Trap-driven Memory Simulation

by

Richard Albert Uhlig

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1995

Doctoral Committee:

Professor Trevor Mudge, Chair
Associate Professor Richard Brown
Assistant Professor Peter Chen
Assistant Professor Stuart Sechrest
Peter Bird, Computer Architect, ACRI, Lyon, France
Associate Research Scientist Peter Honeyman

Diese Doktorarbeit ist ohne Dreckfehler

© Richard Albert Uhlig 1995
All Rights Reserved

Dedication

To my parents, Albert and Trudy Uhlig

Acknowledgments

My parents created me. As my father has often said, they had fun doing it.

Peter Bird encouraged me to start along this path and taught me strategies for finishing before I even started. He saved me at least a year or two with his advice.

Trevor Mudge gave me all the resources and guidance that I needed without ever constraining the direction of this research. His sense of humor helped to break up the monotony.

Stuart Sechrest taught me most of what I know about operating systems. Of all the courses in graduate school, I enjoyed his the most.

Richard Brown, Peter Chen and Peter Honeyman rounded out my committee. I am grateful for their time and interest.

Tim Stanley ported Tapeworm to OSF/1, Chih-Chieh Lee ported Tapeworm to the 486, and Mike Smith of Harvard reviewed Chapter 2 on trace-driven simulation. Their help was much appreciated.

NSF paid for three years of my graduate education and (D)ARPA paid for the rest. This makes paying taxes a little easier.

All the students of Room 2001 and 3003, past and present, and too numerous to mention individually, provided the densest collection of bright people that I've ever been surrounded by. My conversations with Mike Upton, Tom Huff, Tim Stanley and Ajay Chandna were particularly enjoyable.

Hilary DeKraai offered emotional support and proofreading when it was most needed, especially during the final days. I look forward to our upcoming adventures.

I am most indebted to David Nagle. Every aspect of our research, from establishing the direction of study, to writing the initial proposals for funding, to carrying out the work and writing up the results has been a joint effort. Our partnership has taught me how powerful good collaboration can be. He is my dearest friend and I look forward to our continued partnership on future ventures.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

Many researchers feel that uniprocessor cache design is a well-understood process that warrants no further investigation. While caches are among the most thoroughly-studied components of computer architectures, the current level of understanding has not helped industry to consistently produce microprocessors with balanced memory systems. A few recent examples support this point:

- The performance of the Macintosh 680x0 emulator running on a PowerPC 603 is roughly 60% that of the earlier PowerPC 601 running the same emulator at the same clock rate (80 MHz) [MacWeek94; MPReport94]. A more optimized version of the 680x0 emulator that is twice as fast on the 601 delivers only a 5% improvement on the 603. These performance problems have been attributed to the 603 cache structure, which implements split, 2-way, 8-KB instruction and data caches, compared with the unified, 8-way, 32-KB cache of the 601. The disappointing performance of the 603 has delayed the introduction of PowerPC-based Apple Powerbooks, which are awaiting a re-design of the chip, called the 603+, with larger on-chip caches.

- Disappointing performance of Windows NT on the MIPS R4000 processor led to a redesign of the memory system in its successor chip, the R4200. The new design doubles the size of the direct-mapped instruction cache to 16KB, but leaves the direct-mapped data cache at 8KB. Fortunately, simulations prevented designers from making the mistake of allocating die area to the data cache, rather than the instruction cache. The reverse configuration (8-KB I-cache, 16-KB D-cache) yields 7% less performance under Windows NT [MPReport93].

- Although the peak cycles per instruction (CPI) of the first-generation DEC Alpha chip (21064) is 0.5, hardware-monitor measurements reveal that the actual CPI of the chip

> is typically in the range of 2 to 3 on the SPEC92 benchmarks and over 4 on the data transaction processing TPC benchmarks [Cventanovic94]. This 200-MHz chip implements relatively meager, direct-mapped, 8-KB, primary instruction and data caches, resulting in memory stalls that account for more than 60% to 70% of lost performance. Later versions of the Alpha architecture (e.g., the 21164) have implemented much more substantial on-chip caches, enabling the processor to come closer to its potential peak performance.

The belief that cache design is a solved problem is not consistent with the reality that several recent microprocessors exhibit memory-system problems that either prevent them from achieving the true potential of their high clock rates, or that require costly chip redesigns to reach their performance goals.

If caches are so well understood, why do memory-system problems persist? One reason is that hardware technologies are constantly changing, upsetting previous system designs that achieved a balance between the processor and the memory system. In particular, memories are generally getting larger and cheaper, but not faster [Touma92; Jouppi94]. At the same time, the move towards higher clock rates and multi-issue machines are making processors more memory hungry. In his dissertation, Upton shows that microprocessor clock rates have increased at a rate of 40% per year during the past decade, while DRAM speeds have only increased at a rate of 11% per year during the same period [Upton94]. These trends are increasing the sensitivity of overall performance to cache designs. Jouppi notes that disabling the cache of the VAX 11/780, a machine introduced in the late 1970’s, would only increase run times by a factor of 1.6 [Jouppi90]. Disabling the cache of a more recent machine, the HP 9000/735 introduced in the early 1990’s, would increase its run times by a factor of 15 [Upton94].

A more subtle reason behind unbalanced memory-system designs is a shift in the utilization of memory resources caused by changes in operating system and application software. A memory system that has been optimized for a given workload, consisting of certain versions of applications and an operating system, is not necessarily optimal for later versions of the same software. An example of this is shown in Table 1.1 which gives the contribution to processor stall cycles of different memory-system components (TLB, I-cache, D-cache and Write Buffer) in an engineering workstation. The last two rows of the table give the performance of the same application (an MPEG decoder, see Figure 1.1) on the same machine (a MIPS-based DECstation) under two

Operating System	Measurement Method	CPI	TLB	I-cache	D-cache	Write Buffer	Other
None	pixie + cache2000	1.43	0.01(1%)	0.06(14%)	0.05(13%)	0.18(41%)	0.14(32%)
Ultrix	Monster	1.66	0.01(2%)	0.10(15%)	0.26(39%)	0.14(21%)	0.15(23%)
Mach	Monster	2.06	0.15(14%)	0.32(30%)	0.30(28%)	0.21(20%)	0.08(8%)

Table 1.1 The Effect of Operating Systems on CPU Stall Behavior

This table shows three collections of CPU stall measurements for the `mpeg_play` workload (see Figure 1.1) running on a DECstation 3100. The DECstation 3100 has 64-KB, off-chip, direct-mapped instruction and data caches and a 64-entry, fully-associative TLB. The cache line size is one word. The different columns show the total CPI and the contributions of different system components to CPI increases above 1.0 (this is a single-instruction issue machine). Numbers in parenthesis give the relative contribution of each stall type to CPI increases above 1.0. *Other* stands for non-memory related stalls, such as integer and floating-point interlock cycles.

