the standard SPEC output. Nonetheless, there is a clear trade-off between speed and compression ratio. The un-optimized version has a compression ratio of 2.13, whereas the optimized version has 1.77.

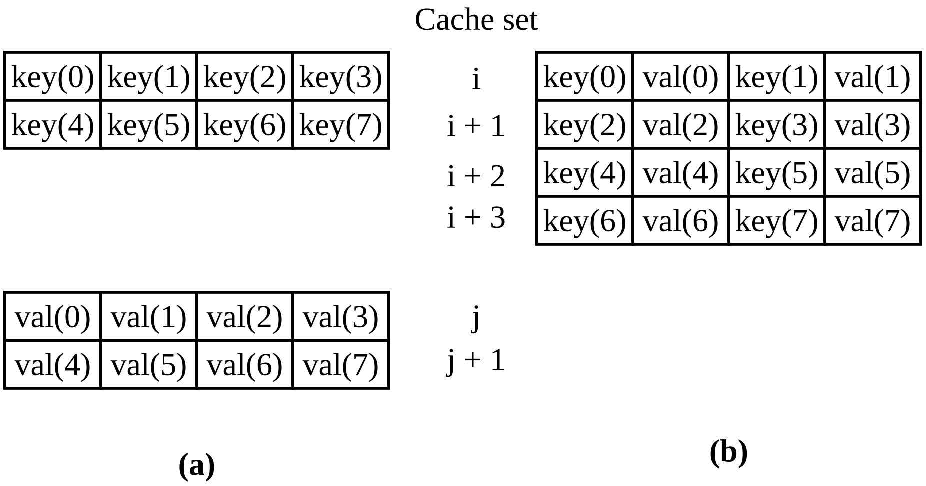
I also tried to improve the cache performance of compress without changing the compression ratio. Although compress has a large number of capacity misses, conflict misses account for 13% of the misses to the key array and 19% of the misses to the value array; the array index is the same for both of these arrays. Although separate arrays reduce the total space requirements (the key is a C `integer` and the value is a `short`; alignment restrictions in C require padding if these are combined into an array of structures), the price is poor spatial locality. After referencing a key, `compress` is likely to reference the corresponding value, which resides in the other array and hence a different cache block (see Figure 23a).

Merging the two arrays into a single array of structures places the key and value in the same cache block (see Figure 23b) and improves spatial locality. With this modification, accesses to the value always hit in the cache, assuming proper alignment, reducing the number of conflict misses and providing speedups of 1.07 on the DECstation 5000/240, 1.05 on the 5000/125 and 1.02 on the 5000/200.

3.5.2 eqntott

The SPEC benchmark `eqntott` is a CAD tool that converts boolean equations into their equivalent truth tables. Execution-time profiling shows that `eqntott` spends 95% of its time in the quick-sort routine [56]. CProf further reveals that most of this time is spent moving the sort keys from memory into the cache; over 90% of the misses are generated in one comparison routine. The offending routine examines two arrays and generates mostly capacity misses, indicating that the program either needs to re-reference blocks while they are in the cache, or bring in fewer blocks. CProf indicates that most of these capacity misses are due to fetching `BIT` structures dynamically allocated at line #44 in `pterm.c`. The `BIT` data type is a 16-bit integer (type `short` in C), and inspection of

[1] This transformation was suggested by James Larus.


Cache set
key(0) key(1) key(2) key(3)
key(4) key(5) key(6) key(7)
val(0) val(1) val(2) val(3)
val(4) val(5) val(6) val(7)
i
i + 1
i + 2
i + 3
j
j + 1
key(0) val(0) key(1) val(1)
key(2) val(2) key(3) val(3)
key(4) val(4) key(5) val(5)
key(6) val(6) key(7) val(7)
(a)
(b)

The initial allocation strategy for the key and value arrays (a) resulted in as many as two cache misses for each successful hash table probe. Merging the two arrays into an array of structures (b) effectively interleaves the elements of the two arrays and results in only one cache miss per successful probe.

Figure 23: Cache mappings for compress.

the source code shows that `BIT` data types only take on values in the set [0,1,2]. Changing the type definition from 16-bit integer to 8-bit integer (`short` to `char`) reduces the number of misses in this routine by half. The speedup in execution time is 1.03 on a 5000/240, 1.11 on a 5000/125 and 1.06 on a 5000/200. The prefetch capabilities of the 5000/240 exploit the sequential accesses of the compare routine, reducing the benefit of my modification.

In `eqntott`, the integer values actually represent the symbolic values ZERO, ONE, and DASH. Although I did not implement this alternative, with the use of enumerated types, a compiler could potentially allocate as few as two bits per array element, resulting in one-eighth the number of cache misses. However, the trade-off between fewer cache misses and the time to unpack the data, is implementation dependent.

3.5.3 xlisp

The SPEC benchmark `xlisp` is a small lisp interpreter solving the nine queens problem. To reduce computation requirements during profiling, I profiled `xlisp` solving the six queens problem; however, the speedup results in Table 7 and Figure 22 are for the standard nine queens problem. Programmers should be aware that cache behavior is sensitive to the input data; programs may exhibit good cache behavior with smaller input sizes, and poor behavior for larger inputs. In this case the results obtained from the smaller input data were sufficient to achieve reasonable speedups with the larger input.

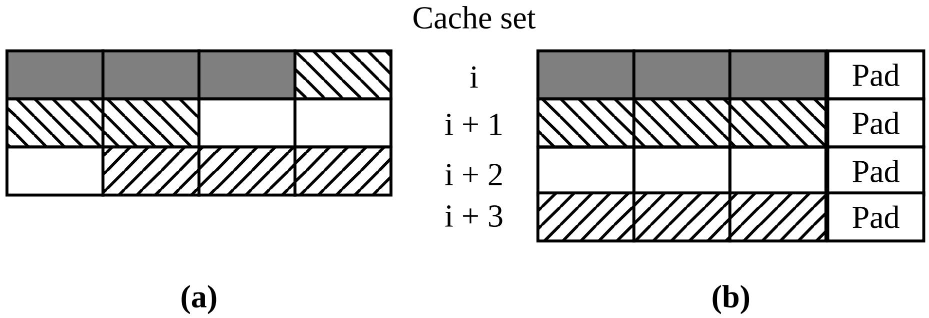
CProf shows that approximately 40% of the cache misses occur during the mark and sweep garbage collection, most of which are conflict misses. During this phase, the program first traverses the reachable nodes and marks them accessible, then sweeps sequentially through the memory segment placing unmarked nodes on the free list. Mark and sweep garbage collection has inherently poor locality, and an alternate algorithm would provide better cache behavior. However, such an extensive modification was outside the scope of this study.

CProf shows that 19% of the cache misses are generated by the single source line that checks the flag (used to mark accessibility) during the sweep. Since conflict misses dominate, I first improved the spatial locality of the sweep routine by separating the flags from the rest of the node structure. By placing the flags in a single array, the sequential sweep exhibited excellent spatial locality: for every miss, the next 15 references hit—eliminating most of the cache misses in the sweep routine. Unfortunately, the change also increased the number of misses in the mark routine which must first fetch a node, then the corresponding flag. This modification increased spatial locality in the sweep at the expense of spatial locality during the mark, resulting in a negligible change in performance.

Returning to CProf, I see that the node structures allocated on line #540 of `xldmem.c` incur a large number of conflict misses. Inspection of the source reveals each node structure occupies 12 bytes, or three-fourths of a 16-byte cache block. Consequently, only half of the nodes reside entirely within a single cache block (see Figure 24). The remaining half of the nodes reside in two contiguous cache blocks, potentially causing two cache misses—when referenced—rather than one. By explicitly padding the original node structure to 16 bytes, the cache block size, and ensuring alignment on cache block boundaries, I obtained a 1.08 speedup on the DECstation 5000/240, 1.06 on the 5000/125 and 1.03 on the 5000/200.

It is important to realize that padding data structures without guaranteeing alignment can be worse than not padding them at all. In this example, I might end up with *all* nodes generating two misses (if not in cache), rather than only half. Similarly, while many memory allocators (e.g., the ULTRIX `malloc()` routine) return cache-block-aligned memory, `xlisp` pre-allocates large chunks and manages them itself, bypassing the alignment performed within the allocator. Application-specific memory managers certainly have a role, but programmers should remember the impact of padding and alignment on cache performance.

Padding data structures also wastes memory space: the xlisp node structures use only 10 bytes of information. Explicit padding increases the allocated size from the 12 bytes required by C language semantics to 16 bytes, a 33% increase in storage. This increase



Each pattern corresponds to a different node structure, while *pad* indicates wasted storage. The initial allocation strategy (a) resulted in two cache misses for half of the nodes not in the cache. Padding the structures to equal a cache block size and alignment on cache block boundaries (b) reduces this to only one cache miss per node not resident in the cache.

Figure 24: Cache Mappings for Xlisp Node Structures

could adversely affect virtual memory performance for larger programs, although it was not an issue for the nine queens problem [56].

3.5.4 tomcatv

`Tomcatv` is a FORTRAN 77 mesh generation program that uses seven two-dimensional data arrays, each of which requires approximately 0.5 M-Byte. The algorithm (see Figure 19) consists of a forward pass in which two arrays are read and the other five written (loops 1,2,3), a backward pass (loop 4) over two arrays to calculate errors, and finally another forward pass (loop 5) to add in these errors.

Since the arrays are much larger than the cache, and the arrays are sequentially accessed I expect to see a large number of capacity misses. However, CProf shows that read accesses to arrays X and Y during the first loop of the initial forward pass, are generating a large number of conflict misses. It is easily observed from the source code that the two arrays are always referenced with the same indices. Hence, to improve spatial locality, I merged them together, placing elements X(I,J) and Y(I,J) in the same cache block. This modification results in speedups of 1.50 on the DECstation 5000/240, 1.32 on the 5000/125 and 1.20 on the 5000/200.

Running CProf on the modified `tomcatv` finds that capacity misses to the RX and RY arrays now dominate. As Figure 25 shows, the forward pass is actually composed of several loops: loop 1 initially references six arrays, including writing RX and RY, followed by loop 2 which computes the maximum values of the RX and RY arrays, and a final pass (loop 3) over the RX and RY arrays to adjust the values. In addition to these disjoint for-

```
         for LL

             /* forward wave */
loop 1.      for j
                 for i
                     X, Y, RX, RY, AA, DD

loop 2.      for j
                 for i
                     RX, RY
loop 3.      for j
                 for i
                     AA, DD, RX, RY, D

             /* backward wave */
loop 4.      for j
                 for i
                     RX, RY, AA, DD


             /* forward wave */
loop 5.      for j
                 for i
                 X, Y, RX, RY
         endfor /* LL */
```

(a)

```
         for LL

             /* forward wave */
loop 1.      for j
                     if (j > 1)
                         for i
                             X, Y, RX, RY
                     for i
                         X, Y, RX, RY, AA, DD
                     for i
                         RX, RY
                     for i
                         AA, DD, RX, RY, D

             /* backward wave */
loop 2.      for j
                 for i
                     RX, RY, AA, D
         endfor /* LL */
```

(b)

The original `tomcatv` algorithm (a) contains several loops within a forward wave. Although the same arrays are referenced in consecutive loops, the data accessed in the beginning of the loop is displaced by data referenced at the end of the previous loop. The loop fused version of `tomcatv` (b) performs all operations of the forward wave on one row of the arrays. This results in speedups of 1.60, 1.47, 1.28 on the DECstation 5000/240, 5000/125, and 5000/200 respectively.

Figure 25: Original tomcatv pseudocode (a), and loop-fused tomcatv (b).

ward pass loops, there is the additional forward pass (loop 5) to add the errors to the X and Y arrays after the backward pass (loop 4) over the RX and RY arrays. The RX and RY arrays are referenced in the same order in each loop of the forward pass (loops 1, 2, 3). However, recall that each array is 0.5M-Bytes in size, which is much larger than the 64K-Byte data cache. Hence, the elements referenced at the start of one loop are not in the cache at the start of the next loop.

The solution is to improve temporal locality by restructuring the program so that all allowable operations are performed on an element when it is resident in the cache. Transforming the program via loop fusion (see Figure 25) merges these loops so that the program contains only one forward loop and one backward loop. I can not perform the operations of both the forward pass and backward pass in the same loop because of data dependencies. Since the error corrections are not used in the last iteration, I can fold the addition of error corrections into the forward pass after the first iteration. Loop fusion, in addition to array merging, produced a speedup of 1.60 on the DECstation 5000/240, 1.47 on the 5000/125 and 1.28 on the 5000/200. These speedups are not as high as I expected because of an increase in the number of conflict misses and a slight increase in the number of instructions executed.

3.5.5 spice

`spice` (`spice2g6`) is an analog circuit simulator written in FORTRAN. The primary data structure is a sparse matrix, which is implemented by several arrays. In particular, there are separate arrays for row pointers, row numbers, column pointers, column numbers, and values. CProf shows that two source lines accessing the row pointer and row number arrays cause 34% of the cache misses. Another two source lines accessing the column pointer and column number arrays contribute an additional 12% of the cache misses. Each pair of source lines is contained in a small loop that locates an element (I,J) in the sparse matrix. CProf shows that the majority of the misses caused by these source lines are conflict misses, indicating a mapping problem. Again, the X-windows interface of CProf allows us to quickly observe that the row (column) pointer and row (column) number arrays are nearly always accessed with the same index. Merging the pointer and number arrays to improve spatial locality, results in a speedup of 1.34 on the DECstation 5000/240, 1.26 on the 5000/125 and 1.25 on the 5000/200.

3.5.6 dnasa7: The NASA kernels

`dnasa7` is a collection of seven floating-point intensive kernels also known as the NAS kernels: `vpenta`, `cholesky`, `btrix`, `fft`, `gmtry`, `mxm`, and `emit`. Each kernel initializes its arrays, copies them to working arrays, then calls the application routine. I discuss the kernels separately, to better describe the cache optimizations. I did not study `emit`, a vortex generation code, or `fft`, a fast Fourier transform code: `emit` has a very low miss ratio on a 64-Kbyte data cache (0.8%), and shuffling FFTs have inherently poor cache performance. The speedup I obtained for the entire collection of kernels is 1.51 on the DECstation 5000/240, 1.30 on the 5000/125, and 1.24 on the 5000/200.

3.5.6.1 vpenta

The vpenta kernel simultaneously inverts three pentadiagonals, a routine commonly used to solve systems of partial differential equations. CProf first finds that the miss ratio is a startling 36%, mostly due to conflict misses. Using CProf to identify the mapping problems, I discovered two nested loops responsible for over 90% of the cache misses. One loop accesses three arrays while the other accesses eight arrays. Recall that I can eliminate conflict misses by changing the allocation of data structures or changing the order that they are accessed. Inspection of the source code reveals that both of these techniques can be applied. I discovered that the loops could be interchanged to traverse the arrays in column order and also identified three opportunities for array merging. These modifications result in speedups of 2.93 on a DECstation 5000/240, 2.10 on a 5000/125 and 1.85 on the 5000/200. It is interesting to note that the original code runs slower on the 5000/240 than on the 5000/200, despite the 60% faster processor cycle time. This is apparently due to the higher miss penalty (the two machines use the same DRAMs, but the 240 incurs approximately 100ns additional delay due to an asynchronous interface). Loop interchange not only increases spatial locality, but results in a sequential access pattern that the 240’s prefetch logic can exploit. The 5000/240 has a speedup of 1.3 over the 5000/200 on the modified code.

As with tomcatv, running CProf on the modified version of vpenta shows that capacity misses now dominate. Fusing loops to improve temporal locality by eliminating multiple passes over the same arrays, results in speedups (over the original version) of 3.46, 2.53, 2.14 on the 5000/240, 5000/125 and 5000/200 respectively.

3.5.6.2 cholesky

cholesky performs cholesky decomposition and substitution. CProf reveals a large number of capacity misses in two nested loops. Inspection of the source code identifies an array traversed in row-major, rather than column-major, order. Statically transposing the array (effectively performing loop interchange but with much simpler code modification), results in speedups of 1.32 on the DECstation 5000/240, 1.16 on the 5000/125 and 1.13 on the 5000/200. Blocking can also be applied to cholesky [41], but I chose to apply a much simpler transformation.

3.5.6.3 btrix

btrix is a tri-diagonal solver. CProf shows that most of the misses are again capacity misses that occur in two nested loops. As always, I first checked the array reference order and immediately noticed that one array is traversed in row order. I also observed that statically transposing this array would allow fusion of six different loops. Notice that I was able to apply several transformations after a single run of CProf. On the DECstation 5000/240, I obtain a speedup of 1.48, 1.32 on the 5000/125 and 1.27 on the 5000/200.

```
        DO 8  I = 1, MATDIM
                RMATRX(I,I) = 1.D0 / RMATRX(I,I)
            DO 8  J = I+1, MATDIM
                    RMATRX(J,I) = RMATRX(J,I) * RMATRX(I,I)
                    DO 8 K = I+1, MATDIM
                        RMATRX(J,K) = RMATRX(J,K)
                            - RMATRX(J,I) * RMATRX(I,K)
8       CONTINUE
```

(a)

```
        DO 8   I = 1, MATDIM
            RMATRX(I,I) = 1.D0 / RMATRX(I,I)
            DO 81 J = I+1, MATDIM
                RMATRX(J,I) = RMATRX(J,I) * RMATRX(I,I)
81      CONTINUE
        DO 8 K = I+1, MATDIM
            DO 8 J = I+1, MATDIM
                RMATRX(J,K) = RMATRX(J,K) - RMATRX(J,I) * RMATRX(I,K)
8        CONTINUE
```

(b)

Figure 26: Gaussian elimination loops: (a) original; (b) interchanged.