In the first row, CPI was determined by a `cache2000` simulation driven by `pixie`-generated traces [MIPS88]. `cache2000` was configured to simulate a memory system with the same parameters as the DECstation 3100. Because `pixie` generates single-task, user-only references, this measurement only includes references made by the `mpeg_play` task itself (40% of total execution activity). The last two rows give CPI under both Ultrix and Mach as measured by direct monitoring of the DECstation hardware, using the Monster hardware monitoring system [Nagle92]. These last two measurements include 100% of system activity.

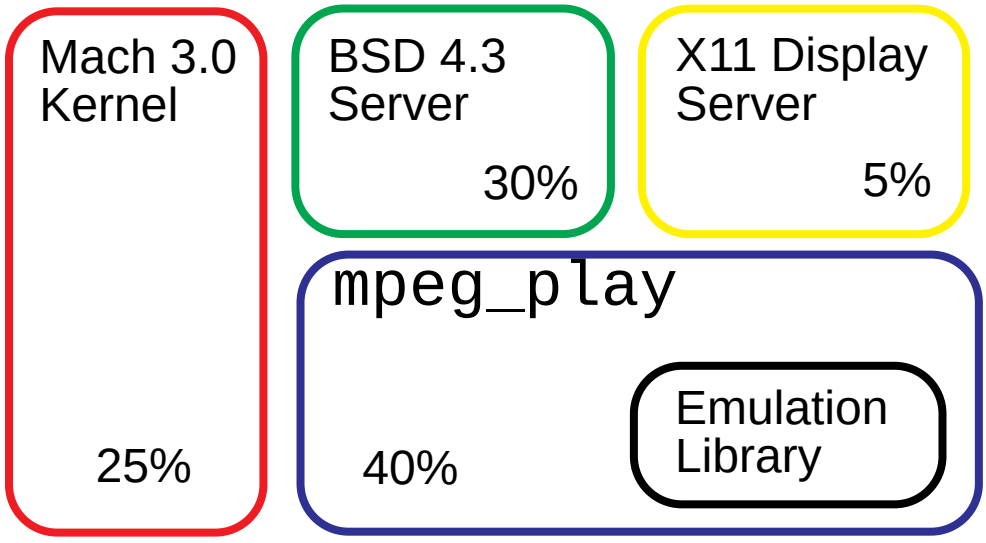


Figure 1.1 The MPEG Decoder Workload

`mpeg_play` decodes an MPEG-compressed video stream and displays the resulting sequence of frames in an X window [Patel92]. We will frequently use `mpeg_play` throughout this dissertation as an example of an application that presents a several difficulties to accurate memory-system simulation.

The components of `mpeg_play`, depicted in this figure, include a dynamically-linked emulation library, an X11 display server, a user-level BSD server, the Mach 3.0 micro-kernel and the `mpeg_play` process itself. The relative importance of each workload component is given as a percentage of total execution time.

The memory-referencing behavior of this application is difficult to analyze because most trace-driven simulators are unable to account for the activity of multiple user-level tasks, dynamically-linked code, or operating system servers and kernels.

different operating systems, Ultrix and Mach. Note that although the workload and machine are the same, the overall CPI under the two operating systems differs by 25%. More importantly, when compared against Ultrix, there is a shift toward greater utilization of the TLB and I-cache under Mach. This shift in memory utilization leads to a different optimal allocation of chip die area for on-chip TLBs and caches [Nagle94].

The increasing sensitivity of overall system performance to cache design, combined with a shift in how memory-system components are utilized by modern applications and operating systems, presents a challenging design problem to computer architects. Before architects commit a

Trace-driven Simulation

```
while (address = next_address(trace)){
   if (search(address))
      hit++;
   else {
      miss++;
      replace(address);
   }
}
```

Trap-driven Simulation

```
traps invoke trap_handler(address):

trap_handler(address){
   miss++;
   clear_trap(address);
   displaced_address = replace(address);
   set_trap(displaced_address);
}
```

Figure 1.2 Trace-driven and Trap-driven Simulation Algorithms

The core execution loops of trace-driven and trap-driven simulators. This code omits many details of actual simulation, such as the treatment of writes and assigning penalties for different types of misses (e.g., in a critical-word-first cache). Single-pass simulators that evaluate multiple memory configurations in a single trace pass also have a more complex structure [Mattson70; Hill87; Thompson89; Sugumar93].

particular design to silicon, they need a way to evaluate the performance of design alternatives more quickly, accurately and cheaply.

Trace-driven memory simulation is probably the most popular method for dealing with this problem [Smith82, Holliday91]. Trace-driven simulation begins by collecting a long sequence of memory references made by some workload of interest. A trace-driven simulator then executes a loop similar to that shown at the top of Figure 1.2. The processing steps include obtaining an

address from a trace, searching for this address in a simulated cache, and then invoking a replacement policy in the event of a miss. The trace addresses can come from a file created by a trace-extraction tool, or they might be generated “on the fly” by an annotated workload. The search procedure involves indexing a data structure that represents the cache and then, depending on the associativity of the cache, performing one or more comparisons to test for a hit. Though a simple operation, the search and test must be performed for every address in the trace.

Over a period of at least 30 years, this basic approach to memory simulation has been continually improved with respect to flexibility, portability, speed, and accuracy. Despite this progress, trace-driven simulators still suffer from an inherent speed bottleneck that is a direct consequence of their basic operation: (1) read *each* address, (2) search and (3) replace on a miss. The result is that the very fastest trace-driven simulators are still from one to two orders of magnitude slower than actual hardware.