3.5.6.4 gmtry

`gmtry` is a kernel dominated by a Gaussian elimination routine (see Figure 26). CProf finds that 99% of the misses, mostly capacity, occur in the Gaussian elimination loop; inspection shows that the RMATRX is traversed in row order. Interchanging the loops, which is trivial in this case, results in a speedup of 2.52 on the DECstation 5000/240, and 1.48 on both the 5000/200 and 5000/125.

3.5.6.5 mxm

`mxm` is a matrix-matrix multiply routine. The naive matrix multiply algorithm is a well-known “cache buster”, because there is little data re-use between loop iterations. The SPEC `mxm` implementation does not use this simple algorithm, instead using a column-blocked implementation (described earlier in the “cookbook”) that re-uses the same four columns throughout the two inner-most loops. It is interesting to note that improving cache performance was not the original rationale for blocking `mxm`; instead, the intent was to improve the opportunity for vectorizing compilers to reuse the contents of vector registers in Cray supercomputers. In this case, the same transformation improves performance for both vector registers and caches. The standard SPEC column-blocked algorithm achieves a speedup of 1.38 over the naive algorithm on the DECstation 5000/240, 2.04 on the 5000/125 and 1.74 on the DECstation 5000/200. For larger matrices, a two-dimensional (row and column) blocked algorithm would perform better, but for the standard SPEC input size the extra overhead decreases performance.

3.5.7 Summary

I have demonstrated how cache profiling and program transformation can be used to obtain significant speedups on six of the SPEC92 benchmarks. The speedups range from 1.02 to 3.46, depending on the machine's memory system, with greater speedups obtained in the FORTRAN programs. Since FORTRAN77 does not support structures, many of the programs exhibit poor spatial locality. Improving the spatial locality by interleaving the elements of disjoint arrays provided substantial improvements in most of the FORTRAN programs. Using FORTRAN90, which provides structures, would greatly simplify these transformations. CProf was very effective at identifying when to merge arrays. Loop interchange also improved the spatial locality in the FORTRAN programs; in many programs, loop interchange is a trivial transformation that can be easily identified by inspection or a compiler. The temporal locality of FORTRAN programs was improved by loop fusion, which requires programmers to perform all allowable operations on data while in the cache, versus performing each operation in turn on all of the data. It is important to remember that some of the transformations discussed in this chapter may be counter-productive on machines with vector registers.

The C programs benefited from padding and alignment of structures, merging arrays into an array of structures, and changing the declaration of a variable to pack more elements into a single cache block. Notice that padding and packing are opposite approaches, and which to use is dependent on the program being profiled.

3.6 Conclusion

As processor cycle times continue to decrease faster than main memory cycle times, memory hierarchy performance becomes increasingly important. Programmers can mentally simulate cache behavior to help select algorithms with good cache performance. Unfortunately, actual cache performance does not always match the programmer's expectations, and many programs are too complex to fully analyze the interactions between memory reference patterns, data allocation, and cache organization. In these cases, a tool like CProf becomes an important element in a programmer's tool box. A crucial factor for the success of a tool like CProf is the time required to simulate the memory system. The active memory abstraction satisfies this requirement by providing the framework necessary to implement an efficient cache profiler. My active memory implementation of CProf executes only 3.4 to 15 times slower than the original application.

CProf provides cache performance information at the source line and data structure level, allowing a programmer to identify hot spots. The insight CProf provides, by classifying cache misses as compulsory, capacity, and conflict, helps programmers select appropriate program transformations that improve a program's spatial or temporal locality, and thus overall performance.

Chapter 4

Dynamic Self-Invalidation

4.1 Introduction

Caches provide a cost-effective alternative to supercomputer-style memories by alleviating the disparity between fast processors and slow, inexpensive main memory. Unfortunately, caches do not work well for all access patterns. Furthermore, microprocessors are designed for a volume market, sacrificing performance for lower cost, and are unlikely to reach supercomputer levels of performance. This potential limitation is mitigated by connecting multiple cost-effective microprocessors, and exploiting parallelism to achieve supercomputer performance, while exploiting the cost-effectiveness of commodity microprocessors.

Parallel processing introduces a plethora of complexities, perhaps the most important of which is programming effort. Shared-memory multiprocessors simplify parallel programming by providing a single address space, even when memory is physically distributed across many workstation-like processor nodes. Most shared-memory multiprocessors use cache memories to automatically replicate and migrate shared data and implement a coherence protocol to maintain a consistent view of the shared address space [9,28,40,49].

Write-invalidate protocols allow multiple processors to have copies of shared-readable blocks, but force a processor to obtain an exclusive copy before modifying it [3,8, 9,29,49,75]. Directory-based protocols invalidate outstanding copies by sending explicit messages to the appropriate processor nodes [3,8,75]. When a node receives an invalidation message, it invalidates its local copy and sends an acknowledgment message back to the directory. (This message also contains the data for exclusive blocks).

The performance of these protocols might improve significantly if I could eliminate the invalidation messages (without changing the memory semantics). An oracle could do this by simply making the processors replace blocks just before another processor makes a conflicting access, allowing the directory to immediately respond with the data, and avoiding the need to send invalidation messages. Thus, the processors would *self-invalidate* their own blocks instead of waiting for the directory to send explicit invalidation messages. This would improve performance by reducing the latency and bandwidth required to satisfy conflicting memory requests.

The principal contribution of this work is a practical approach for *dynamic self-invalidation (DSI)* of cache blocks. I show how the directory can dynamically identify which blocks should be self-invalidated, how this information is conveyed to the cache in response to a miss, and how the cache controller can later self-invalidate the selected blocks at an appropriate time.

Self-invalidation cannot make a correct program incorrect, since it has exactly the same semantics as a cache replacement. However, self-invalidating blocks too early can cause unnecessary cache misses, hurting rather than helping performance. Therefore, the DSI implementation must minimize the number of explicit invalidations without significantly increasing the number of misses.

In this chapter I evaluate two methods for identifying which blocks to self-invalidate: additional directory states and version numbers. My results indicate that 1-bit version numbers generally performs better than the additional state method, and there is no benefit from using larger version numbers. Version numbers allow processors to identify blocks for self-invalidation independent of other processors. In contrast, all processors make the same decision using additional states.

I also investigate two techniques for the cache controller to self-invalidate the blocks: using a FIFO buffer and using selective cache flushes at synchronization operations. Simulations show that selective flushing is more effective because the FIFO’s finite size can cause self-invalidation to occur too early, resulting in additional cache misses. I propose two implementations for selectively flushing blocks from the cache: a modified flash clear circuit and a hardware linked list. Both of these techniques process only blocks marked for self-invalidation and can be implemented in the second level cache controller.

The benefit of DSI is significant when coherence traffic dominates communication. For most of my benchmarks, a sequentially consistent memory system with DSI performs comparably to a weakly consistent implementation that allows up to 16 outstanding requests for exclusive blocks, but stalls on read misses. Execution times with the sequentially consistent protocol improve by up to 41%, depending on the cache size and network latency. When used with a weakly consistent memory system, DSI can eliminate both invalidation and acknowledgment messages by allowing nodes to obtain copies of a cache block without updating the directory state. My results show that while DSI improves the performance of one benchmark (sparse) by 18%, it has little effect on execution time for most programs. However, combining DSI and weak consistency can eliminate 50–100% of the invalidation messages, reducing the total number of messages by up to 26%. Note that my implementations to do not require any changes to the processor chip. The results presented in this chapter may not hold for implementations that change the processor chip.

This work evaluates DSI in the context of a conventional all hardware shared-memory multiprocessor. However, DSI is applicable to many other types of shared-memory multiprocessors. In particular, the effectiveness of cache memories in shared-memory multiprocessors has produced systems that utilize a portion of main memory to cache remote data [28,61]. In these systems, data is seldom replaced from the cache, and coherence traffic dominates communication. DSI should be of increased benefit in these systems.

Another trend in multiprocessor design is to reduce cost by connecting existing workstations [5]. However, these systems generally have relatively slow networks, and my results indicate that DSI should have increased benefit because coherence overhead increases with the network latency.

This chapter is organized as follows. Section 4.2 reviews invalidation-based coherence protocols and discusses related work. Section 4.3 presents dynamic self-invalidation and discusses the design space. Section 4.4 describes my implementations of dynamic self-invalidation protocols, Section 4.5 evaluates their performance, and Section 4.6 concludes this chapter.

4.2 Background and Related Work

DSI techniques are applicable to hardware [49], software [63], and hybrid systems [9,40,61]. In this chapter, I evaluate DSI in the context of a full-map, directory-based hardware cache coherence protocol [3]. I assume a typical write-invalidate protocol with three states (see Figure 27): no outstanding copies (Idle), one or more outstanding shared-readable copies (Shared), or exactly one outstanding readable and writable copy (Exclusive). A processor must obtain an exclusive copy of a block before modifying it; the directory enforces this by sending explicit invalidation messages to eliminate any outstanding copies.

The overhead of these invalidation messages is particularly significant under *sequential consistency* [42], the programming model most programmers implicitly assume. A multiprocessor is sequentially consistent if the execution corresponds to some interleaving of the processes on a uniprocessor. Conventional directory-based write-invalidate coherence protocols maintain sequential consistency by stalling a processor on a write miss until it receives acknowledgment that all cached copies have been invalidated,[1] as shown in

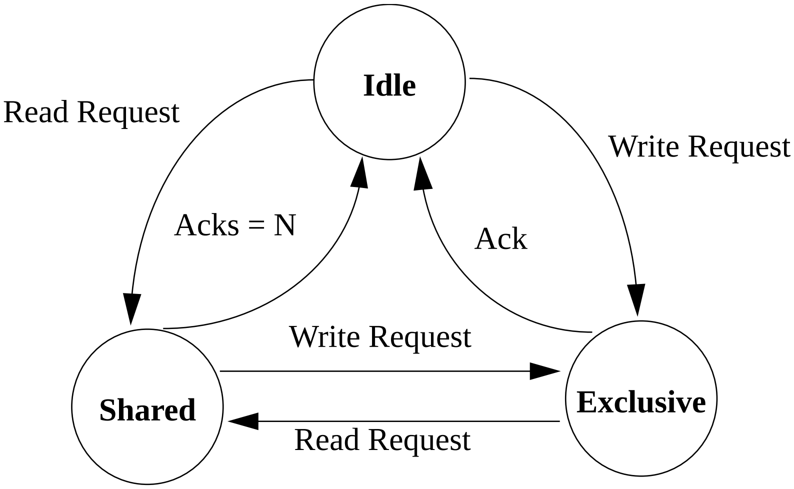


Figure 27: Cache Block Directory States

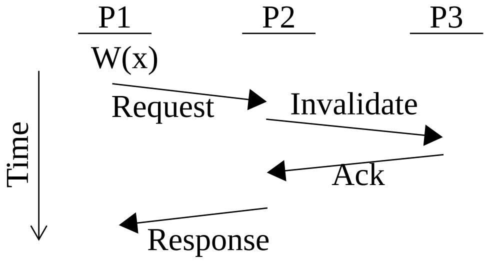


Figure 28: Coherence Overhead

Figure 28. Unfortunately, the latency of sending invalidations and collecting acknowledgments may lie on the program's critical path, and therefore degrade performance.

Self-invalidation techniques can eliminate the invalidation (Invalidate) and acknowledgment (Ack) messages from the sequence, illustrated in Figure 28, significantly reducing the latency required to obtain a cache block. When self-invalidation is performed perfectly, read requests always find the block in state Idle or Shared, and write requests always find the block in state Idle.

Previous self-invalidation techniques rely on memory system directives inserted by the compiler, profile-based tools, or the programmer. *Compiler-directed coherence* [10,16,18,54] eliminates the directory, placing the entire burden of maintaining cache coherence on the compiler. Unfortunately, this technique requires sophisticated analysis, and has only been demonstrated to work well for regular scientific applications and one-word cache blocks.

[1] It is possible to have P2 respond immediately with the data, and P3 send an acknowledgment directly to P1 [49]. However, P1 stalls until the acknowledgment is received.

Other self-invalidation techniques combine memory system directives with a conventional directory-based write-invalidate protocol. In CICO, the programmer [29,82] or a profile-based tool [11] annotates the program with `check_in` directives to inform the memory system when it should invalidate cache blocks. In contrast to compiler-directed coherence, `check_in` directives are only performance hints to the memory system; the directory hardware is still responsible for correctness.

Self-invalidation can be used with other techniques that reduce the impact of coherence overhead. Prefetching cache blocks before their expected use hides the latency to obtain a cache block [55,27]. Multithreading [69,27] tolerates latency by rapidly switching to a new computation thread when a remote miss is encountered. Migratory data optimizations [14,72] speculate about future write requests by the same processor when responding to a read request. Self-invalidation is complementary to these optimizations and could be combined with them. For example, the SPARC V9 `prefetch-read-once` instruction [7] indicates that a block should be prefetched, but then self-invalidated after the first reference.

Weak consistency models [2,20,24] also reduce the impact of coherence overhead. A system that provides weak consistency appears sequentially consistent, provided that the program satisfies a particular synchronization model [2]. Weak consistency models allow the use of memory access buffering techniques—e.g., write buffers. They also allow the directory to respond with the data in parallel with the invalidation of outstanding copies, and the processor can proceed as soon as it receives the data. The acknowledgments can be sent directly to the requesting processor [49], or collected by the directory which forwards a single acknowledgment. The processor stalls at synchronization operations, depending on the specific consistency model, until all preceding writes are acknowledged. As discussed in Section 4.3.3, self-invalidation can eliminate acknowledgment messages when combined with weak consistency. Adve and Hill proposed a similar scheme for sequential consistency [1]; however, their technique requires modification of the level-one cache to observe all cache accesses.

4.3 Dynamic Self-Invalidation

In this section, I present a general framework for performing dynamic self-invalidation (DSI). Similar to other forms of self-invalidation, DSI attempts to ensure that data is available at the directory (home node) when another processor requests access. However, DSI does not rely on programmer intervention; instead, self-invalidation is performed automatically by the coherence protocol.

Write-invalidate coherence protocols generally involve the following operations (see Figure 29):

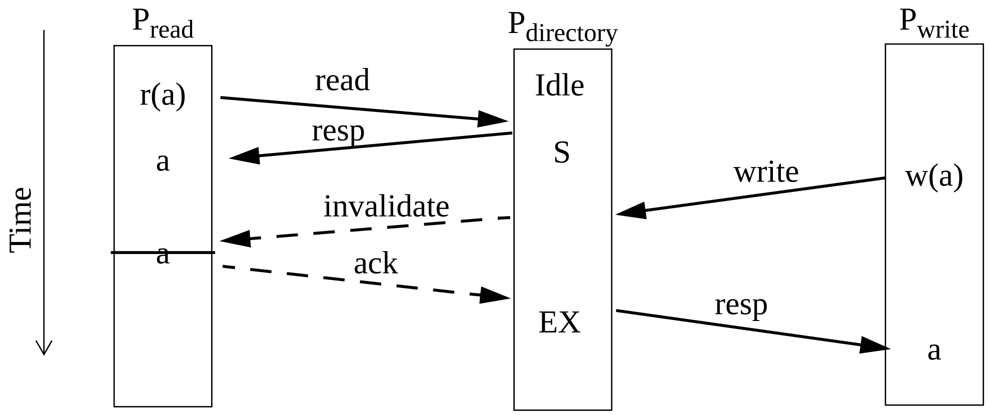


Figure 29: Write Invalidate Coherence Protocol

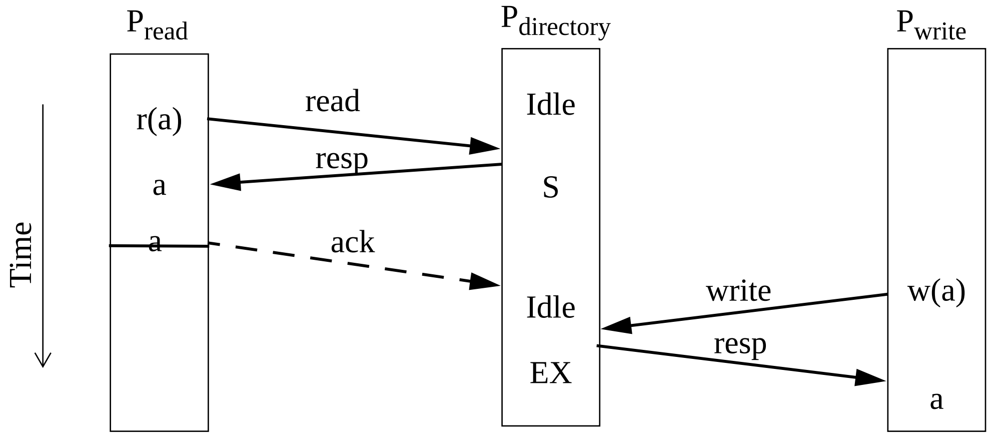


Figure 30: DSI Decoupled Identification and Invalidation of Cache Blocks

1. identify that a cache block requires invalidation,
2. perform the invalidation, and
3. acknowledge the invalidation, if necessary.

Conventional protocols tightly couple the identification of a block for invalidation (step 1) with its invalidation (step 2). The directory explicitly invalidates outstanding copies when servicing cache misses. In contrast, DSI decouples these steps, speculatively identifying which blocks to invalidate when they are brought into the cache, but deferring the invalidation itself to a future time (see Figure 30).

The remainder of this section discusses the dynamic self-invalidation design space. Section 4.3.1 discusses techniques for the directory controller, cache controller, or software to identify blocks for self-invalidation. Section 4.3.2 discusses performing self-invalidation with the cache controller or software, and Section 4.3.3 discusses the acknowledgment of invalidation messages.