This dissertation is about a different method for memory-system analysis, called *trap-driven memory simulation*. This method is based on the observation that most memory references, such as cache hits, do not change a cache’s state. Those references that *do* change cache state, such as cache misses resulting in cache-line refills, are far less-frequent events. A trap-driven simulator is designed to only be invoked (trapped to) when the state of a simulated memory structure changes.

Trap-driven simulation is best explained by contrasting its essential features against those of traditional trace-driven cache simulation (see Figure 1.2). As discussed previously, the processing steps in trace-driven simulation include reading each address from a trace, searching for each address in a simulated cache, and then invoking a replacement policy whenever an address misses the simulated cache. A trap-driven simulator operates on a different principle. It is driven not by address traces, but by traps into an operating system kernel where it resides. A trap-driven simulation begins by setting traps on all memory locations in a workload’s address space.[1] Locations with traps set represent memory that is not currently resident in a simulated cache structure. As the workload executes, the first reference to each such location causes a trap into the kernel, which is directed to the trap-driven simulator. Because all such traps represent simulated cache misses, there is no need to search a data structure representing the simulated cache. The

1. Setting a trap on a memory location means that any reference to it results in a kernel trap that invokes the trap-driven simulator. We discuss a variety of methods for achieving this effect in later chapters of this dissertation.

simulator simply counts the miss and then clears the trap on the required memory location. Clearing the trap effectively caches the memory location in the simulated cache structure because subsequent references to the location will proceed uninterrupted. As a simulated cache begins to fill, incoming traps may collide with previously cached memory locations that map to the same cache line. To mimic the displacement of such lines from an actual cache, the simulator sets a new trap on a previously-cached memory location in accordance with some replacement policy.

The most important aspect of this form of simulation is that it offers the potential for increased simulation speed. Because simulated cache hits do not trap to the simulator, they are processed at the full speed of the underlying hardware. Overall simulation slowdowns are therefore determined by the cost to process cache misses, which are a far less frequent event than cache hits. Trap-driven simulation is, in principle, a promising approach to memory simulation because it overcomes bottlenecks inherent in trace-driven simulation speed. However, a number of practical questions must be answered to determine if trap-driven simulation is actually a viable alternative to trace-driven simulation:

(1) *Flexibility:* What is the range of memory configurations, policies, and performance metrics that can be simulated by a trap-driven simulator? Are there any inherent limitations in the flexibility of the method?

(2) *Portability:* How difficult is it to implement a trap-driven simulator, and what hardware support is required? How could future hardware be designed to better support trap-driven simulation?

(3) *Speed:* How much overhead is incurred by a simulator trap? Is it so large that overall simulation slowdowns are worse than traditional trace-driven simulation?

(4) *Accuracy:* What forms of simulation error are exhibited by a trap-driven simulator and how do these errors compare with trace-driven simulation? Is it possible to take into account references made by all workload components, including multiple user tasks, operating system servers and the kernel?

The purpose of this dissertation is to answer these questions. We begin in Chapter 2 by establishing the current state-of-the-art in trace-driven simulation with a survey of over 50 recent trace-driven simulation tools. The conclusions drawn from this survey will serve as a basis for comparison in our study of trap-driven simulation, which begins in Chapter 3. We introduce the

design of a prototype trap-driven simulator, named *Tapeworm*, which we then use in the remaining chapters to examine the four issues of *flexibility, portability, speed* and *accuracy.*

We will show that although first-generation trap-driven simulators have exhibited some problems with flexibility and portability, many of these limitations can be overcome through careful simulator design, and with the help of minor and inexpensive modifications to future host hardware. More importantly, we show that trap-driven simulation delivers on its potential to break past bottlenecks in trace-driven simulation speeds. The high speed of trap-driven simulation, combined with its ability to accurately account for multi-task and operating system effects, makes it an attractive alternative to trace-driven simulation in future memory-system designs.

Chapter 2

Trace-driven Simulation

Trace-driven simulation has been used in the evaluation of memory systems for decades. In his 1982 survey of cache memories, A. J. Smith gives examples of trace-driven memory-system studies that date as far back as 1966 [Smith82]. This chapter is a survey of developments in trace-driven memory simulation during the past 10 years. It plays an important role in this dissertation because it helps to define the classic problems associated with memory-system simulation and the motivating factors behind the vast body of published works on trace-driven techniques. Because the trap-driven approach is an alternative method for memory-system simulation, any arguments for its use must include a comparison against more established techniques. Therefore, a second objective for this chapter is to evaluate the current state-of-the-art in trace-driven simulation in preparation for a comparison with trap-driven simulation. In later chapters, we will see that many of the advances in trace-driven memory evaluation, such as multi-configuration simulation algorithms and set sampling, can also be applied to trap-driven simulation. Thus, a final justification for this survey is that it also benefits the further development of trap-driven simulation.

This is not the first survey of trace-driven simulation techniques. Holliday examined the topic for uniprocessor and multiprocessor memory-system design [Holliday91] and Stunkel et al. studied trace-driven simulation in the specific context of multiprocessor design [Stunkel91]. However, developments in the field during the past three to four years, together with the need to elaborate issues of importance in this dissertation warrant a new examination of these methods.

Although trace-driven methods have been successfully applied to other domains of computer architecture, such as the simulation of instruction microarchitectures, this survey will focus on trace-driven *memory-system* simulation. In particular, we are primarily interested in uniprocessor memory-system design that takes into account operating-system activity. Throughout this chapter, we will comment on the strengths and weaknesses of various trace-driven approaches in terms of

desirable features for any memory simulation technique, such as *flexibility, portability, speed,* and *accuracy.*

This study will begin by defining the characteristics of an ideal trace-driven simulation environment. This is followed by an analysis of over 50 actual implementations of trace-driven simulation tools, noting how they compare against the ideal. This chapter has been organized so that it can be read at two different levels of detail. A good sense of the overall goals and strategies of trace-driven simulation, along with the capabilities of existing tools can be obtained by reading just the first and last sections (Section 2.1 and Section 2.3) and then scanning the summary tables that accompany each subsection in Section 2.2. An in-depth reading of the text in Section 2.2 offers additional information and insights that are not captured by the summary tables alone.