4.3.1 Identifying Blocks

Identifying blocks for self-invalidation requires speculating the likelihood that a block will be invalidated in the near future. This identification can be implemented by the directory controller, the cache controller, software, or any combination of the three.

Software approaches issue directives to the memory system to identify which blocks to self-invalidate. Unfortunately, these techniques require either programmer annotations, a sophisticated compiler, or a profile-based tool. Furthermore, implementing these directives either requires special instructions not present in all instruction sets, or additional memory mapped loads and stores. These additional instructions may increase program execution time.

In this work, I focus on hardware techniques that automatically identify blocks for self-invalidation. A directory controller can identify a block for self-invalidation by maintaining a history of its sharing pattern. When servicing a request for a cache block, the directory uses this extra information to predict if the block is likely to be invalidated in the future, and conveys this information to the caching node with the response.

Similarly, a cache controller can identify blocks for self-invalidation by maintaining information for recently invalidated blocks [17] (e.g., the number of times a block is invalidated). When servicing a cache miss, this history information is used by the controller to decide if it should self-invalidate the block at a later time.

4.3.2 Performing Self-Invalidation

Software, hardware, or a combination can be used to perform self-invalidation. The caching node must record the identity of the blocks selected for self-invalidation and invalidate them at a point in the future that maximizes performance.

Systems that maintain cache coherence in software [36,40,61,63] can use arbitrary data structures to store block identities. The blocks are self-invalidated using the same primitives required to process explicit invalidation messages. Alternatively, hardware managed caches can maintain a hardware data structure, such as an auxiliary buffer or an extra bit in the cache tag. Software examines this hardware data structure, self-invalidating the blocks by issuing directives to the memory system.

There are many alternatives for performing self-invalidation entirely in hardware, and thus remove the burden from the programmer or compiler. In Section 4.4, I present two hardware methods that can be implemented in the (second-level) cache controller. The first scheme uses a first-in-first-out (FIFO) buffer; blocks are self-invalidated when they fall out of the buffer. The second technique performs self-invalidation at synchronization operations using custom hardware.

4.3.3 Acknowledging Invalidation Messages

In conventional directory-based, write-invalidate protocols, the directory records—or *tracks*—the identity of nodes holding copies of a cache block. By tracking blocks, the directory can always explicitly invalidate cached copies when necessary. Thus, self-invalidation is semantically equivalent to a cache replacement with notification, and places no restrictions on the memory consistency model. Some systems do not notify the directory when a block is replaced from the cache [9]. Instead, the directory always sends an invalidation to the node, and an acknowledgment is always required. Self-invalidation of tracked blocks can reduce latency and eliminate invalidation messages. However, acknowledgment messages are still required to inform the directory that a node has invalidated its copy of the block.

I can eliminate both invalidation and acknowledgment messages by guaranteeing to self-invalidate blocks at specific points according to the memory consistency model. For these blocks—called *tear-off* blocks—the directory does not track the outstanding copy. Note that tear-off blocks are only useful for shared-readable blocks, since the acknowledgment for exclusive blocks is generally coupled with the transfer of modified data.

Scheurich observed that the invalidation of a cache block could be delayed until the subsequent cache miss and still maintain sequential consistency [62]. The intuition behind this observation is that a processor can continue to access data until it “sees” new data generated by another processor. To maintain sequential consistency, the cache controller must invalidate tear-off blocks at subsequent cache misses. Therefore, a cache may contain at most one tear-off block.

A further caveat is that using tear-off blocks with sequential consistency does not guarantee forward progress. If a processor obtains a tear-off block containing a spin lock [4], it may never experience a subsequent cache miss. The spin lock will never be invalidated, and the processor will not proceed. To overcome this, the tear-off block could be self-invalidated periodically, e.g., at context switches.

Tear-off blocks are potentially much more significant under weaker consistency models. A processor can cache multiple tear-off blocks since the model does not guarantee that a processor can “see” data generated by another processor until it performs a synchronization operation. By self-invalidating its local tear-off blocks at each synchronization point, a processor ensures that it can see all other processors’ modifications to shared data (see Figure 31).

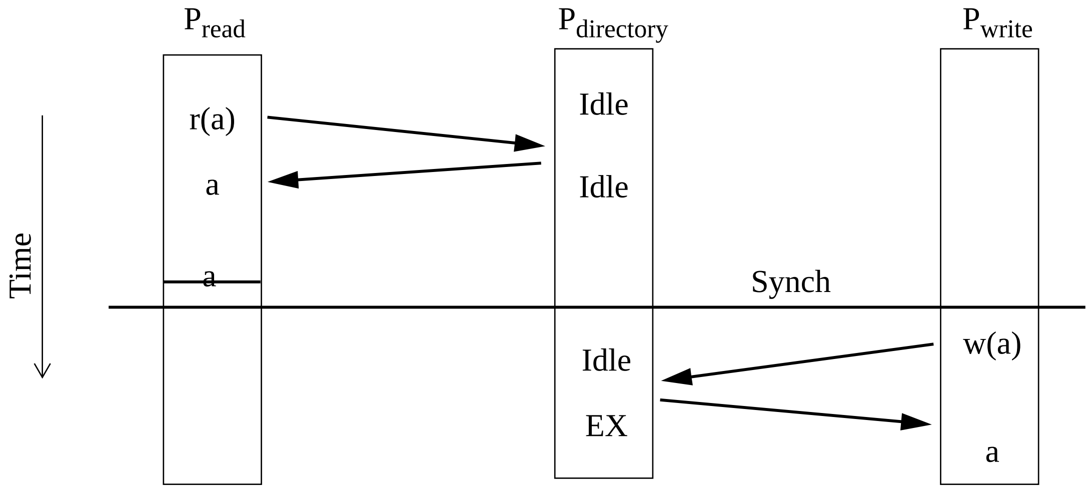


Figure 31: DSI Using Tear-Off Blocks

4.4 Implementation

In this section, I present several different implementations of DSI. I focus on techniques where the directory identifies which blocks should be self-invalidated, and the cache controller performs the self-invalidation. The directory conveys self-invalidation information to the cache when responding to a miss. Blocks that are not self-invalidated are explicitly invalidated in the conventional manner. I describe two methods for the directory to identify blocks for self-invalidation, followed by two techniques for the cache to perform the invalidations. I evaluate these implementations in Section 4.5.

4.4.1 Identifying Blocks

The directory controller provides a single point for monitoring a cache block’s sharing patterns. This section presents two techniques for the directory controller to identify which blocks should be self-invalidated: additional states and version numbers. Both implementations are extensions to a standard three state, full-map, directory-based write-invalidate protocol, such as Dir_nNB [3]. Appendix B provides more details on each of the protocols.

Both implementations use the sharing history to speculate about the future: blocks that have recently had conflicting accesses—and hence would have needed invalidations—are candidates for self-invalidation. Thus, shared-readable blocks are marked for self-invalidation if they have been modified since the last reference by the processor. Likewise, exclusive blocks are marked for self-invalidation if they have been read or modified by a different processor since the writing processor's last access.

Through experimentation I found two special cases where it is better to avoid self-invalidation. First, blocks are not self-invalidated from the home node's cache (although on other systems this may not be the recommended approach, since local invalidation latencies may be high.) Second, under sequential consistency, exclusive blocks are not marked for self-invalidation if the writing processor had a shared-readable copy and there are no other outstanding copies. This upgrade case can cause unnecessary self-invalidation of exclusive blocks, which causes additional write misses, degrading performance for some programs under sequential consistency. This special case is not needed under weak consistency, since the write buffer hides the latency of the additional write misses.

4.4.1.1 Additional States

The first implementation uses four additional states to identify which blocks should be self-invalidated. When servicing a read request, the directory responds with a self-invalidate block if the current state is exclusive. These blocks enter a new state (Shared_SI) that causes all subsequent read requests to obtain a block marked for self-invalidation. I also add two new states (Idle_X, Idle_S) to detect transitions into the idle state from the exclusive or shared-readable state resulting from self-invalidation. Finally, I add one state (Idle_SI) to detect transitions into the idle state resulting from the cache replacement of a self-invalidate block.

The directory responds to a write request with a self-invalidate block if the current state is: Shared, Shared_SI, Exclusive, Idle_S, Idle_SI, or Idle_X where a different processor had the block exclusive. Read requests obtain a self-invalidate block if the current state is: Exclusive, Idle_X, Shared_SI or Idle_SI.

If tear-off blocks are supported, each directory entry requires one additional bit to indicate that there is more than one outstanding tear-off block. This bit allows correct identification of exclusive blocks for self-invalidation when servicing a write request from a processor that had a tear-off block.

4.4.1.2 Version Numbers

Version numbers provide an alternative scheme that identifies when blocks are modified by different processors. This additional information allows processors to decide independently whether to obtain a self-invalidate block. By contrast, all processors make the same decision using additional states to identify blocks.

The directory maintains a version number for each block and increments it each time any processor requests an exclusive copy. This scheme requires the cache controller to store the version number with the associated block. On a miss, if there is a tag match, and

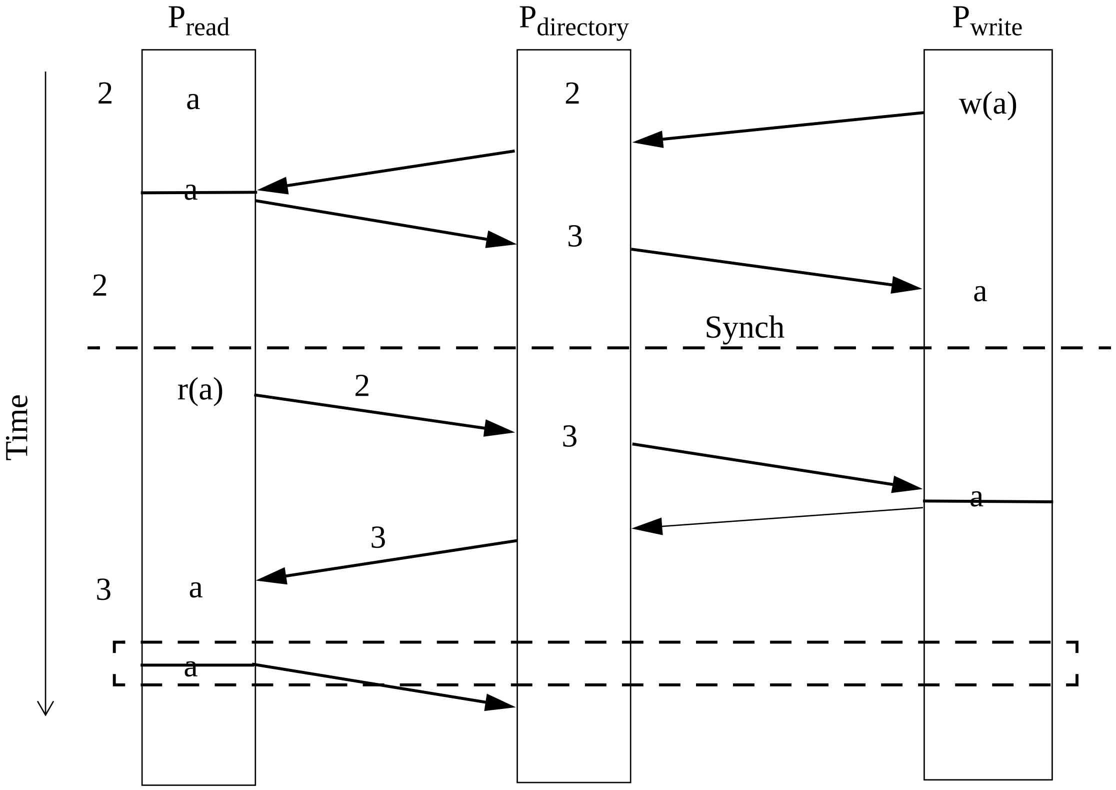


Figure 32: Identification of Blocks Using Version Numbers

if the block is invalid, the corresponding version number is sent with the request for the block. The directory responds with a self-invalidate block if the current version number is different from the version number of the request. If the cache controller does not provide a version number (i.e., there is not a tag match), the directory responds with a normal block.

Consider the producer-consumer example shown in Figure 32. Initially, processor Pread has a shared-readable copy of a block (a). Processor Pwrite writes to the same block, causing the directory to increment the version number (from 2 to 3) and invalidate Pread’s copy. When Pread next accesses the block, it incurs a cache miss and includes the cached version number for the block (2). Upon receipt of the request message, the directory compares the version number in the request (2) to the current version number (3) and marks the block for self-invalidation since they are not equal. Pread records the identity of the block for self-invalidation when it receives the response and self-invalidates the block at a future time.

Although I assume the version numbers are cached by the cache controller, it is possible to cache the version numbers at the directory,[1] thus avoiding modifications to the cache controller. Since the version number is only a performance hint, I can use a small number of bits and allow wrap-around without violating correctness.

Exclusive blocks are identified for self-invalidation when the version numbers do not match, and when the version numbers do match but another processor has read the block. To detect this situation, I add two bits to each directory entry that count the number of shared-readable copies distributed for the current version of the block. Each time the directory responds with a shared-readable block, a ‘one’ is shifted into the low-order bit. Both bits are cleared when the version number is incremented. Therefore, write requests obtain a self-invalidate exclusive block if either the version numbers do not match, or the current version has been read by at least two processors (which may include a previous read by the writing processor).

4.4.2 Performing Self-Invalidation

In this section, I present two techniques for the cache controller to self-invalidate blocks using information readily available from many commodity processors.

The first implementation uses a first-in-first-out (FIFO) policy for self-invalidation blocks. When the cache controller receives a self-invalidate block, it records the identity of the block in the FIFO. Blocks are self-invalidated when an entry in the FIFO is replaced. In addition, if I can identify synchronization operations, such as `test&set` or `swap`, then I can also flush the FIFO at those points.

Implementing the FIFO requires the addition of a small memory to store the identity of the blocks to self-invalidate. This buffer—similar to a victim cache [34] or the HP PA7200 assist cache [39]—is unlikely to exceed 64 entries. Nonetheless, this is an attractive approach since it does not rely on any information from the processor.

If the cache controller can identify synchronization operations, then there are other schemes for performing self-invalidation. In particular, I can eliminate the FIFO and flush all self-invalidate blocks from the cache after one or more synchronization operations [12,22]. In this chapter, I focus on invalidating blocks at each synchronization point.

The precise implementation depends on the specific DSI protocol. All the implementations require an additional bit, *s*, associated with each cache tag. The *s* bit indicates that the block should be self-invalidated, which is accomplished by clearing the corresponding valid bit. When a new block is brought into the cache, the *s* bit is set if the block has been selected for self-invalidation.

[1] This strategy was suggested by Peter Hsu of Silicon Graphics Inc.

Self-invalidation of tracked blocks requires the cache controller to send an acknowledgment (or notification) message to the directory. The control logic must find which blocks to self-invalidate—marked by the *s* bit—and recreate the full addresses by concatenating the cache index with the cache tag.

The naive implementation sequentially examines each cache frame, self-invalidating the block and sending a message if necessary. However, the overall latency will be proportional to the number of cache frames, even though many blocks may not be self-invalidated.

I can reduce this latency using a circuit that sequences through only the blocks that must be self-invalidated. One implementation uses a modified flash clear circuit to determine the next cache set that contains a block to self-invalidate, and requires an encoder to recreate the cache index (See Figure 33). This encoder is roughly the same size as the set index decoder, and, for set-associative caches, it can be shared by all cache frames in the same cache set.

Alternatively, I could use a hardware linked list (see Figure 34) which adds a pointer to each cache set, and maintains a head and a tail pointer. The pointers store the cache index of the next block to self-invalidate. When a self-invalidate block is brought into the cache, its corresponding pointer is assigned to the current value of the tail, and the tail is updated to point to the new block. At synchronization operations, the list is traversed from tail to head. Set-associative caches require only one pointer per cache set. A set is inserted in the list when it receives its first self-invalidate block; during self-invalidation, the set must be searched for all blocks with the *s* bit equal to one.

These implementations achieve similar performance, processing only blocks that require self-invalidation. Note that self-invalidation of tracked blocks can overlap with the execution of the processor, staging out the messages and possibly avoiding severe network congestion or synchronization delays. However, the quantitative results in this chapter assume that the processor does not proceed past synchronization points until all blocks are self-invalidated, and that messages are injected as rapidly as the network can accept them.

Self-invalidating tracked blocks always requires messages to the directory, and the latency to perform self-invalidation is proportional to the number of blocks self-invalidated. However, when both tear-off blocks and exclusive blocks are self-invalidated, only the exclusive blocks require a message to the directory. The tear-off blocks can be self-invalidated in a single cycle using a simple flash clear circuit; the exclusive blocks must be sequentially self-invalidated using one of the techniques described above.