2.1 Ideal Trace-driven Memory Simulation

An ideal trace-driven memory simulation environment supports three main activities: *trace collection*, *trace reduction* and *trace processing* (See Figure 2.1).

Trace collection is the process of determining the exact sequence of memory references made by some workload of interest. To ensure accurate simulations, the collected address trace should be as close as possible to the stream of memory references made by the workload when running on an actual system. In particular, an ideal trace should be *complete*, *detailed*, and free of any *distortions*.

A *complete* trace includes all memory references made by each component of the system, including all user-level tasks and the operating system kernel. User-level tasks include not only applications, but also OS server and daemon tasks that provide services such as a file system or network access. Because dynamically compiled and linked code is becoming increasingly important in applications such as multiple API and ABI emulation [Nagle94; Cmelik94], a complete trace should also include references made by this form of code.

An ideal *detailed* trace is one that is annotated with information beyond simple raw addresses. Useful annotations include changes in VM *page-table state* for translating between physical and virtual addresses, *context switch points* with identifiers specifying newly activated tasks, and tags that mark each address with a *reference type* (read, write, execute), *size* (word, half word, byte) and a *timestamp*.

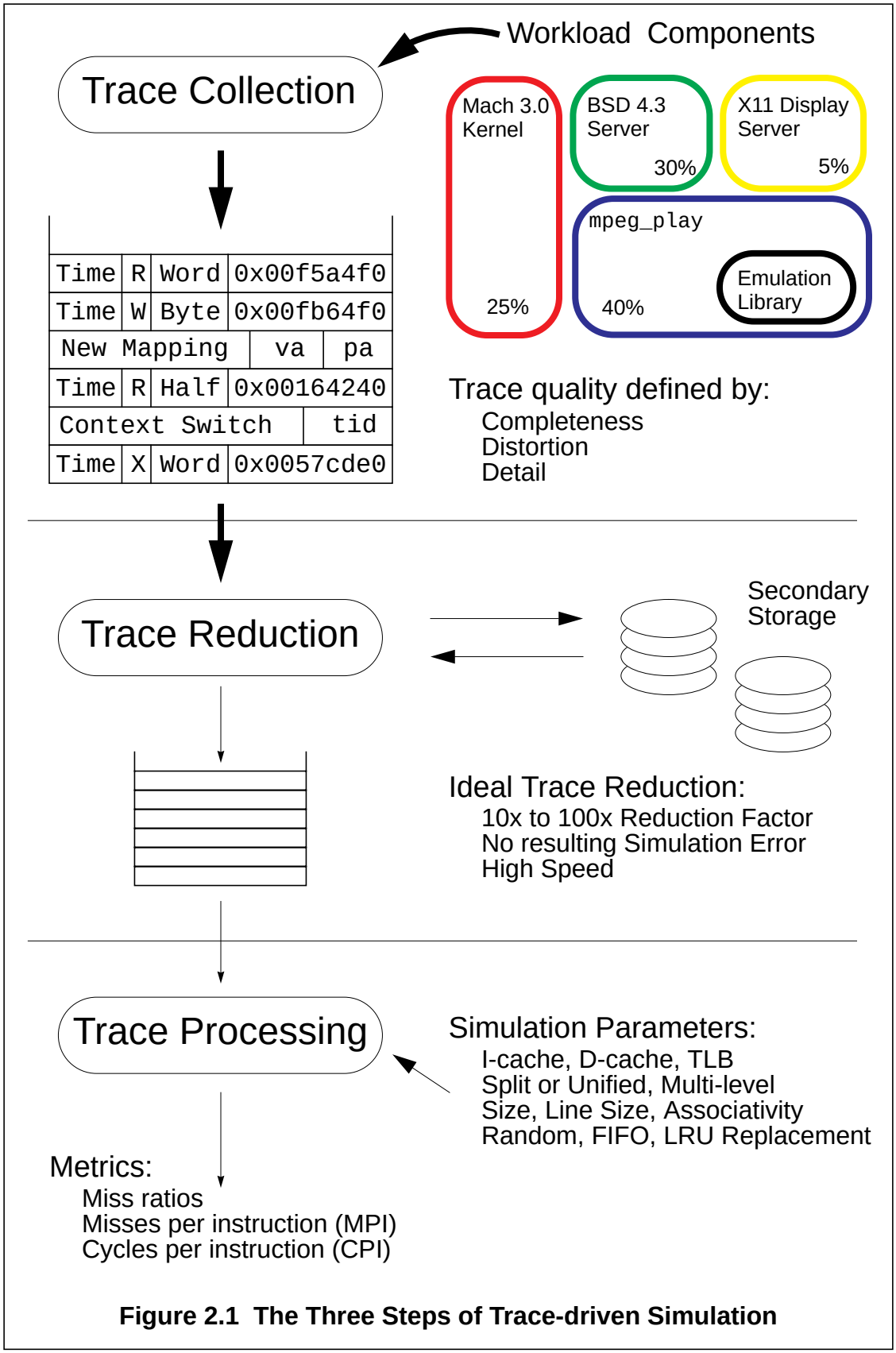


Figure 2.1 The Three Steps of Trace-driven Simulation

Ideal traces should be *undistorted* so that they do not include any additional memory references, or references that appear out of order. Though this is usually not a problem with single-task traces, various forms of distortion can affect the quality of more complete traces that include multi-task and operating-system references. These distortions, which will be discussed in greater detail later, include *trace discontinuities*, *time dilation* and *memory dilation*.

Although the three aspects of trace quality described above are the most important considerations, ideal trace collectors should have other characteristics as well. In particular, *portability*, both in moving to other machines of the same type and to machines that are architecturally different is important. Finally, an ideal trace collector should be *fast*, *inexpensive* and *easy to operate*.

Address traces can be very large, consuming both storage space and processing time. An ideal trace-driven simulation environment should include *trace-reduction* techniques that help to reduce these space and time requirements by removing unneeded or redundant data from full address traces. Ideal trace reduction should achieve very high levels of data compression without affecting the accuracy of simulations performed by the reduced traces. It may be acceptable to relax the constraint of exact trace reduction if higher levels of compression can be attained and if the resulting degree of simulation error is low. As with ideal trace collection, ideal trace reduction should be fast, inexpensive and easy to use.