Although in this chapter I assume a modified cache controller, I can avoid these modifications by implementing the linked list as a separate hardware structure. This structure must be able to invalidate blocks from the processor's cache. Therefore, it can be placed

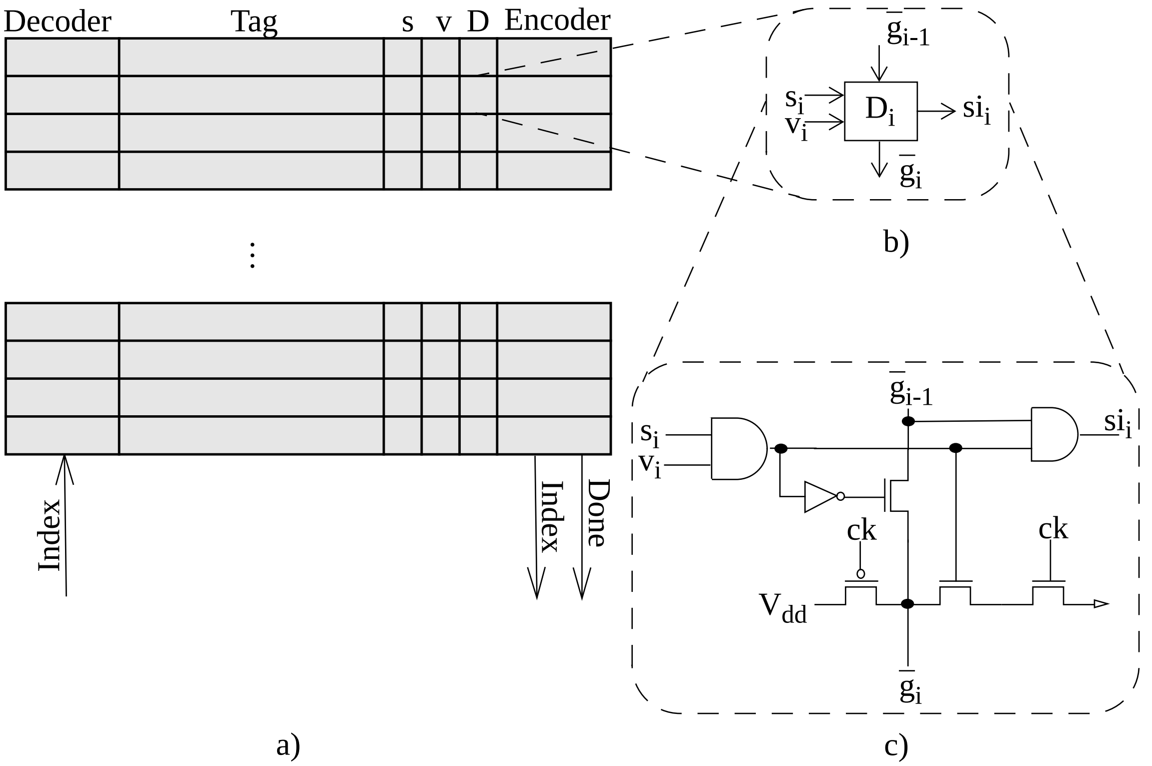


Figure 33: Self-Invalidation Circuit

The implementation of this circuit is similar to a Manchester carry chain [80]. A cache block is selected for self-invalidation only if no preceding blocks require self-invalidation. If a block should be self-invalidated, its *s* and *v* bits are set, and the frame *generates* a self-invalidate signal, otherwise it *propagates* the signal from the previous cache frame, $g_i = (s_iv_i) + g_{i-1}$. A cache frame is selected, i.e., the signal *si* bit is active, only if it generates a self-invalidate and no previous cache frame generated a self-invalidate, $si_i = s_iv_ig_{i-1}$. This results in a single *si* bit being set for all the cache frames, see Figure 33c.

The *si* bit is used to clear the corresponding valid bit and enable the output of the cache tag. To create the corresponding block address, the *si* bits are used as the input to an encoder to create the cache index, see Figure 33a. This encoder is roughly the same size as the decoder used to select a cache set. The control logic cycles until there are no blocks to self-invalidate.

Although this implementation can be applied to any cache configuration, the size of the encoder can be reduced in caches with associativity greater than one, since they have multiple cache frames with the same cache index. All cache frames generate the self-invalidate signal, *si*, in the same manner as a direct-mapped cache. However, rather than using each *si* bit directly as an encoder input, a single input is generated for each cache set by the OR of the *si* bits for each frame in the set.

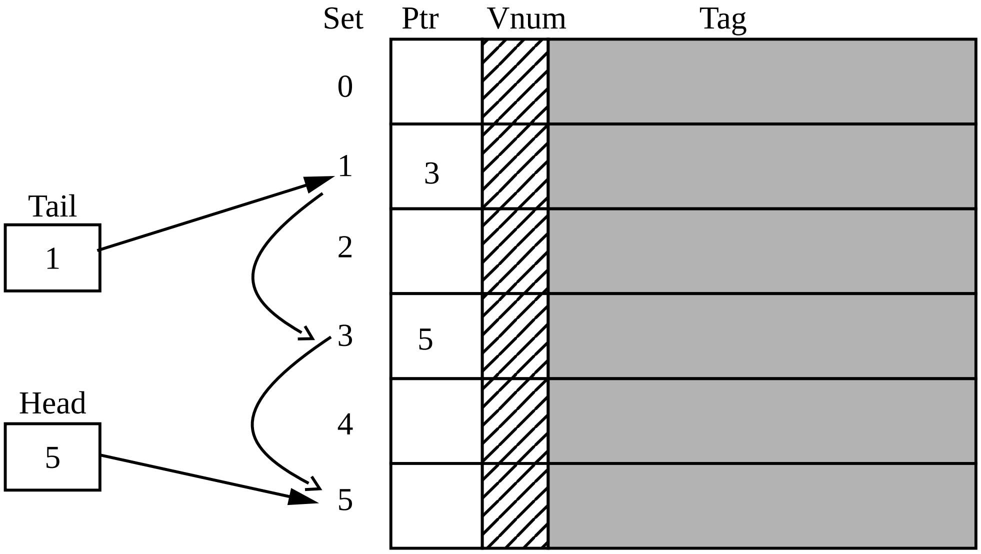


Figure 34: Hardware Linked List for Self-Invalidation

between the level two cache and the system bus, or it can be a bus master, similar to the remote access cache on DASH [49]. This approach eliminates the *s* bit from each cache frame, forcing each self-invalidate block to have an entry in the list, precluding the use of the flash clear circuit. Furthermore, if the device is a bus master, additional overhead is introduced since each block that is self-invalidated would require a bus transaction.

4.5 Performance Evaluation

Now I evaluate the effectiveness of DSI by comparing it to a full-map protocol [3]. Section 4.5.2 evaluates the detection and self-invalidation mechanisms under sequential consistency. In Section 4.5.3, I evaluate the benefit of adding dynamic self-invalidation to a weak consistency implementation that allows up to 16 outstanding requests for exclusive blocks. Finally, Section 4.5.4 examines the impact of block size on DSI performance.

4.5.1 Methodology

I use a modified version of the Wisconsin Wind Tunnel [60] to simulate 32-processor systems with 256K-byte and 2M-byte 4-way set-associative caches with 32-byte blocks. Cache misses occupy the cache controller for 3 cycles and the directory controller for 10 cycles, plus message injection time. The message injection overhead is 3 cycles, with an additional 8 cycles if a cache block must be sent. I assume a constant 100 cycle network latency and do not model contention in the switches. However, contention is accurately modeled at the directory, cache and network interface. Instruction execution time is

.

Name	Input Data Set
Barnes	8192 bodies, 5 iterations
EM3D	192,000 nodes, degree 5, 5% remote
Ocean	98x98, 1 day
Sparse	512x512 dense, 5 iterations
Tomcatv	512x512 5 iterations

This table describes the benchmarks used in this chapter. Sparse is locally-written [82], EM3D is from the Berkeley Split-C group [15], Barnes and Ocean are from the Stanford SPLASH suite [65], and Tomcatv is a locally written, parallel version of the SPEC benchmark.

Table 8: Application Programs

obtained by modeling the SuperSPARC processor, which can issue up to three instructions per cycle. I assume that SPARC `swap` instructions and a hardware barrier, with a 100 cycle latency from the last arrival, are visible to the memory system.

The base cache coherence protocols are all full-map protocols. The sequentially consistent implementation stalls the processor on all misses. The directory invalidates outstanding copies and collects acknowledgments before forwarding the block to the requesting processor.

For weak consistency, I use a 16-entry coalescing write buffer. Each entry in the write buffer contains an entire cache block, and write misses that match an outstanding request are merged into the existing entry. The directory in my weak consistency protocol grants exclusive access to a block in parallel with the invalidation of outstanding shared-readable blocks. A single acknowledgment is sent to the owning processor after the directory collects the invalidation acknowledgments. The processor stalls at `swap` and `barrier` operations until all previous writes are acknowledged. The processor also stalls on read misses until the block is obtained.

I present results from five benchmarks in my evaluation of DSI (see Table 8.) I focus specifically on the parallel portion of the programs, clearing all statistics after initialization.

4.5.2 Sequential Consistency Results

I evaluate DSI in the context of sequential consistency, beginning with an evaluation of the detection mechanisms, described in Section 4.4.1. I assume that I have custom hardware to perform the self-invalidation at synchronization operations. This is followed by a discussion of performing self-invalidation with a FIFO buffer.

The main results from this study are that:

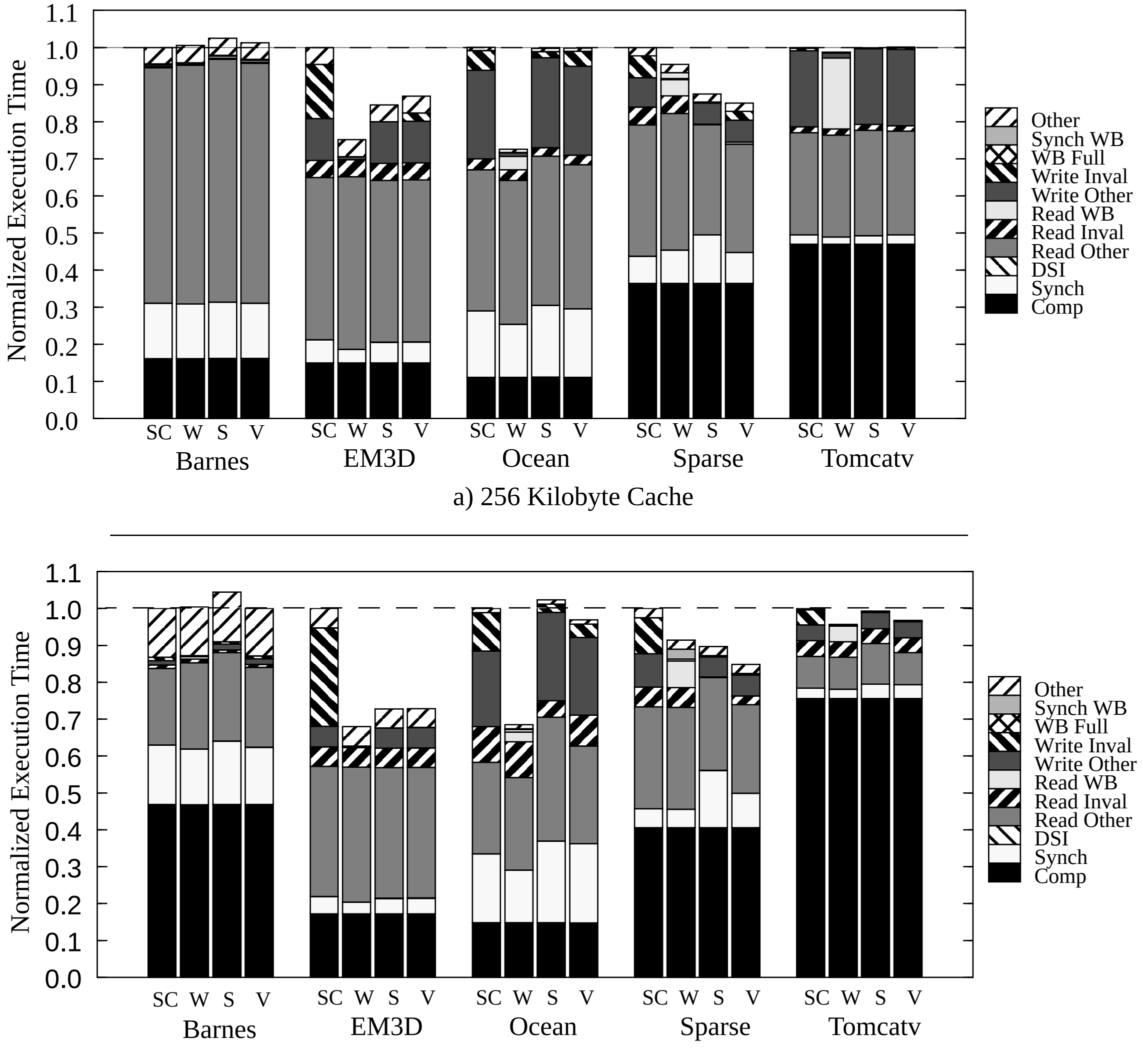
1. DSI can give sequential consistency performance comparable to an implementation of weak consistency.
2. Version numbers are more effective than additional states for detecting which blocks to self-invalidate, and a single bit for the version number is sufficient.
3. Performing self-invalidation at synchronization operations is better than using a finite-size FIFO.
4. DSI is most effective when coherence overhead dominates communication.

DSI improves execution time by up to 41%, depending on the cache size and network latency. For all but one of my benchmarks, these execution times are comparable to my weakly consistent implementation. Furthermore, the benefit of DSI is much larger when coherence overhead is high. When coherence overhead is low, neither weak consistency nor DSI have much effect on execution time.

4.5.2.1 Detection Mechanisms

In this section, I examine the performance of detecting blocks using additional states and version numbers. Figure 35 shows execution time normalized to the base sequentially consistent protocol. The left most bar is the base sequentially consistent protocol (SC), followed by the weakly consistent (W) and DSI protocols with additional states (S) and 4-bit version numbers (V), respectively. Based just on total execution time, the results indicate that sequentially consistent DSI achieves performance roughly comparable to the base weakly consistent implementation for all programs except `ocean`.

To look further, I refine execution time into computation, synchronization, read invalidation, read other, write invalidation, write other, and other (e.g., TLB misses and I/O). Read (write) invalidation is the time spent waiting at the directory for outstanding copies to be invalidated, and represents the maximum time DSI can eliminate. For weak consistency, I also include the time spent waiting at synchronization points for the write buffer to drain (synch wb), waiting on read misses for which there is already an outstanding write miss (read wb), and waiting when the write buffer is full (wb full). For the DSI protocols, I



a) 256 Kilobyte Cache

b) 2 Megabyte Cache

SC = Sequential Consistency, W = Weak Consistency, S = DSI using additional states, V = DSI using version numbers

Figure 35: Performance of DSI Under Sequential Consistency

include the time spent waiting for the self-invalidation to complete (DSI). Although, my simulations show that this time is too small to perceive.

This breakdown shows that `Barnes` has very little invalidation delay, and neither weak consistency nor DSI yield performance improvements.

`EM3D` spends most of its time waiting for cache misses. DSI reduces the write invalidation time, producing improvements within 5% of the weakly consistent protocol, which eliminates all write latencies. For the 256K-byte cache, execution time improves by 25%

for weak consistency, 15% for DSI using states, and 13% for the version number implementation. For the 2M-byte cache, improvements are 32%, 27%, and 27%, respectively. DSI does not reduce the read invalidation time because `EM3D` uses local allocation, and all modifications to shared data occur on the home node. Recall that blocks are not self-invalidated from the home nodes cache.

DSI has little effect on the execution time of `ocean` for either cache size because of unsynchronized accesses to shared data. In contrast, weak consistency reduces execution time by 27% for the 256K-byte cache and 32% for the 2M-byte cache.

For `sparse`, DSI reduces both read invalidation and write invalidation delays, *outperforming* weak consistency by as much as 10%. Weak consistency improves performance by 5% for the 256K-byte cache and 9% for the 2M-byte cache. DSI provides 13% and 10% improvements using additional states, and 15% for both cache sizes using version numbers.

For the 256K-byte cache, `tomcatv` shows no change in execution time for any of the protocols since its data set is too large for the cache. Weak consistency eliminates the write stall time, but read stalls increase because there is a read miss for a block with an outstanding write miss. For the larger cache, `tomcatv`'s execution time is dominated by computation, and weak consistency and DSI with version numbers improve execution time by only 4% and 3% respectively.

4.5.2.2 Version Number Size

The results presented thus far assume 4-bit version numbers. I now analyze the performance of smaller version numbers, specifically 1, 2, and 4 bit version numbers. Figure 36 shows the execution time for sequentially consistent and weakly consistent DSI normalized to the base sequentially consistent protocol as a function of the version number size. These results clearly show that DSI performance is independent of the number of bits used for version numbers—a single bit is sufficient, significantly reducing the space required to store the version numbers. This is primarily due to the producer-consumer nature of the benchmarks that I studied.