The final stage of trace-driven simulation is *trace processing*. An ideal address-trace processor would be able to simulate a wide range of memory system hierarchies consisting of caches, TLBs, and primary- and secondary-storage devices. Ideally, these simulations would produce a range of performance metrics, such as miss ratios, misses per instruction (MPI) and memory cycles per instruction (CPI) for each component of the memory system.[1] An ideal trace processor would be able to simulate all memory system configurations of interest in a single pass over the trace data, quickly and cheaply, producing error-free performance metrics, even from reduced trace data.

The individual stages of trace-driven simulation must be connected through *trace interfaces* so that trace data can flow from one stage to the next. In the ideal case, the overhead of these interfaces would be negligible, adding little to the overall simulation time. Ideal trace interfaces

1. These performance metrics, along with others, are defined in greater detail in Chapter 4.

would also offer unlimited buffering so that different stages of trace-driven simulation can be performed at different times and so that trace data can be reused without having to be recollected.

2.2 An Analysis of the State-of-the-art in Trace-driven Simulation

Achieving all of the characteristics of ideal trace-driven simulation as defined above is difficult to do in practice. Nevertheless, considerable progress has been made during the past decade to develop good trace-driven simulation environments. In this section we study the techniques, the algorithms and several example implementations of actual trace-collection, trace-reduction and trace-processing tools to determine how close they come to the ideal.

Before we begin our analysis, we must define an important metric that we use to compare the speed of different trace-driven simulation tools. The rate at which addresses are collected, reduced or processed is one natural way to measure speed, but this metric makes it difficult to compare trace collectors or processors that have been implemented on dissimilar hardware. Because the number of addresses processed per second by a particular trace processor is a function of the speed of the host hardware on which it is implemented, it is not meaningful to compare this rate against a different trace-processing method implemented on slower host hardware. To overcome this difficulty, we report all speeds in terms of *slowdown* relative to the host hardware from which traces are collected from or processed on. Depending on the context, slowdowns are computed in a variety of ways:

Slowdown = (Address Collection Rate) / (Host Address Generation Rate) (Eqn 2.1)

Slowdown = (Address Processing Rate) / (Host Address Generation Rate) (Eqn 2.2)

Slowdown = (Total Simulation Time) / (Normal Host Execution Time) (Eqn 2.3)

Note that each of these definitions divides by the speed of the host hardware and thus allows a rough comparison of two methods implemented on different hosts.

2.2.1 Trace Collection

Of the three basic stages of trace-driven simulation, the problem of trace collection is probably the most difficult and has certainly received the most attention in terms of published work.

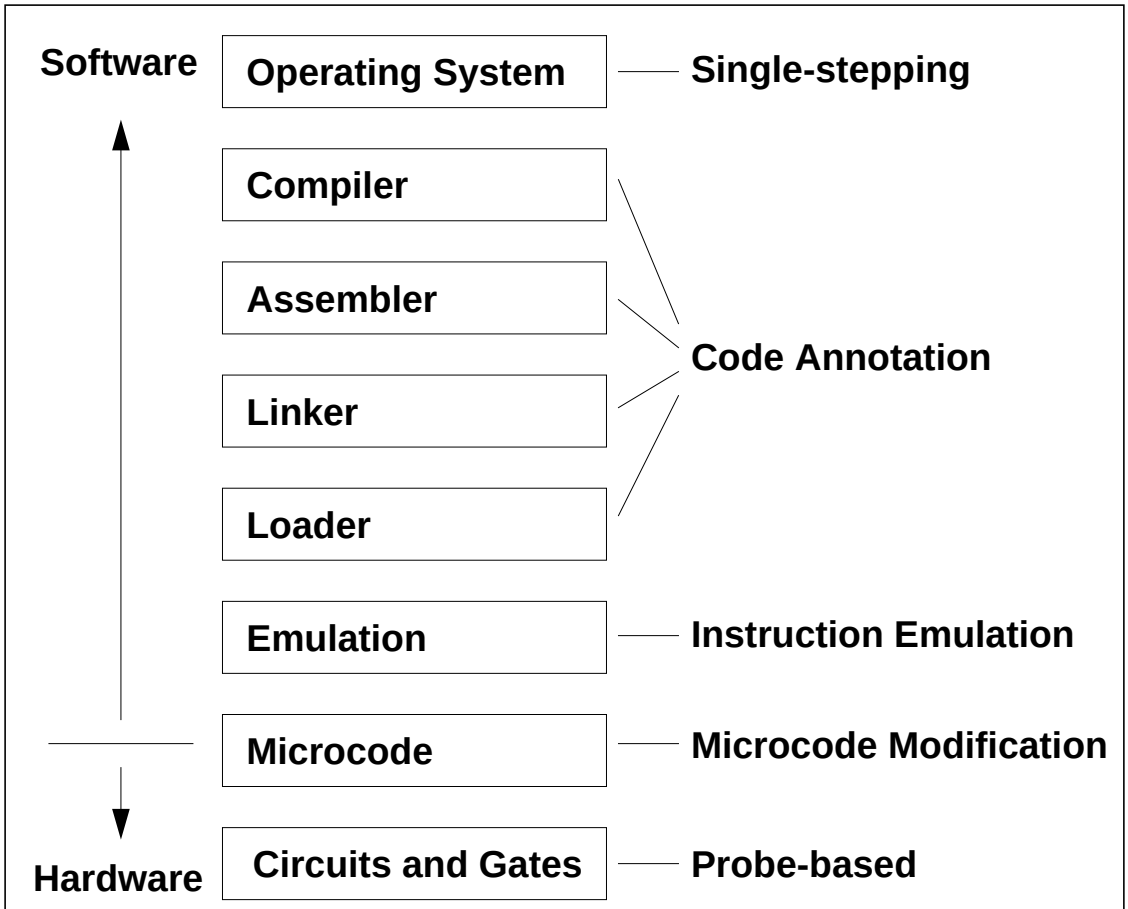


Figure 2.2 Levels of System Abstraction and Trace Collection Methods

This figure illustrates the relationship between different trace-collection methods and the various levels of system abstraction. Collecting traces at any given level holds certain advantages and disadvantages.

Address traces have been extracted at virtually every system level, starting at the circuit and microcode levels all the way up to the compiler and operating system levels (see Figure 2.2). Table 2.1 arranges approaches to trace collection into five basic categories, along with a summary of their defining characteristics. After examining each of these different methods in greater detail, we will return to this table to draw general conclusions about the state-of-the-art in trace collection techniques.

Characteristics		Probe-based	Microcode Modification	Instruction-set Emulation	Static Code Instrumentation	Single-step Execution
Completeness	Multi-task	Yes	Yes	Maybe	Maybe	No
	Kernel	Yes	Yes	Maybe	Maybe	No
	Dynamically-compiled	Yes	Yes	Yes	No	No
	Dynamically-linked	Yes	Yes	Yes	Maybe	No
Detail	Tags (R / W / X / Size)	Yes	Yes	Yes	Yes	Yes
	Virtual Addresses	Maybe	Yes	Yes	Yes	Yes
	Physical Addresses	Yes	Yes	Emulated	No	Yes
	Task Identifiers	Maybe	Yes	Emulated	Maybe	N/A
	Time Stamps	Yes	No	Maybe	No	No
Distortions	Discontinuities	Yes	Yes	No	Maybe	N/A
	Time Dilation	No	10 - 20	No	2 - 30	N/A
	Memory Dilation	No	No	No	4 - 10	N/A
Slowdown		1,000 +	10 - 20	15 - 75	10 - 30	50 - 100
Memory		Ext. Buffer	Host Buffer	4 - 40	10 - 30 + Buffer	Host Buffer
Expense		High	Medium	Medium-Low	Medium-Low	Low
Portability		Low	Very Low	High-Medium	Medium	High
Ease-of-Use		Low	High	High	High-Low	High

Table 2.1 Summary of Trace-collection Methods

This table summarizes the characteristics of five common methods for collecting address traces. For the descriptions of trace quality (*completeness, detail* and *distortions*) a *Maybe* entry means that the method has inherent difficulty providing data with the given characteristics, but there are examples of tools in the given category that overcome these limitations. The ranges given in the *slowdown* row exclude times for excessively bad implementations.

External Hardware Probes

A straightforward method for collecting address traces is to record signals from electrical probes physically connected to the address bus of a computer while it runs a workload. These address and control signals are fed into a high-speed memory buffer at the full speed of the monitored system. Some mechanism is required to move the contents of the hardware trace buffer to a standard storage device, such as tape or disk, so that it can be processed at a later time. If a long, continuous address trace is desired, then the buffer must either be very large or there must be some way to stall the monitored system. If there is no way to stall the system, then several discontinuous address-trace samples can be acquired and concatenated together. The resulting trace exhibits a form of distortion that we will call *trace discontinuity.* Under certain circumstances (see the discussion on time sampling in Section 2.2.2), traces with discontinuities can be used to perform the same sort of memory simulations enabled by longer, continuous traces, but with some sampling error. Table 2.2 summarizes several probe-based trace collectors recently described in the literature. We discuss each in greater detail below.

Most commercial logic analyzers provide the necessary hardware to construct a trace-collection system [Tektronix94; HP91]. Alexander et al. used this approach by connecting a logic analyzer to a National Semiconductor 32016-based workstation running Genix to collect address traces for TLB and cache simulation [Alexander85; Alexander86]. The small size of the trace buffer (4096 entries of 32 bits each) required the design of some circuitry to place the processor in a stalled state while the buffer was unloaded to a secondary-storage device. A similar approach was used by a team including the author who connected a DAS 9200 logic analyzer to an R2000-based DECstation 3100 to obtain traces in a system called *Monster* [Nagle92]. In this system, the operating system kernel was modified to stall the machine in a software loop, avoiding the need to design any additional hardware. Some logic analyzers provide interchangeable probes to more easily support multiple architectures. For example, in addition to support for MIPS processors, the DAS 9200 also has probe modules for most other popular microprocessors. Fuentes has demonstrated this portability by using a DAS 9200 to collect addresses from both Alpha-based and Pentium-based workstations [Fuentes93].

When a probe-based trace collector stalls a processor, subsequent activity by that processor may be different from the activity that would have ensued had the stall not occurred. In other words, the traces become distorted. The resulting traces still capture all activity, but they still exhibit *trace discontinuities* because of gaps in real time. For example, if only the processor is

Reference	Name	Processor	Buffer Size		Stall Method	Completeness	Download Channel
			Entries	Entry Size			
[Alexander85]	—	NS 32016	4 K	32 bits	HOLD Logic	All References	Serial
[Nagle92]	Monster	R2000	512 K	96 bits	Kernel Idle Loop	All References	Ethernet
[Fuentes93]	—	Alpha, Pentium	512 K	156 bits	None	Cache Misses	Ethernet
[Happel92]	—	R2000	8 M	40 bits	—	All References	—
[Fuentes93]	MTM	i486	33 M	80 bits	None	Bus Transactions	Ethernet
[Flanagan92] [Flanagan94]	BACH	i486, 68030, SPARC	80 M	96 bits	High-priority Interrupt	All References	Parallel DIO Board
[Torellas92]	DASH	R3000	2 M	72 bits	Master Process	Bus Transactions	Ethernet
[Biomation91]	K450M	—	80 M	64 bits	—	—	12 Mbits/sec DMA

Table 2.2 Probe-based Trace Collectors

All of the probe-based trace collectors in this table collect complete address traces with multi-task and operating system references. Buffer size determines the maximum sample size of uninterrupted memory references that can be captured. Most collectors can stall the monitored system when the trace buffer becomes full, but some cannot (see *Stall Method*). The speed with which the trace buffer can be unloaded is determined by the *Download Channel* which typically moves data at much lower bandwidths than the rate at which traces are acquired. Some probe-based trace collectors are only able to collect cache misses or bus transactions, not complete address traces (see *Completeness*).

stalled, external I/O devices, such as disks or network controllers, will continue to operate and will appear to the stalled processor to respond faster than they normally would. Stalling too long or too frequently may result in distorted traces that do not accurately reflect true system operation.