4.5.2.3 Impact of Network Latency

As processor cycle times continue to decrease relative to network latencies, the impact of coherence overhead increases. To evaluate the benefit of DSI under these conditions, I increased the network latency to 1000 cycles (10µs @ 100 MHz). This generally increases the benefit of both DSI and weak consistency. With a 256K-byte cache, weak consistency reduces execution time by 2% for `barnes`, 33% for `EM3D`, 32% for `ocean`, 15% for `sparse`, and 1% for `tomcatv`. DSI improves `EM3D`'s performance by 32% using states and 26% using version numbers. `Barnes` and `tomcatv` show very little change from the

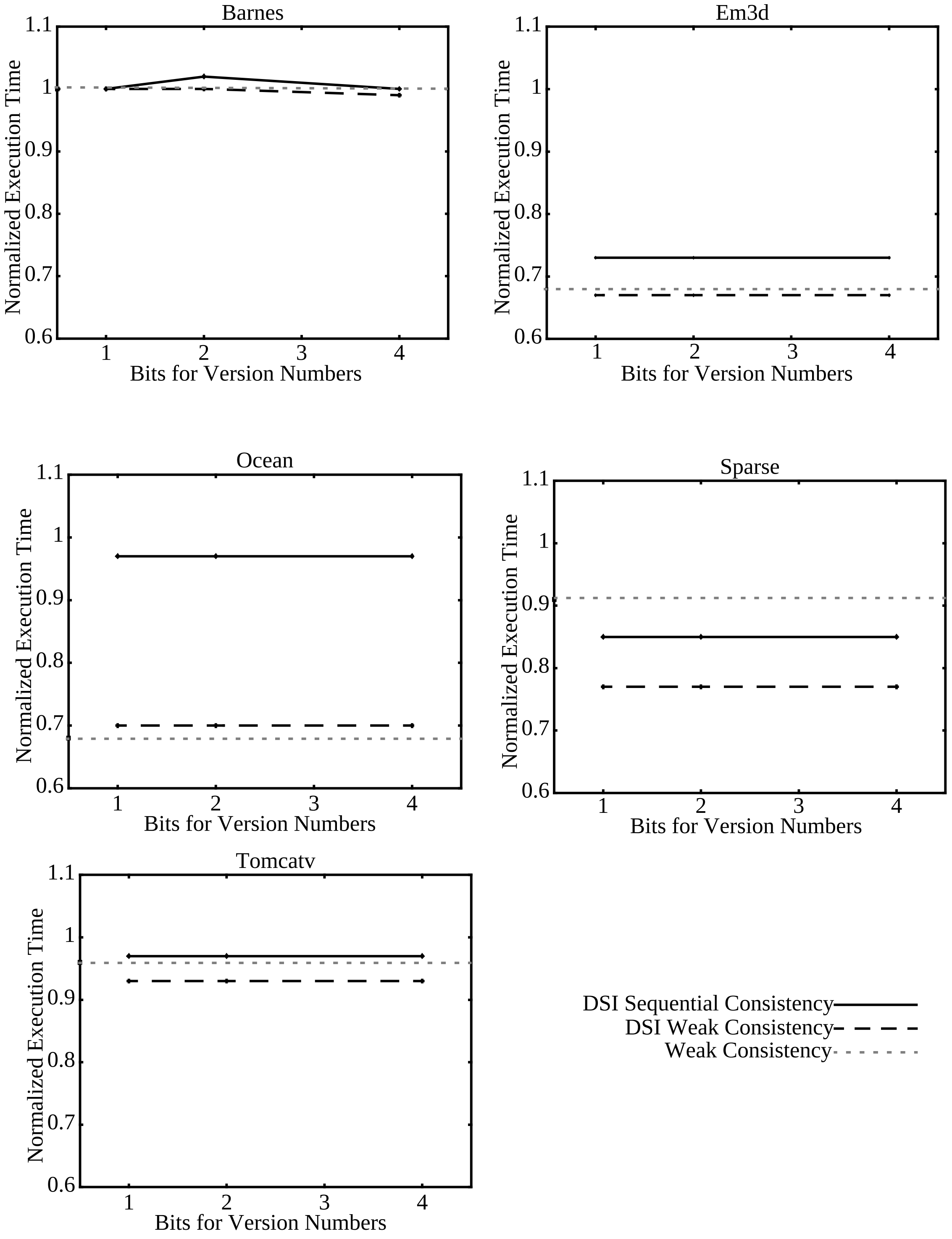
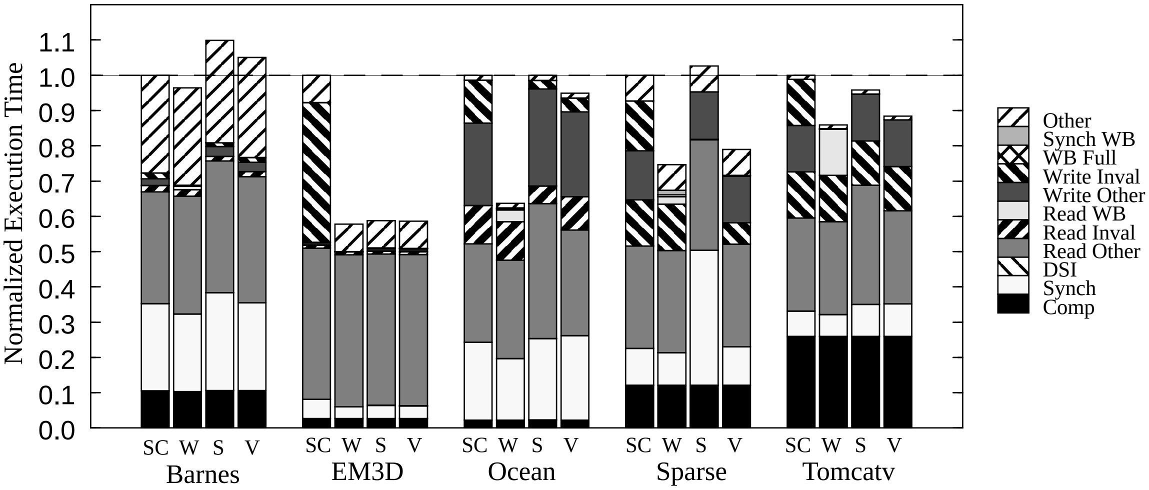


Figure 36: Performance Impact of Version Number Size



2M-byte cache, 1000 cycle network latency
SC = Sequential Consistency, W = Weak Consistency, S = DSI using additional states, V = DSI using version numbers

Figure 37: Impact of Network Latency

100 cycle network. DSI provides less benefit for `sparse` with the higher network latency, improving performance by only 2% using states and 9% using version numbers.

For a 2M-byte cache size (see Figure 37), DSI using version numbers reduces execution time by 41% for `EM3D`, 5% for `ocean`, 21% for `sparse`, and 12% for `tomcatv`, but increases `barnes` execution time by 5%. Using additional states to detect self-invalidate blocks improves `EM3D`'s execution time by 41% and `tomcatv`'s by only 4%. `Ocean`'s execution time is unaffected, while this method increases the execution time of `barnes` by 10% and `sparse` by 3%. Thus, version numbers generally perform better than additional states.

The results in this section show that DSI can improve the performance of a sequentially consistent, full-map directory-based protocol by eliminating invalidation latencies. For all but one of my benchmarks, DSI achieves performance comparable to an implementation of weak consistency. The benefit of DSI is most pronounced when coherence activity dominates communication. When a program's data set does not fit in the cache, coherence overhead is low, and the benefit of DSI decreases. These results suggest that systems using main memory as a cache for remote data, e.g., COMA, [28,61] may benefit significantly from self-invalidation.

4.5.2.4 Self-Invalidation Mechanisms

In this section I evaluate the FIFO and selective flush techniques for self-invalidating blocks from the cache. The FIFO buffer has 64 entries and is flushed at each synchroniza-

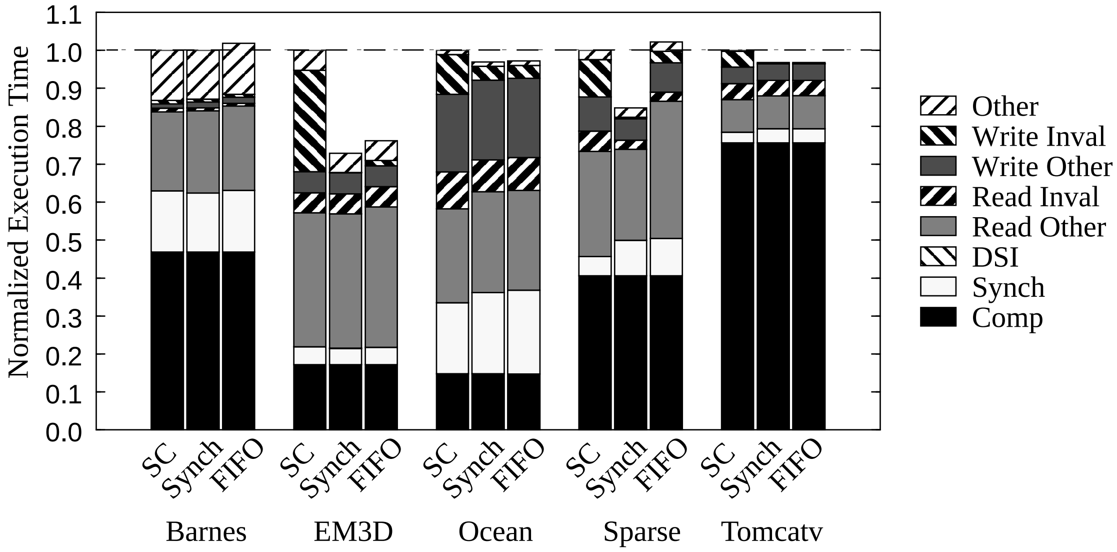


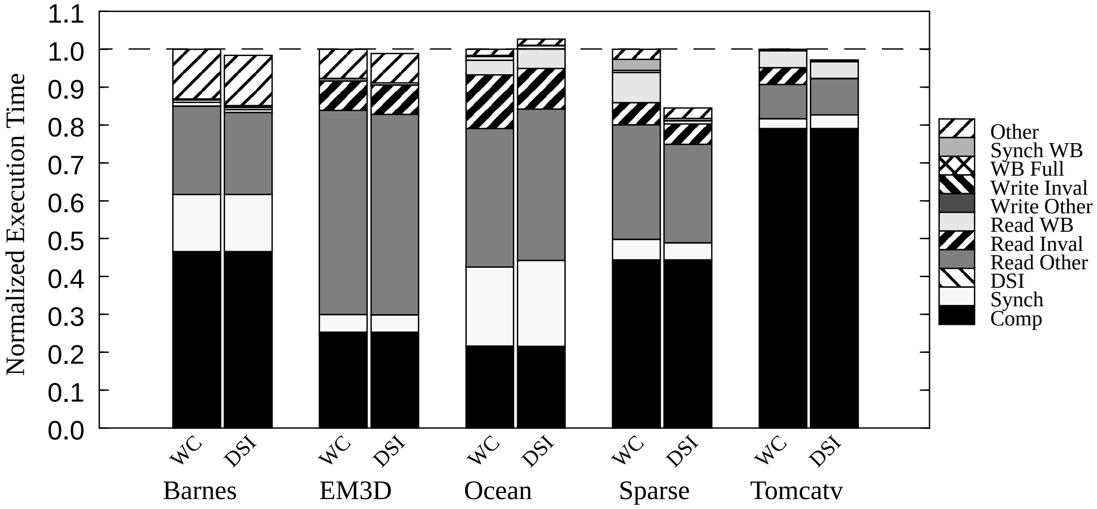
Figure 38: Self-Invalidation Mechanisms

tion operation. The selective flush hardware uses a linked list to invalidate marked blocks at each synchronization point. The directory uses version numbers to determine which blocks should be self-invalidated.

The results, shown in Figure 38, are for a 2M-byte cache; however, there is no qualitative difference for a 256K-byte cache. For EM3D, ocean, and tomcatv there is little difference between the self-invalidation schemes. However, self-invalidating at synchronization operations outperforms the FIFO for barnes and sparse, which exhibits a dramatic difference. The FIFO is unable to contain all of the self-invalidate blocks in the program's working set. Blocks are self-invalidated too early, causing an additional subsequent miss which obtains a normal cache block. This is a fundamental problem with a finite size buffer, and can significantly undermine the benefit of DSI.

4.5.3 DSI and Weak Consistency

This section evaluates the benefit of DSI in the context of weak consistency. DSI and weak consistency both reduce the impact of coherence overhead, and the results in Section 4.5.2 show they often achieve comparable reductions in execution time. When DSI is used in conjunction with weak consistency, the directory can utilize tear-off blocks, as described in Section 4.3.3, to eliminate acknowledgment messages. Furthermore, the write buffer can mitigate the effects of self-invalidating exclusive blocks incorrectly, and I can eliminate the special case for exclusive blocks, described in Section 4.4.1.



2M-byte cache, 100 cycle network latency, DSI with version numbers

Figure 39: DSI Performance Under Weak Consistency

Benchmark	100 cycle network		1000 cycle network	
	256 KB	2 MB	256 KB	2 MB
Barnes	1.00	0.98	1.00	1.03
EM3D	0.99	0.99	1.00	1.00
Ocean	1.00	1.02	0.99	1.04
Sparse	0.82	0.84	0.90	0.96
Tomcatv	1.00	0.97	1.00	0.86

Table 9: Weakly Consistent DSI Normalized Execution Time

For most of my programs, there is very little effect on execution time, as shown in Table 9 and Figure 39. `Sparse` is the exception, where DSI with weak consistency improves performance by up to 18% over weak consistency alone. `Tomcatv` shows a 14% reduction in execution time for the 2M-byte cache with a 1000 cycle network. This is a direct consequence of eliminating the special case for exclusive blocks; DSI eliminates both write invalidation and read invalidation latencies.

DSI with tear-off blocks eliminates both invalidation and acknowledgment messages. Tear-off blocks potentially reduce both the total message traffic and the directory controller occupancy. The latter may have a significant effect on systems that cannot process local memory accesses in parallel with protocol events (e.g., FLASH [40], Blizzard [63])

Benchmark	Reduction of Total Messages		Reduction of Invalidation Messages		Increase in Remote Misses	
	256 KB	2 MB	256 KB	2 MB	256 KB	2 MB
Barnes	0%	5%	37%	41%	2%	-1%
EM3D	17%	26%	85%	100%	0%	0%
Ocean	4%	12%	32%	52%	1%	9%
Sparse	7%	1%	54%	66%	-1%	1%
Tomcatv	0%	21%	45%	100%	1%	4%

Table 10: DSI Message Reduction Under Weak Consistency

or on systems that experience contention in the network (my simulations do not model contention at the network switches). The results in Table 10 show that DSI reduces the total number of messages by up to 17% for a 256K-byte cache and 26% for a 2M-byte cache. To the first order, directory controller occupancy will be reduced by the same amount. Table 10 also shows that DSI eliminates between 32% and 85% of invalidation messages for the 256K-byte cache and between 41% and 100% for the 2M-byte cache. DSI changes the replacement of blocks from the cache, potentially producing different executions of the program and reducing the number of remote misses, as shown for barnes and sparse. These results provide an estimate of the accuracy of my DSI implementation. DSI dramatically reduces the number of invalidation messages without significantly increasing the number of remote misses.

4.5.4 Effect of Larger Cache Block Size

The previous performance studies all use 32-byte cache blocks. I now examine the effect of larger block sizes on the performance of DSI. I performed simulations of a 2M-byte cache with 64-byte and 128-byte blocks. Figure 40 shows the execution time of the various protocols, normalized to the base sequentially consistent protocol with 32-byte blocks. These results reveal that increasing the block size reduces the benefit of DSI. EM3D is the only program that benefits significantly from DSI when using 128-byte blocks. For most of my benchmarks, the base sequentially consistent protocol improves significantly with larger cache blocks. Barnes is the exception, exhibiting a significant decrease in performance for larger block sizes due to false sharing. For the other benchmarks, the larger block size requires fewer invalidation messages to transfer the necessary data. Therefore, there is less invalidation delay, decreasing the potential advantage of DSI for the systems that I studied.

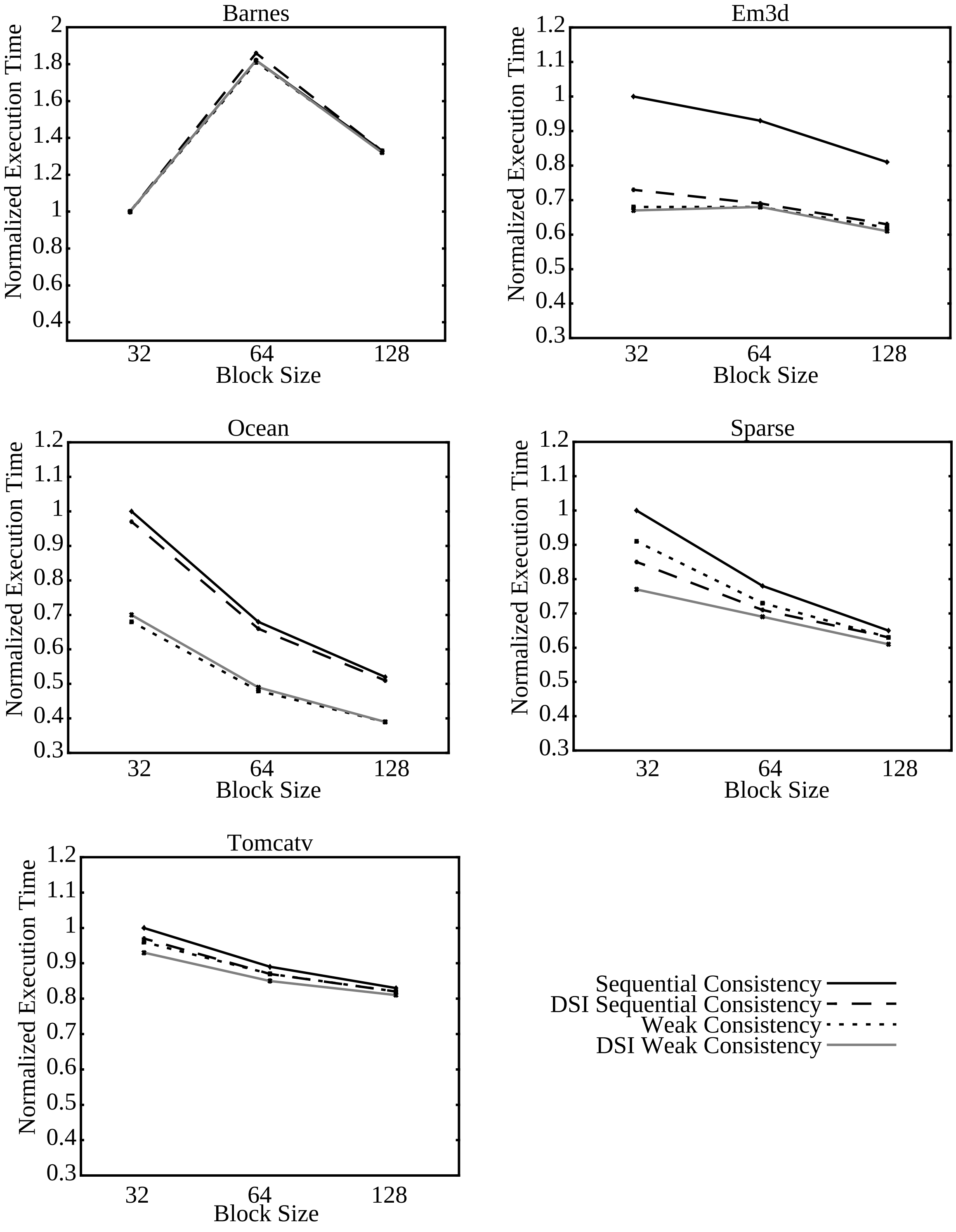


Figure 40: Effect of Block Size on DSI

4.6 Conclusion

Coherence overhead in directory-based, write-invalidate protocols can significantly degrade performance. In this chapter, I evaluated dynamic self-invalidation (DSI), a new technique for reducing coherence overhead. DSI eliminates invalidation messages by having processors automatically invalidate local copies of cache blocks before another processor accesses the block. Therefore, the directory can immediately respond with the data when processing a request for the block.