A problem with hardware monitors based on logic analyzers is that their trace buffer sizes are often relatively small (4 K-entries to 128 K-entries), resulting in frequent processor stalls, and thus greater distortion in the trace due to discontinuities. Special-purpose hardware with very large, high-speed memories has been built to reduce the extent of this problem. Biomation Corporation builds a trace-collection system with 80 million trace buffer entries [Biomation91]. The trace collector described in [Happel92] has a 40 M-byte trace buffer, large enough to hold 8 million memory references at a time. The *Magellan Trace Machine (MTM)* has buffer that can hold 33 million bus transactions [Fuentes93], and recent versions of the *Bach* system use similarly large buffers [Flanagan94].

The trend towards higher levels of chip integration creates a problem for probe-based trace collection. Most recent microprocessors implement at least their primary caches and TLBs on-chip, making many of their important address and control signals inaccessible to external probes [MPReport93; MPReport94]. Examples of probe-based trace collectors that are limited in this way are described in [Torrellas92] and [Fuentes93]. One solution to this problem is to deactivate on-chip caches to force all load and store operations off chip where they can be detected by external probes. However, this action can perturb the behavior of the system. Even if the resulting trace distortion is considered acceptable, some processors do not support disabling of on-chip caches in a general way [Digital92; Fuentes92]. Although full address traces are desirable, a trace of just cache misses is by no means worthless. As we will see in Section 2.2.2 on trace reduction, a trace of cache misses can still be used to simulate other cache configurations, subject to certain restrictions.

There are numerous other examples of hardware-based monitors that give performance information specific to the particular memory system implemented by the host hardware. These monitors, which usually consist of collections of event counters [Nagle92; Digital92; Cvetanovic94; IBM90] or address histograms [Emer84; Clark85b], do not collect address traces and therefore cannot be used to determine the performance of memory systems that are different from the monitored system.

The main advantage of all of the probe-based trace collectors described above is their ability to capture trace sequences complete with both user and kernel memory references, and free of most forms of trace distortion, provided the trace buffer is deep enough. Although the traces are complete, this does not necessarily mean that they are easy to interpret. Hardware events such as cache misses, integer- and floating-point-unit stalls, exceptions and interrupts all must be separated from run cycles to determine the actual type (read, write, execute) and size (word, half word, byte) of the memory references made by a monitored processor. This essentially amounts to implementing the inverse function of the processor sequencer, either in the trace-collecting hardware, or in a trace post-processing tool [Nagle92]. Because the addresses captured by a probe-based monitor are usually physical addresses, special methods that require cooperation from the host OS must be used to reverse translate these physical addresses to their matching virtual addresses [Grimsrud93]. For similar reasons, it is often difficult to relate a given memory reference to the task that made it without assistance from a modified OS kernel that emits *trace markers* or other annotations as clues [Torellas92; Nagle92; Fuentes93]. These problems all follow from the fact that probe-based trace collectors are external to the monitored system and therefore do not have easy access to operating system data structures held inside the system.

A common misconception regarding trace collection using hardware probes is that the technique is very fast. While it is true that acquisition of the trace proceeds at the full speed of the monitored system, it is important to include the overhead of managing trace-buffer overflow and unloading the trace-buffer memory. This overhead is typically not reported in published papers, but because most systems can only unload these buffers through some form of relatively low-bandwidth channel (see Table 2.2), this overhead is necessarily high. For a system where overhead data is available (Monster), approximately 12 hours are required to obtain 11 seconds of real-time system activity in a fully-automated system. Fuentes has reported that a similar delay of 45 minutes is required to download about one second of real-time activity captured by the MTM system [Fuentes93]. These delays represent effective slowdowns of more than a thousand times the speed of unmonitored hardware. The overhead from both these systems comes from moving trace-buffer data over an Ethernet to a machine with SCSI-connected disks. Most of the other systems listed in Table 2.2 use similar or even lower-bandwidth interconnect to the trace buffer, so their overheads should be comparable or higher.

Hardware probe-based methods share other common disadvantages. The first is expense. Logic analyzers with deep trace memories cost in the range of $50,000 to $200,000 [Tektronix94;

HP91]. These amounts are probably low compared to the engineering costs associated with designing custom hardware as in [Flanagan92] or [Torellas92]. A second problem is portability. Although logic analyzers like the DAS 9200 support most popular microprocessors with custom probes, making it easier to move to other architectures, it is often necessary to physically modify the motherboard or chassis of the monitored system to enable probe access to the signals of interest [Nagle92; Fuentes94]. Finally, these systems are generally not very user friendly, often requiring a long start-up period to learn to operate.

As noted above, the advent of on-chip caches is making it increasingly difficult to build trace collection hardware as an afterthought. The future of probe-based trace collection will therefore be determined mainly by the level of support *designed* into systems for this activity. A small, on-chip trace buffer that traps to the kernel whenever it becomes full is an example of the sort of support that could be provided. However, even a very small buffer of 2048 entries with 32-bits per entry (8 K-bytes) is about the size of on-chip caches in current microprocessors [Nagle94] and thus would be relatively costly in terms of chip area. We are not aware of any existing microprocessor that includes such hardware.

Microcode Modification

The high cost circuit-level probing has motivated many researchers to develop methods for collecting traces at higher levels of system abstraction. In this section we examine a form of trace collection that takes place at the microcode level, which is on the borderline between the hardware and software levels of a system (see Figure 2.2).

From the beginnings of the IBM 360 series (1964) until the DEC VAX machines, the most common method for implementing control logic was microcode. When implemented off-chip, a microcode memory is often writable or can be modified through replacement, making it possible to change the behavior of instructions, or to support multiple instruction sets. Agarwal realized that this mechanism makes it possible to collect address traces [Agarwal86; Agarwal88]. He modified the microcode on a VAX 8200 to cause all instructions to deposit the addresses of their memory references into a reserved area of main memory as a side effect of their execution.