I evaluated DSI in the context of a full-map, hardware cache coherence protocol. In my implementations, the directory identifies cache blocks for self-invalidation, and the cache controller performs the self-invalidation. Under sequential consistency—where the latency of invalidating outstanding copies may lie on a program's critical path—DSI reduces execution by up to 41%, depending on the cache size and network latency. Under weak consistency, DSI generally had little effect on execution time, although one benchmark improved by 18%. However, combining DSI and weak consistency permits exploitation of *tear-off blocks*, where the directory does not track the outstanding copies. This eliminates both invalidation and acknowledgment messages, reducing the total number of messages by up to 26%.

I presented two techniques for the directory to identify which blocks should be self-invalidated: additional states and version numbers. My simulations reveal that version numbers generally outperform additional states. I also evaluated two approaches for the cache controller to perform the self-invalidation: a FIFO buffer, and at synchronization operations using custom hardware. Self-invalidation at synchronization operations utilizes the full capacity of the cache, and significantly outperforms the finite-size FIFO for some applications.

DSI is a general technique, applicable to hardware, software, and hybrid cache coherent shared-memory multiprocessors. Current trends in parallel architectures, e.g., faster processors and larger caches, can make coherence overhead a significant fraction of execution time. If this trend continues, DSI should be of increasing benefit.

Chapter 5

Conclusion

The memory system performance of modern microprocessors has a significant impact on sustained execution rates. As processor cycle times decrease, memory system performance becomes ever more critical to overall performance. Market forces prevent the use of expensive supercomputer-style memory systems in microprocessor-based machines. Instead, computer architects utilize cache memories to develop cost-effective memory systems.

Continually changing technology and workloads create a moving target for computer architects in their effort to design cost-effective memory systems. Meeting the demands of ever changing workloads and technology requires the following:

- Efficient techniques for evaluating memory system performance,
- Tuning program’s to better use the memory system, and
- New memory system designs.

This thesis makes contributions in each of these areas.

5.1 Thesis Summary

Simulation is the most common technique for evaluating memory system performance. As technology and workloads change, the ability to rapidly simulate memory systems will become increasingly important to hardware and software designers. Hardware designers rely on simulation to evaluate new memory systems. Software designers can use simulation to obtain information about their program’s memory system behavior.

To meet the demands of ever changing technology and workloads, simulation techniques must change. The conventional method for memory system simulation relies on the reference trace abstraction. Unfortunately, this simple abstraction limits simulator performance because the simulator must examine each memory reference. In many simulations, most memory references do not change the state of the memory system and require no action by the simulator.

This thesis evaluates a new approach for on-the-fly simulation of memory systems—the *active memory* abstraction. The primary contribution of active memory is the ability to optimize memory system simulation for the common case—references that do not change the state of the simulated memory system. Active memory associates a state with each memory block, specifying a function to be invoked when the block is accessed. Simulation is controlled by having the simulator manipulate the memory block states. A predefined NULL function can be optimized by active memory implementations, allowing efficient execution of the common no-action case.

My implementation of the active memory abstraction, Fast-Cache, inserts 9 instructions before each memory reference, to quickly determine whether a simulator action is required. Fast-Cache simulation times are only 2 to 6 times slower than the original, uninstrumented program on a SPARCstation 10/51; a procedure call based trace-driven simulator is 7 to 16 times slower than the original program, and a trace-driven simulator that buffers the address trace is 3 to 8 times slower. Furthermore, the system features required for trap-driven simulation are not always available, and if they are available, they can be used to implement the active memory abstraction.

As the impact of memory system performance increases, the ability to rapidly evaluate memory system performance will become more important. Hardware and software designers will increasingly rely on simulation to evaluate new ideas. In particular, programmer's must be aware of their program's cache behavior, and restructure their code to better utilize the memory hierarchy. The size and complexity of today's programs precludes a simple mental simulation of cache behavior. Instead, the programmer requires a cache profile that provides insight for selecting program transformations that will improve performance.

In this thesis, I show how to use CProf, a cache profiler, to select appropriate program transformations from a set of well-known transformations. CProf provides cache performance information at the source line and data structure level allowing a programmer to identify hot spots. The insight CProf provides, by classifying cache misses as compulsory, capacity, and conflict, helps programmers select appropriate program transformations that improve a program's spatial or temporal locality, and thus overall performance. I used CProf to profile and tune six of the SPEC92 benchmarks. Execution time speedups for these programs range from 1.02 to 3.46, depending on the machine's memory system.

The third contribution of this thesis is *dynamic self-invalidation* (DSI), a new memory system design. DSI is a technique for reducing coherence overhead in shared-memory multiprocessors. The fundamental concept of DSI is for processors to automatically replace a block from their cache before another processor wants to access the block. This allows the directory to immediately respond with the data when processing a request for the block, reducing invalidation latency and message overhead.

Invalidation latency is particularly important under sequential consistency, since it may lie on the program's critical path. Using DSI produces execution times comparable to an implementation of weak consistency, reducing execution by up to 41%, depending on the cache size and network latency. When used with weak consistency, improvements in execution time were less dramatic, however one program did exhibit an 18% improvement. Under weak consistency, the DSI protocol can exploit *tear-off* blocks, which allow a processor to cache a copy of the block without recording this information at the directory. Tear-off blocks eliminate the need to acknowledge when a block is invalidated from a processors cache, reducing the number of messages by up to 26%.

Although I focussed on all hardware implementations that avoid modifications to the processor chip, DSI can be applied to a variety of cache coherent shared-memory multiprocessors (e.g., COMA, hardware, software, and hybrid). Furthermore, the benefit of DSI should increase as processors get faster, and caches get larger.

5.2 What Next?

The work presented in this thesis can be extended in many ways. First, as detailed in Section 2.7, the active memory abstraction can be extended to include accurate cycle counts for timing dependent simulations (e.g., prefetching and lock-up free caches). Another avenue for future investigation is a multi-program workload simulation environment, including the operating system. One interesting approach is to utilize a single 64-bit address space to simulate an entire system. In this case, the active memory abstraction can be used to provide both efficient simulation and software fault isolation.

CProf currently profiles only user-level uniprocessor applications. Extending it to profile operating systems, multiprogramming, and multiprocessor systems would be useful for most of the computing community. Many of these extensions would come for free, if the active memory abstraction were extended to facilitate these paradigms.

There are many possible extensions to the DSI studies. First, is the use of DSI in the context of cache only memory architectures (COMA). These systems utilize main memory as a cache for remote data, and coherence traffic is likely to dominate communication, increasing the potential benefit of DSI. Coherence traffic is also a potential problem when placing multiple processors on a single chip. It would be interesting to evaluate DSI as a means for alleviating this situation.

Another interesting study would examine the effectiveness of tear-off blocks at reducing false-sharing. In particular, an investigation of a combination of loosely coherent memory [44] and DSI in a distributed shared-memory system. The final possible study is a comparison between DSI and program annotations (e.g., check_in), and the possible combination of the two (e.g,. check_out_tearoff).

Bibliography

[1] Sarita V. Adve and Mark D. Hill. Implementing Sequential Consistency In Cache-Based Systems. In *Proceedings of the 1990 International Conference on Parallel Processing (Vol. I Architecture)*, pages I47–I50, August 1990.

[2] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[3] Anant Agarwal, Richard Simoni, Mark Horowitz, and John Hennessy. An Evaluation of Directory Schemes for Cache Coherence. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 280–289, 1988.

[4] Thomas E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. II Software)*, pages II170–II174, August 1989.

[5] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[6] 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.

[7] Brian Case. SPARC V9 Adds Wealth of New Features. *Microprocessor Report*, 7(9), February 1993.

[8] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[9] David Chaiken, John Kubiatowics, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[10] Hoichi Cheong and Alexander V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. *IEEE Computer*, 23(6):39–48, June 1990.

[11] Trishul M. Chilimbi and James R. Larus. Cachier: A Tool for Automatically Inserting CICO Annotations. In *Proceedings of the 1994 International Conference on Parallel Processing (Vol. II Software)*, pages II–89–98, August 1994.

[12] Lynn Choi and Pen-Chung Yew. A Compiler-Directed Cache Coherence Scheme with Improved Intertask Locality. In *Proceedings of Supercomputing 94*, pages 773–782, November 1994.

[13] 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.

[14] Alan L. Cox and Robert J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[15] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, November 1993.

[16] Ron Cytron, Steve Karlovsky, and Kevin P. McAuliffe. Automatic Management of Programmable Caches. In *Proceedings of the 1988 International Conference on Parallel Processing (Vol. II Software)*, pages 229–238, August 1988.

[17] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Combined Performance Gains of Simple Cache Protocol Extensions. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 187–197, April 1994.

[18] Ervan Darnell and Ken Kennedy. Cache Coherence Using Local Knowledge. In *Proceedings of Supercomputing 93*, pages 720–729, November 1993.

[19] 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.

[20] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[21] John H. Edmondson, Paul Rubinfeld, Ronald Preston, and Vidya Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.

[22] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–213, November 1994.

[23] 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.

[24] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[25] Aaron J. Goldberg and John L. Hennessy. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[26] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. *ACM SIGPLAN Notices*, 17(6):120–126, June 1982. Proceedings of the SIGPLAN ’82 Conference on Programming Language Design and Implementation.

[27] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[28] E. Hagersten, A. Landin, and S. Haridi. DDM—A Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[29] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V, October. 1992.

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

[31] Peter Yan-Tek Hsu. Designing the TFP Microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[32] Cray Research Inc. Cray Research Unveils Powerful Cray T90 High-End Product Line World’s First Wireless Supercomputer Systems, February 1995. URL = http://www.cray.com/PUBLIC/WHATS_NEW/PRODUCTS/T90_announce.html.

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

[34] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[35] 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.

[36] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwanenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operations Systems. Technical Report COMP TR93-214, Department of Computer Science, Rice University, November 1993.

[37] Richard E. Kessler and Mark D. Hill. Page Placement Algorithms for Large Real-Index Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992.

[38] 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.

[39] Gordon Kurpanek, Ken Chan, Jason Zheng, Eric Delano, and William Bryg. PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface. In *Compcon*, pages 375–382, 1994.

[40] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[41] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74, April 1991.

[42] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

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

[44] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, 1994.

[45] 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.

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

[47] 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.

[48] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, June 1995.

[49] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Sang Lyul Min and Jean-Loup Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):25–44, January 1992.

[55] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[56] D. N. Pnevmatikatos and M. D. Hill. Cache Performance of the Integer SPEC Benchmarks on a RISC. *ACM SIGARCH Computer Architecture News*, 18(2):53–68, June 1990.

[57] 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.

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

[59] 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.

[60] 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.

[61] 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.

[62] Christoph Ernst Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989. Also available as technical report No. CENG 89-19.

[63] 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.

[64] A. Shatdal, C. Kant, and J. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, September 1994.

[65] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

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

[67] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

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

[69] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proceedings of the Int. Soc. for Opt. Engr*, pages 241–248, 1982.

[70] SPEC. SPEC Newsletter, December 1991.

[71] 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.

[72] Per Stenstrom, Mats Brorsson, and Lars Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[73] 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.

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

[75] C. K. Tang. Cache System Design in the Tightly Coupled Multiprocessor System. In *Proc. AFIPS*, pages 749–753, 1976.

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

[77] 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.

[78] 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.

[79] 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.

[80] Neil Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design A Systems Perspective*. Addison-Wesley, 1988.

[81] 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.

[82] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.

[83] 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.

[84] Benjamin Zorn and Paul N. Hilfinger. A Memory Allocation Profiler for C and LISP. Technical Report UCB/CSD 88/404, Computer Science Division (EECS), University of California at Berkeley, February 1988.

Appendix A: Simulator Implementation Details

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 procesor. 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, I use 0x0 as a place holder for immediates. Instructions for computing the effective address are shown using immediates, they may change to register + register addressing when the snippet is inserted.

I 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 inline 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 I 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, I 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 I 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
```

Appendix B: Dynamic Self-Invalidation Protocols

This appendix describes, in detail, the DSI protocols for detecting blocks for self-invalidation. There are numerous extensions one could make to the base $\mathrm{Dir}_n\mathrm{NB}$ protocol. Unfortunately, the design space is too large to fully investigate, and one can spend an endless amount of time tuning the performance of existing implementations. Nonetheless, examining a specific protocol in more detail reveals the intricacies involved in detecting blocks for self-invalidation. In this appendix I use RO to indicate the shared (S) state and RW to indicate the exclusive (EX) state.

Additional States

For shared-readable blocks, two additional states are used to detect that a block has been modified. One state, IDLE_SI, indicates the block has gone from RW to IDLE, a block enters this state whenever the exclusive block is replaced or self-invalidated. The second additional state, RO_SI, indicates there are already outstanding blocks that will be self-invalidated. When servicing a request for a shared-readable block, if the current state is RW, IDLE_SI, or RO_SI, the directory responds with a block that should be self-invalidated.

We also use the IDLE_SI state to indicate that a shared-readable block identified for self-invalidation was replaced from the cache. A subsequent miss by the processor is the result of incorrect self-invalidation, hence we distinguish between replacements and self-invalidations. Self-invalidations place the block to in the RO or IDLE state, depending on the number of outstanding copies, preventing subsequent requests for shared-readable blocks from being self-invalidated. In contrast, cache replacements keep the block in the RO_SI state if there are other outstanding copies, or cause a transition to the IDLE_SI state.

Exclusive blocks should be self-invalidated if it has been accessed by a different processor since the last access by the writing processor. This is easily identified when there are outstanding shared-readable copies, since the state of the block is RO or RO_SI. However, a processor may obtain a shared-readable copy of the block which is self-invalidated or replaced from the cache. This may cause the block to enter the IDLE state. To identify this situation, we add another state, IDLE_R. A similar situation can arise when an exclusive block is self-invalidated or replaced, placing the block in the IDLE state. Therefore, another state, IDLE_W, is added to differentiate between blocks that become idle because they were shared-readable and those that were RW. Self-invalidations cause transitions into IDLE_R or IDLE_W, for shared-readable and exclusive blocks respectively. Cache replacements still place the block in IDLE_SI.

The directory identifies exclusive blocks for self-invalidation if the current state is RO, RO_SI, IDLE_R, or IDLE_SI, or IDLE_W and a different processor was the last writer.

Shared-readable blocks are identified for self-invalidation when the state is IDLE_W, IDLE_SI, RW, or IDLE_SI.

If tear-off blocks are used, the directory does not send invalidations if all the outstanding shared-readable copies are tear-off blocks. This allows us to eliminate the RO_SI state, instead the block can be placed directly in the IDLE_SI or IDLE_R state, depending on whether exclusive blocks are also self-invalidated. We also add one bit to each directory entry to indicate there are more than one outstanding tear-off blocks. The first tear-off block causes a transition into the state IDLE_R, subsequent tear-off blocks set the bit. This bit is used when processing requests for exclusive blocks that result from write misses to tear-off blocks. In this scenario, the current state of the block is IDLE_R, but the requesting processor may be the only processor with a copy of the block. In this case, the bit is zero and the block is not self-invalidated, if more than one processor has a tear-off block, then the exclusive copy is identified for self-invalidation. Rather than add an extra bit, we could have encoded this information in the state of the block by separating IDLE_R into IDLE_R1, and IDLE_R2.

Table 11 shows the detailed specification of the DSI protocol that uses states to identify blocks for self-invalidation when tracked blocks are used. The columns are message type, current state of the block, the output of a counter for determining when all acknowledgments have been received (this is a check for zero), a test to see if the requesting processor already had a copy of the block, a test to determine if the requesting processor is the home node, a list of actions taken, and the new state of the block. In the table an * indicates don't care, and - indicates no change in the state of the block, however some actions result in state changes. Events not listed in the table are protocol violations, and they result in a system panic. The end of this appendix includes source code for the various actions.

Message	State	Z	B	Self	Actions	New State
GET_RW	*	*	*	*	Nack	-
GET_RW	IDLE	*	*	*	Send, Set_Ptr	RW
GET_RW	IDLE_R	*	*	E	Send, Set_Ptr	RW
GET_RW	IDLE_R	*	*	NE	SI_Send, Set_Ptr	RW
GET_RW	IDLE_W	*	*	E	Send, Set_Ptr	RW
GET_RW	IDLE_W	*	*	NE	Ck_Ptr, Set_Ptr	RW
GET_RW	IDLE_SI	*	*	E	Send, Set_Ptr	RW
GET_RW	IDLE_SI	*	*	NE	SI_Send, Set_Ptr	RW
GET_RW	RO	*	*	*	Inval_RO, Set_Ptr	RW_GET

Table 11: DSI Protocol Using States and Tracked Blocks

Message	State	Z	B	Self	Actions	New State
GET_RW	RO_SI	*	*	*	Inval_RO, Set_Ptr	RW_GET
GET_RW	RW	*	*	*	Inval_RW, Set_Ptr	RW_GET
GET_RO	*	*	*	*	Nack	-
GET_RO	IDLE	*	*	*	Send, Set_Bit	RO
GET_RO	RO	*	*	*	Send, Set_Bit	-
GET_RO	RW	*	*	*	Inval_RW, Set_Ptr	RO_GET
GET_RO	IDLE_W	*	*	E	Zero, Send, Set_Bit	RO_SI
GET_RO	IDLE_W	*	*	NE	Zero, SI_Send, Set_Bit	RO_SI
GET_RO	IDLE_R	*	*	*	Send, Set_Bit	RO
GET_RO	IDLE_SI	*	*	E	Zero, Send,Set_Bit	RO_SI
GET_RO	IDLE_SI	*	*	NE	Zero, SI_Send, Set_Bit	RO_SI
GET_RO	RO_SI	*	*	E	Send,Set_Bit	-
GET_RO	RO_SI	*	*	NE	SI_Send,Set_Bit	-
PUT_DATA	RW	*	*	*	IDLE_SI	
PUT_DATA	SWAP_RW	*	*	*	Swap_Ptr	IDLE_W
PUT_DATA	RO_GET	*	*	E	Forward_RO	RO_SI
PUT_DATA	RO_GET	*	*	NE	SI_Forward_RO	RO_SI
PUT_DATA	RW_GET	*	*	*	SI_Forward_RW	RW
SI_PUT_DATA	RW	*	*	*	IDLE_W	
SI_PUT_DATA	SWAP_RW	*	*	*	Swap_Ptr	IDLE_W
SI_PUT_DATA	RO_GET	*	*	E	Forward_RO	RO_SI
SI_PUT_DATA	RO_GET	*	*	NE	SI_Forward_RO	RO_SI
SI_PUT_DATA	RW_GET	*	*	*	SI_Forward_RW	RW
PUT_NO_DATA	RO	*	*	*	Clr_Bit_to_IRO	-
PUT_NO_DATA	RO_SI	*	*	*	Clr_Bit_to_SI	-
PUT_NO_DATA	RW_GET	NZ	*	*	Decr	-

Table 11: DSI Protocol Using States and Tracked Blocks

Message	State	Z	B	Self	Actions	New State
PUT_NO_DATA	RW_GET	Z	*	*	SI_Forward_RW	RW
PUT_NO_DATA	SWAP_RO	NZ	*	*	Decr	-
PUT_NO_DATA	SWAP_RO	Z	*	*	Swap_Ptr	IDLE_SI
SI_PUT_NO_DATA	RO	*	*	*	Clr_Bit_to_IRO	-
SI_PUT_NO_DATA	RO_SI	*	*	*	Clr_Bit_to_IRO	-
SI_PUT_NO_DATA	RW_GET	NZ	*	*	Decr	-
SI_PUT_NO_DATA	RW_GET	Z	*	*	SI_Forward_RW	RW
SI_PUT_NO_DATA	SWAP_RO	NZ	*	*	Decr	-
SI_PUT_NO_DATA	SWAP_RO	Z	*	*	Swap_Ptr	IDLE_SI
SWAP	*	*	*	*	Swap_Nack	-
SWAP	IDLE	*	*	*	Swap	-
SWAP	IDLE_SI	*	*	*	Swap	-
SWAP	IDLE_R	*	*	*	Swap	-
SWAP	IDLE_W	*	*	*	Swap	-
SWAP	RO	*	S	*	Inval_RO, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RO	*	C	*	Inval_RO, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RO_SI	*	C	*	Inval_RO, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RW	*	*	*	Inval_RW, Save_Swap, Set_Ptr	SWAP_RW
UPGRADE	RO	*	*	*	Clr_Bit, RW_Upgrade_RO	-
UPGRADE	RO_SI	*	*	*	Clr_Bit, RW_Upgrade_RO	-
UPGRADE	RW_GET	NZ	*	*	Nack, Decr	-

Table 11: DSI Protocol Using States and Tracked Blocks

Message	State	Z	B	Self	Actions	New State
UPGRADE	RW_GET	Z	*	*	Nack, Forward_RW	RW
UPGRADE	SWAP_RO	NZ	*	*	Nack, Decr	-
UPGRADE	SWAP_RO	Z	*	*	Swap_Ptr, Send,Set_Ptr	RW

Table 11: DSI Protocol Using States and Tracked Blocks

Table 12 specifies the protocol for using states to detect blocks and using tear-off blocks under weak consistency.

Message	State	Z	Self	Actions	New State
GET_RW	*	*	*	Nack	-
GET_RW	IDLE	*	*	Send_RW_Ack, Set_Ptr, Clr_Read	RW
GET_RW	IDLE_R	*	E	Send_RW_Ack, Set_Ptr, Clr_Read	RW
GET_RW	IDLE_R	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read	RW
GET_RW	IDLE_W	*	E	Send_RW_Ack, Set_Ptr, Clr_Read	RW
GET_RW	IDLE_W	*	*	Ck_Ptr, Set_Ptr, Clr_Read	RW
GET_RW	IDLE_SI	*	E	Send_RW_Ack, Set_Ptr, Clr_Read	RW
GET_RW	IDLE_SI	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read	RW
GET_RW	RO	*	E	Inval_RO, Send, Set_Ptr, Clr_Read	GET_ACKS
GET_RW	RO	*	NE	Inval_RO, SI_Send, Set_Ptr, Clr_Read	GET_ACKS
GET_RW	RO_SI	*	E	Inval_RO, Send, Set_Ptr, Clr_Read	GET_ACKS

Table 12: DSI Protocol Using States and Tear-Off Blocks

Message	State	Z	Self	Actions	New State
GET_RW	RO_SI	*	NE	Inval_RO, SI_Send, Set_Ptr, Clr_Read	GET_ACKS
GET_RW	RW	*	*	Inval_RW, Set_Ptr, Clr_Read	RW_GET
GET_RO	*	*	*	Nack	-
GET_RO	IDLE	*	*	Send, Zero, Set_Bit, Set_Read	RO
GET_RO	IDLE_W	*	E	Zero, Send, Set_Bit, Set_Read	RO_SI
GET_RO	IDLE_W	*	NE	Zero, SI_Send, Set_Read	IDLE_SI
GET_RO	IDLE_R	*	*	Send, Zero, Set_Bit, Set_Read	RO
GET_RO	IDLE_SI	*	E	Send, Zero, Set_Bit, Set_Read	RO_SI
GET_RO	IDLE_SI	*	NE	SI_Send, Set_Read	-
GET_RO	IDLE_SI	*	*	Send, Zero, Set_Bit	RO
GET_RO	RO	*	*	Send, Set_Bit	-
GET_RO	RO_SI	*	E	Send, Set_Bit, Set_Read	-
GET_RO	RO_SI	*	NE	SI_Send, Set_Read	-
GET_RO	RW	*	*	Inval_RW, Set_Ptr, Set_Read	RO_GET
PUT_DATA	RW	*	*		IDLE_W
PUT_DATA	RO_GET	*	E	Forward_RO, Set_Read	RO
PUT_DATA	RO_GET	*	NE	SI_Forward_RO, Set_Read	IDLE_R
PUT_DATA	RW_GET	*	*	SI_Forward_RW_ACK	RW
PUT_DATA	GET_ACKS	*	*		IDLE_GET_ACKS
PUT_DATA	SWAP_RW	*	*	Swap_Ptr	IDLE

Table 12: DSI Protocol Using States and Tear-Off Blocks

Message	State	Z	Self	Actions	New State
SI_PUT_DATA	RW	*	*		IDLE_SI
SI_PUT_DATA	RO_GET	*	E	Forward_RO, Set_Read	RO
SI_PUT_DATA	RO_GET	*	NE	SI_Forward_RO, Set_Read	IDLE_R
SI_PUT_DATA	RW_GET	*	*	SI_Forward_RW_ACK	RW
SI_PUT_DATA	GET_ACKS	*	*		IDLE_GET_ACKS
PUT_NO_DATA	RO	*	*	Clr_Bit_to_IRO	-
PUT_NO_DATA	RO_SI	*	*	Clr_Bit_to_SI	-
PUT_NO_DATA	GET_ACKS	Z	*	Ack_RW	RW
PUT_NO_DATA	GET_ACKS	NZ	*	Decr	-
PUT_NO_DATA	IDLE_GET_ACKS	Z	*	Zero, Ack_RW	IDLE_W
PUT_NO_DATA	IDLE_GET_ACKS	NZ	*	Decr	-
PUT_NO_DATA	RW_GET	*	*	Panic	-
PUT_NO_DATA	SWAP_RO	Z	*	Swap_Ptr	IDLE_W
PUT_NO_DATA	SWAP_RO	NZ	*	Decr	-
SWAP	*	*	*	Swap_Nack	-
SWAP	IDLE	*	*	Swap	-
SWAP	IDLE_R	*	*	Swap	-
SWAP	IDLE_W	*	*	Swap	-
SWAP	IDLE_SI	*	*	Swap	-
SWAP	RO	*	*	Inval_RO, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RO_SI	*	*	Inval_RO, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RW	*	*	Inval_RW, Save_Swap, Set_Ptr	SWAP_RW
UPGRADE	IDLE_W	*	*	Ck_Ptr, Set_Ptr, Clr_Read	RW

Table 12: DSI Protocol Using States and Tear-Off Blocks

Message	State	Z	Self	Actions	New State
UPGRADE	IDLE_R	*	E	Send_RW_Ack, Set_Ptr, Clr_Read	RW
UPGRADE	IDLE_R	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read	RW
UPGRADE	IDLE_SI	*	E	Send_RW_Ack, Set_Ptr, Clr_Read	RW
UPGRADE	IDLE_SI	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read	RW
UPGRADE	RW	*	*	Inval_RW, Zero, Set_Ptr, Clr_Read	RW_GET
UPGRADE	RO	*	*	RC_RW_Upgrade_RO, Clr_Read	-
UPGRADE	RO_SI	*	*	RC_RW_Upgrade_RO, Clr_Read	-
UPGRADE	GET_ACKS	NZ	*	Nack, Decr	-
UPGRADE	GET_ACKS	Z	*	Nack, Ack_RW	RW
UPGRADE	IDLE_GET_ACKS	NZ	*	Nack, Decr	-
UPGRADE	IDLE_GET_ACKS	Z	*	Nack, Zero, Ack_RW	IDLE
UPGRADE	RW_GET	NZ	*	Nack, Decr	-
UPGRADE	RW_GET	Z	*	Nack, Forward_RW	RW
UPGRADE	SWAP_RO	NZ	*	Nack, Decr	-
UPGRADE	SWAP_RO	Z	*	Swap_Ptr, Send, Set_Ptr	RW

Table 12: DSI Protocol Using States and Tear-Off Blocks

Version Numbers

Table 13 specifies the protocol for detecting blocks using version numbers. The column

Message	State	Z	B	V# ==	Actions	New State
GET_RW	*	*	*	*	Nack	-

Table 13: DSI Protocol Using Version Numbers and Tracked Blocks

Message	State	Z	B	V# ==	Actions	New State
GET_RW	IDLE	*	*	E	Send, Set_Ptr, Clr_Read, IncV	RW
GET_RW	IDLE	*	*	NE	SI_Send, Set_Ptr, Clr_Read, IncV	RW
GET_RW	RO	*	*	*	Inval_RO, Set_Ptr, Clr_Read, IncV	RW_GET
GET_RW	RW	*	*	*	Inval_RW, Set_Ptr, Clr_Read, IncV	RW_GET
GET_RO	*	*	*	*	Nack	-
GET_RO	IDLE	*	*	E	Send, Set_Bit, Set_Read	RO
GET_RO	IDLE	*	*	NE	SI_Send, Set_Bit, Set_Read	RO
GET_RO	RO	*	*	E	Send, Set_Bit, Set_Read	-
GET_RO	RO	*	*	NE	SI_Send, Set_Bit, Set_Read	-
GET_RO	RW	*	*	*	Inval_RW, Set_Ptr, Set_Read	RO_GET
PUT_DATA	RW	*	*	*	Zero	IDLE
PUT_DATA	RW_GET	*	*	*	Forward_RW	RW
PUT_DATA	SWAP_RW	*	*	*	Swap_Ptr	IDLE
PUT_DATA	RO_GET	*	*	E	Forward_RO, Set_Read	RO
PUT_DATA	RO_GET	*	*	NE	SI_Forward_RO, Set_Read	RO
PUT_NO_DATA	RO	*	*	*	Clr_Bit	-
PUT_NO_DATA	RW_GET	NZ	*	*	Decr	-
PUT_NO_DATA	RW_GET	Z	*	*	SI_Forward_RW, Clr_Read	RW
PUT_NO_DATA	SWAP_RO	NZ	*	*	Decr	-
PUT_NO_DATA	SWAP_RO	Z	*	*	Swap_Ptr	IDLE

Table 13: DSI Protocol Using Version Numbers and Tracked Blocks

Message	State	Z	B	V# ==	Actions	New State
SWAP	*	*	*	*	Swap_Nack	-
SWAP	IDLE	*	*	*	Swap, Set_Read, IncV	-
SWAP	RO	*	*	*	Inval_RO, Set_Read, IncV, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RW	*	*	*	Inval_RW, Set_Read, IncV, Save_Swap, Set_Ptr	SWAP_RW
UPGRADE	RW_GET	NZ	*	*	Nack, Decr	-
UPGRADE	RW_GET	Z	*	*	Nack, Forward_RW	RW
UPGRADE	SWAP_RO	NZ	*	*	Nack, Decr	-
UPGRADE	SWAP_RO	Z	*	*	Swap_Ptr, Send, Set_Ptr	RW
UPGRADE	RO	*	S	E	Clr_Bit, Upgrade_RO, Clr_Read, IncV	-
UPGRADE	RO	*	S	NE	Clr_Bit, RW_Upgrade_RO, Clr_Read, IncV	-
UPGRADE	RO	*	C	*	Panic	-

Table 13: DSI Protocol Using Version Numbers and Tracked Blocks

labeld V# == indicates the output of the version number comparison, it also includes the test of the home node equals the requesting node, and the test for the number of outstanding readable copies. Table 14 specifies the protocol using version numbers to detect blocks and the protocol utilizes tear-off blocks under weak consistency. Note the additional states required to handle the case where there are outstanding tear-off blocks.

Message	State	Z	V	Actions	New State
GET_RW	*	*	*	Nack	-

Table 14: DSI Protocol Using Version Numbers and Tear-Off Blocks

Message	State	Z	V	Actions	New State
GET_RW	IDLE	*	E	Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
GET_RW	IDLE	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
GET_RW	IDLE_ONE	*	*	SI_Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
GET_RW	IDLE_TWO	*	*	SI_Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
GET_RW	RO	*	*	Inval_RO, Send, Set_Ptr, Clr_Read, IncV	GET_ACKS
GET_RW	RO_SI	*	*	Inval_RO, SI_Send, Set_Ptr, Clr_Read, IncV	GET_ACKS
GET_RW	RW	*	*	Inval_RW, Set_Ptr, Clr_Read, IncV	RW_GET
GET_RO	*	*	*	Nack	-
GET_RO	IDLE	*	E	Send, Zero, Set_Bit, Set_Read	RO
GET_RO	IDLE	*	NE	SI_Send, Set_Read	IDLE_ONE
GET_RO	IDLE_ONE	*	E	Send, Zero, Set_Bit, Set_Read	RO_SI
GET_RO	IDLE_ONE	*	NE	SI_Send, Set_Read	IDLE_TWO
GET_RO	IDLE_TWO	*	E	Send, Zero, Set_Bit, Set_Read	RO_SI
GET_RO	IDLE_TWO	*	NE	SI_Send, Set_Read	-
GET_RO	RO	*	E	Send, Set_Bit, Set_Read	-
GET_RO	RO	*	NE	SI_Send, Set_Read	RO_SI

Table 14: DSI Protocol Using Version Numbers and Tear-Off Blocks

Message	State	Z	V	Actions	New State
GET_RO	RO_SI	*	E	Send, Set_Bit, Set_Read	-
GET_RO	RO_SI	*	NE	SI_Send, Set_Read	-
GET_RO	RW	*	*	Inval_RW, Set_Ptr, Set_Read	RO_GET
PUT_DATA	RO_GET	*	E	Forward_RO, Set_Read	RO
PUT_DATA	RO_GET	*	NE	SI_Forward_RO, Set_Read	IDLE_ONE
PUT_DATA	RW_GET	*	*	SI_Forward_RW_ACK	RW
PUT_DATA	GET_ACKS	*	*	-	IDLE_GET_ACKS
PUT_DATA	RW	*	*	Zero	IDLE
PUT_DATA	SWAP_RW	*	*	Swap_Ptr	IDLE
PUT_NO_DATA	RO	*	*	Clr_Bit	-
PUT_NO_DATA	RW_GET	NZ	*	Decr	-
PUT_NO_DATA	RW_GET	Z	*	Forward_RW	RW
PUT_NO_DATA	SWAP_RO	NZ	*	Decr	-
PUT_NO_DATA	SWAP_RO	Z	*	Swap_Ptr	IDLE
PUT_NO_DATA	RO_SI	*	*	RW_SI_Clr_Bit	-
PUT_NO_DATA	GET_ACKS	Z	*	Ack_RW	RW
PUT_NO_DATA	GET_ACKS	NZ	*	Decr	-
PUT_NO_DATA	IDLE_GET_ACKS	Z	*	Zero, Ack_RW	IDLE
PUT_NO_DATA	IDLE_GET_ACKS	NZ	*	Decr	-
SWAP	*	*	*	Swap_Nack	-
SWAP	IDLE	*	*	Swap, Set_Read, IncV	-
SWAP	IDLE_ONE	*	*	Swap, Set_Read, IncV	IDLE
SWAP	IDLE_TWO	*	*	Swap, Set_Read, IncV	IDLE
SWAP	RO	*	*	Inval_RO, Set_Read, IncV, Save_Swap, Set_Ptr	SWAP_RO

Table 14: DSI Protocol Using Version Numbers and Tear-Off Blocks

Message	State	Z	V	Actions	New State
SWAP	RO_SI	*	*	Inval_RO, Set_Read, IncV, Save_Swap, Set_Ptr	SWAP_RO
SWAP	RW	*	*	Inval_RW, Set_Read, IncV, Save_Swap, Set_Ptr	SWAP_RW
UPGRADE	IDLE	*	E	Ck_Ptr, Set_Ptr, Clr_Read, IncV	RW
UPGRADE	IDLE	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
UPGRADE	IDLE_W	*	E	Ck_Ptr, Set_Ptr, Clr_Read, IncV	RW
UPGRADE	IDLE_W	*	NE	SI_Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
UPGRADE	IDLE_ONE	*	*	Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
UPGRADE	IDLE_TWO	*	*	SI_Send_RW_Ack, Set_Ptr, Clr_Read, IncV	RW
UPGRADE	RO	*	*	RC_RW_Upgrade_RO, Clr_Read, IncV	-
UPGRADE	RO_SI	*	*	RC_RW_Upgrade_RO_SI, Clr_Read, IncV	-
UPGRADE	RW	*	*	Inval_RW, Set_Ptr, Clr_Read, IncV	RW_GET
UPGRADE	GET_ACKS	NZ	*	Nack, Decr	-
UPGRADE	GET_ACKS	Z	*	Nack, Ack_RW	RW
UPGRADE	IDLE_GET_ACKS	NZ	*	Nack, Decr	-
UPGRADE	IDLE_GET_ACKS	Z	*	Nack, Zero, Ack_RW	IDLE

Table 14: DSI Protocol Using Version Numbers and Tear-Off Blocks

Message	State	Z	V	Actions	New State
UPGRADE	RW_GET	NZ	*	Nack, Decr	-
UPGRADE	RW_GET	Z	*	Nack, Forward_RW	RW
UPGRADE	SWAP_RO	NZ	*	Nack, Decr	-
UPGRADE	SWAP_RO	Z	*	Swap_Ptr, Send, Set_Ptr	RW

Table 14: DSI Protocol Using Version Numbers and Tear-Off Blocks

Actions

Following is a complete list of actions required to support the above protocols. For the most part, these actions are very simple. However, some the actions perform complex computations for events that do not fit well into the table drive model. This is the source code from the directory module of my simulator.

```
/* Bad message received */
DO_ACTION(DA_Panic,
       fatal_error(“process_req: Protocol violation %x\n”,
                   aep->action);
       )
DO_ACTION(DA_Inval_RW,
       dp->inval_start = dir_srv.vt;
       send_put_req(dp->u.ptr_ctr.ptr, self_address, BLK_ALIGN(LVA));
       );


DO_ACTION(DA_Set_Read,
       dp->been_read += 1;
       );

DO_ACTION(DA_Clr_Read,
       dp->been_read = 0;
       );

DO_ACTION(DA_Inval_RO,
       dp->inval_start = dir_srv.vt;
       for (i = 0; i < num_nodes; i++) {
         if (BV_TST(dp->u.owners, i)) {
             send_put_req(i, self_address, BLK_ALIGN(LVA));
         }
          }
       );

DO_ACTION(DA_Forward_RW,
       if (!prot_shared_args.release_consistency)
           rw_inval_delay += (dir_srv.vt - dp->inval_start);
       send_get_resp(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA),
           dp->tid[dp->u.ptr_ctr.ptr], dp->version);
       );


DO_ACTION(DA_SI_Forward_RW,
       if (!prot_shared_args.release_consistency)
           rw_inval_delay += (dir_srv.vt - dp->inval_start);
       if (dp->u.ptr_ctr.ptr != self_address)
           send_si_get_resp(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA),
           dp->tid[dp->u.ptr_ctr.ptr], dp->version);
       else
           send_get_resp(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA),
           dp->tid[dp->u.ptr_ctr.ptr], dp->version);
       );

DO_ACTION(DA_Forward_RW_ACK,
       if (!prot_shared_args.release_consistency)
           rw_inval_delay += (dir_srv.vt - dp->inval_start);
       send_get_resp(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA),
           ACK_MASK | dp->tid[dp->u.ptr_ctr.ptr],
           dp->version);
       );

DO_ACTION(DA_SI_Forward_RW_ACK,
       if (!prot_shared_args.release_consistency)
           rw_inval_delay += (dir_srv.vt - dp->inval_start);
       if (dp->u.ptr_ctr.ptr != self_address)
           send_si_get_resp(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA),
           ACK_MASK | dp->tid[dp->u.ptr_ctr.ptr],
           dp->version);
       else
           send_get_resp(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA),
           ACK_MASK | dp->tid[dp->u.ptr_ctr.ptr],
           dp->version);
       );


DO_ACTION(DA_Ck_Ptr,
       int ack = 0;

       if (prot_shared_args.release_consistency)
           ack = ACK_MASK;

       if (dp->u.ptr_ctr.ptr == source || source == self_address)
           send_get_resp(source, BLK_ALIGN(LVA),
           ack|dp->tid[source], dp->version);
       else
           send_si_get_resp(source, BLK_ALIGN(LVA),
           ack|dp->tid[source], dp->version);

       );

DO_ACTION(DA_Forward_RO,
       int node;

       ro_inval_delay += (dir_srv.vt - dp->inval_start);
       node = dp->u.ptr_ctr.ptr;
       send_get_resp(node, BLK_ALIGN(LVA),dp->tid[node],
                      dp->version);
       bzero((char *) dp->u.owners, sizeof(dp->u.owners));
       BV_SET(dp->u.owners, node);
       );
```

```
DO_ACTION(DA_SI_Forward_RO,
       int node;

       ro_inval_delay += (dir_srv.vt - dp->inval_start);
       node = dp->u.ptr_ctr.ptr;
       send_si_get_resp(node, BLK_ALIGN(LVA),dp->tid[node],
              dp->version);
       bzero((char *) dp->u.owners, sizeof(dp->u.owners));
       if (!prot_shared_args.tearoff)
           BV_SET(dp->u.owners, node);
       );

DO_ACTION(DA_Ack_RW,
       send_inval_ack(dp->u.ptr_ctr.ptr, BLK_ALIGN(LVA));
       );

/* Actions requiring old values go above this point */
/* Actions modifying values go below this point */

DO_ACTION(DA_Zero, bzero((char *) dp->u.owners,
          sizeof(dp->u.owners)););
DO_ACTION(DA_Decr, dp->u.ptr_ctr.ctr--;);

DO_ACTION(DA_IncV,
       dp->version =
           (dp->version + 1) % ((1 << prot_shared_args.vbits));
       );

DO_ACTION(DA_Set_Ptr,
       int cnt = 0;
       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->u.owners, i))
               ++cnt;
       dp->u.ptr_ctr.ptr = source;
       dp->u.ptr_ctr.ctr = (cnt == 0 ? 0 : cnt-1);
       );

DO_ACTION(DA_Set_Bit,
       assert(!BV_TST(dp->u.owners, source));
       BV_SET(dp->u.owners, source);
       );

DO_ACTION(DA_Clr_Bit,
       assert(BV_TST(dp->u.owners, source));
       BV_CLR(dp->u.owners, source);
       if (dp->u.owners[0] == 0 && dp->u.owners[1] == 0)
           dp->state = DIR_IDLE;
       else
           dp->state = DIR_RO;
       );
```

```
DO_ACTION(DA_RW_SI_Clr_Bit,
       assert(BV_TST(dp->u.owners, source));
       BV_CLR(dp->u.owners, source);
       if (dp->u.owners[0] == 0 && dp->u.owners[1] == 0)
           dp->state = DIR_IDLE_TWO;
       else
           dp->state = DIR_RO_SI;
       );


DO_ACTION(DA_Clr_Bit_to_SI,
       assert(BV_TST(dp->u.owners, source));
       BV_CLR(dp->u.owners, source);
       if (dp->u.owners[0] == 0 && dp->u.owners[1] == 0)
           dp->state = DIR_IDLE_SI;
       );


DO_ACTION(DA_Clr_Bit_to_IRO,
       assert(BV_TST(dp->u.owners, source));
       BV_CLR(dp->u.owners, source);
       if (dp->u.owners[0] == 0 && dp->u.owners[1] == 0)
           dp->state = DIR_IDLE_R;
       );


DO_ACTION(DA_SI_Set_Bit,
       BV_SET(dp->si_bits, source);
       );

DO_ACTION(DA_SI_Clr_Bit,
       BV_CLR(dp->si_bits, source);
       );


DO_ACTION(DA_SI_Check_Bits,
           /* if any of the self invalidation bits are set */
           /* then the new state is IDLE_SI */
           /* else the state is IDLE */
       int cnt = 0;
       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->si_bits, i))
               ++cnt;
       if (cnt == 0)
           dp->state = DIR_IDLE;
       else
           dp->state = DIR_IDLE_SI;
       );

DO_ACTION(DA_Upgrade_RO,
           /* find out how many copies are outstanding */
           /* if there are not any, just send the response */
           /* and set the state to RW */
       int cnt = 0;
       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->u.owners, i))
               ++cnt;
       if (cnt == 0) {
           send_get_resp(source, BLK_ALIGN(LVA),
           dp->tid[source], dp->version);
      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = 0;
      dp->state = DIR_RW;
       }
       else {

      dp->inval_start = dir_srv.vt;
      for (i = 0; i < num_nodes; i++) {
          if (BV_TST(dp->u.owners, i)) {
         send_put_req(i, self_address, BLK_ALIGN(LVA));
          }
      }
      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = cnt-1;
      dp->state = DIR_RW_GET;
       }
       );


DO_ACTION(DA_RW_Upgrade_RO,
           /* find out how many copies are outstanding */
           /* if there are not any, just send the response */
           /* and set the state to RW */
       int cnt = 0;
       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->u.owners, i))
               ++cnt;
       if (cnt == 0) {
          send_get_resp(source, BLK_ALIGN(LVA),
           dp->tid[source], dp->version);

      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = 0;
      dp->state = DIR_RW;
       }
       else {

      dp->inval_start = dir_srv.vt;
      for (i = 0; i < num_nodes; i++) {
          if (BV_TST(dp->u.owners, i)) {
         send_put_req(i, self_address, BLK_ALIGN(LVA));
          }
      }
      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = cnt-1;
      dp->state = DIR_RW_GET;
       }
       );

DO_ACTION(DA_RC_Upgrade_RO,
           /* find out how many copies are outstanding */
           /* if there are not any, just send the response */
           /* and set the state to RW */
       int cnt = 0;

       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->u.owners, i))
               ++cnt;
       if (cnt == 0) {
        /* respond with data and ack */
      if (prot_shared_args.dsi_rw && source != self_address)
          send_si_get_resp(source, BLK_ALIGN(LVA),
              ACK_MASK|dp->tid[source], dp->version);
      else
          send_get_resp(source, BLK_ALIGN(LVA),
              ACK_MASK|dp->tid[source], dp->version);

      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = 0;
      dp->state = DIR_RW;
       }
       else {
        /* respond with data, cache must wait for ack’s */
      if (prot_shared_args.dsi_rw && source != self_address)
          send_si_get_resp(source, BLK_ALIGN(LVA),
              dp->tid[source], dp->version);
      else
          send_get_resp(source, BLK_ALIGN(LVA),
              dp->tid[source], dp->version);

      dp->inval_start = dir_srv.vt;
      for (i = 0; i < num_nodes; i++) {
          if (BV_TST(dp->u.owners, i)) {
         send_put_req(i, self_address, BLK_ALIGN(LVA));
          }
      }
      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = cnt-1;
      dp->state = DIR_GET_ACKS;
       }
       );

DO_ACTION(DA_RC_RW_Upgrade_RO,
           /* find out how many copies are outstanding */
           /* if there are not any, just send the response */
           /* and set the state to RW */
       int cnt = 0;
       assert(prot_shared_args.dsi_rw);

       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->u.owners, i))
               ++cnt;

           /* if there is only one ro block and it is this upgrade */
           /* then don’t give a self invalidate */
       if (cnt == 1 && BV_TST(dp->u.owners, source))
       {
           send_get_resp(source, BLK_ALIGN(LVA),
           ACK_MASK|dp->tid[source], dp->version);

      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = 0;
      dp->state = DIR_RW;
       }
       else {

      if (source != self_address)
          send_si_get_resp(source, BLK_ALIGN(LVA),
             dp->tid[source], dp->version);
      else
          send_get_resp(source, BLK_ALIGN(LVA),
             dp->tid[source], dp->version);
      if (BV_TST(dp->u.owners, source))
      {
          BV_CLR(dp->u.owners, source);
          --cnt; /* take off for the upgrade */
      }
      dp->inval_start = dir_srv.vt;
      for (i = 0; i < num_nodes; i++) {
          if (BV_TST(dp->u.owners, i)) {
         send_put_req(i, self_address, BLK_ALIGN(LVA));
          }
      }
      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = cnt-1;
      dp->state = DIR_GET_ACKS;
       }
       );

DO_ACTION(DA_RC_RW_Upgrade_RO_SI,

       /* always give a self inval block */

       int cnt = 0;
       assert(prot_shared_args.dsi_rw);
       for (i = 0; i < num_nodes; i++)
           if (BV_TST(dp->u.owners, i))
               ++cnt;

           /* if there is only one ro block and it is this upgrade */
           /* then don’t give a self invalidate */
       if (cnt == 1 && BV_TST(dp->u.owners, source))
       {
      if (source != self_address)
          send_si_get_resp(source, BLK_ALIGN(LVA),
              ACK_MASK|dp->tid[source], dp->version);
      else
          send_get_resp(source, BLK_ALIGN(LVA),
              ACK_MASK|dp->tid[source], dp->version);

           dp->u.ptr_ctr.ptr = source;
           dp->u.ptr_ctr.ctr = 0;
           dp->state = DIR_RW;
       }
       else {
           if (source != self_address)
          send_si_get_resp(source, BLK_ALIGN(LVA),
             dp->tid[source], dp->version);
      else
          send_get_resp(source, BLK_ALIGN(LVA),
             dp->tid[source], dp->version);

      if (BV_TST(dp->u.owners, source))
      {
          BV_CLR(dp->u.owners, source);
          --cnt; /* take off for the upgrade */
      }
      dp->inval_start = dir_srv.vt;
      for (i = 0; i < num_nodes; i++) {
          if (BV_TST(dp->u.owners, i)) {
         send_put_req(i, self_address, BLK_ALIGN(LVA));
          }
      }
      dp->u.ptr_ctr.ptr = source;
      dp->u.ptr_ctr.ctr = cnt-1;
      dp->state = DIR_GET_ACKS;
       }
       );

DO_ACTION(DA_Send,
           send_get_resp(source, BLK_ALIGN(LVA),dp->tid[source],
          dp->version);
       );

DO_ACTION(DA_SI_Send,

              send_si_get_resp(source, BLK_ALIGN(LVA),
                 dp->tid[source], dp->version);
        );

 DO_ACTION(DA_Send_RW_Ack,
            send_get_resp(source, BLK_ALIGN(LVA),
           ACK_MASK|dp->tid[source], dp->version);
        );

 DO_ACTION(DA_SI_Send_RW_Ack,
            send_get_resp(source, BLK_ALIGN(LVA),
           ACK_MASK|dp->tid[source], dp->version);
        );

DO_ACTION(DA_Nack,
        send_busy_nack(source, LVA, dp->tid[source]);
        );

 /*
  ** Perform swap for processor that has been waiting.
 */
 DO_ACTION(DA_Swap_Ptr,
        int old_data;
        old_data = *dp->swap_lva;
        *dp->swap_lva = dp->swap_data;
        send_swap_resp(dp->u.ptr_ctr.ptr, dp->swap_lva, old_data);
        bzero((char *) dp->u.owners, sizeof(dp->u.owners));
        );

 /*
  ** Save the swap address and data
 */
 DO_ACTION(DA_Save_Swap,
        dp->swap_lva = LVA;
        dp->swap_data = new_data;
        );

 /*
  ** Perform swap
 */
 DO_ACTION(DA_Swap,
        int old_data;
        int *ip;

        ip = (int *) LVA;
        old_data = *ip;
        *ip = new_data;
        send_swap_resp(source, LVA, old_data);
        );

 /*
 ** Nack a swap when too much is going on.
 */
```

```
DO_ACTION(DA_Swap_Nack,
    send_swap_nack(source, LVA);
);

/* Set the new stat for the block */
DO_ACTION(DA_Set_State,
          dp->state = NEWSTATE(aep);
);
```