This method, which Agarwal calls *address tracing using microcode (ATUM)*, offers a number of advantages. The first is completeness. Because the microcode runs “beneath” the operating system, all user and kernel references are captured, as well as those from dynamically-compiled

and dynamically-linked code. Because ATUM has access to internal system state, it is easily able to annotate traces with access-type tags, context switch points, and page-map information. Another advantage is speed. ATUM acquires address traces with a slowdown of only about 10 to 20, and because the addresses can be processed directly out of the trace buffer in main memory, there is not the same overhead for buffer unloading as with external probe-based trace collection. Finally, because no additional hardware is required, the only cost associated with ATUM is the engineering effort required to modify microcode to produce the desired results.

The ATUM method suffers from a few minor disadvantages and one major one. First, ATUM traces exhibit some discontinuity distortion because the processor is not stalled when the trace buffer becomes full. Buffer size can only be increased to a certain point because it takes away from the usable memory of the host system.[2] Agarwal has developed a method, called *trace stitchin*g, to treat this problem [Agarwal89]. Microcode modification also introduces another form of trace distortion, commonly called *time dilation*. Because instructions take 10 to 20 times as long to execute as they normally would, external events such as clock interrupts will occur more frequently. This causes more frequent invocations of the code that handles clock interrupts, disturbing TLB and cache state. More frequent clock interrupts can also cause scheduling quanta to expire more rapidly, resulting in changes in the behavior of the CPU scheduler. Another disadvantage is that the method captures only microcode-controlled activities, meaning that I/O activity and other memory transactions controlled by hardwired logic are not captured. The primary disadvantage of the microcode-modification technique is that it suffers from poor portability. Because most modern processors have a hardwired control, rather than being microprogrammed, this technique is no longer generally applicable [Hennessey90]. In such systems, trace collection is more conveniently performed at higher levels in the system. For example, RISC processors are typically implemented with hardwired control and are therefore not amenable to microcode modification. However, the relatively simple and uniform coding of RISC instruction sets has lead to the development of fast instruction-set emulators and binary-rewriting tools that instrument executables to produce traces as a side effect of their normal execution (see the following sections on *instruction-set emulation* and *code annotation*).

2. In Agarwal's implementation, the size was limited to 2 MB, which holds 400,000 memory references.

While most RISC machines no longer use microcode, some CISC machines as the x86 series from Intel still do. However, this microcode resides in an unmodifiable, on-chip ROM. This suggests that microcode modification may still be a viable method,[3] but as with probe-based monitoring, it can no longer be added to a processor as an afterthought; if future processors wish to take advantage of this technique, it must be *designed* in from the start.

Instruction-set Emulation

An instruction-set architecture (ISA) is the set of instructions that defines the interface between hardware and software for a particular machine. A microcode engine is an ISA interpreter that is implemented in hardware. It is also possible to interpret an instruction set in software through the use of an *instruction-set emulator*. Emulators typically execute one instruction set (the *target* ISA) in terms of another instruction set (the *host* ISA) and are usually used to enable software development for a machine that has not yet been built, or to ease the transition from an older ISA to a newer one [Sites92]. As with microcode, an instruction-set emulator can be modified to cause an emulated program to generate address traces as a side-effect of its execution.

Conventional wisdom holds that instruction-set emulation is very inefficient, with slowdowns estimated to be in the range of 1,000 to 10,000 [Agarwal89; Wall89; Borg89; Stunkel91; Flanagan92]. Clearly, the degree of slowdown is related to the level of emulation detail. For some applications, such as the verification of a processor logic design, the level of detail required is very high and the corresponding slowdowns may agree with those cited above. However, in the context of this review, we consider an instruction-set emulator to be sufficiently detailed for the purposes of address-trace collection if it produces an accessible stream of instruction and data references as part of its emulation. Given this minimal requirement, there are several recent examples of instruction-set emulators that have achieved slowdowns much lower than 1,000 (see Table 2.3).

Tracer [Henry83] and *Spa* [Cmelik93] are examples of slow instruction-set emulators. They work by reading, interpreting and dispatching instructions one at a time. Instructions are re-interpreted each time they are encountered. Reported slowdowns for these iterative emulators range from 500 to about 1,500.

3. Not all Pentium instructions come out of the microcode store, so such modifications could be challenging.

Reference	Name	Target(s)	Host(s)	Method	Multi-task	Kernel	Slowdown
[Henry83]	Tracer	VAX 11/780	VAX 11/780	Iterative Interpretation	No	No	1,500
[Cmelik93]	Spa	SPARC	SPARC	Iterative Interpretation	No	No	600
[Larus91]	SPIM	MIPS-I	SPARC, 680x0, MIPS, x86, HP-PA	Predecode to IR	No	Some	25
[Cmelik93]	Shadow	SPARC	SPARC	Dynamic Compilation	No	No	64
[Magnusson93]	gsim	88100	HP-PA, SPARC	Block Compilation	Yes	Some	35 - 45 50 - 74
[Veenstra94]	MINT	R3000	R3000	Block Compilation	Yes	No	18 - 65
[Bedicheck94]	mg88	88100	SPARC, 680x0, 88100	Threaded Code	Yes	Some	54 - 74
[Cmelik94]	Shade	SPARC-V8, SPARC-V9, MIPS	SPARC-V8	Dynamic Compilation	No	No	9 - 14

Table 2.3 Trace Collection through Instruction-set Emulation

We define an instruction-set emulator to be a program that directly reads executable images written in one ISA (the target) and that emulates it using another ISA (the host). In general, the target and host ISAs do not need to be the same, although they may be. We only consider instruction-set emulators that also generate address traces. For a more complete survey of instruction-set emulators in general, see [Cmelik94]. Some of these tools support multi-processor simulation. In these cases, slowdowns are given for a single process, but may include some un-removable overhead due to the multi-processor support.

The faster emulators use a variety of techniques to achieve their speed. In *SPIM*, a MIPS-I executable is read and translated, in its entirety, to an intermediate representation understood by the emulation engine [Larus91]. After translation, emulation slowdown relative to the host hardware is a factor of approximately 25. *Shadow* dynamically compiles only target instructions that are actually executed into a form that can be run directly by the host [Cmelik93].

A later generation of Shadow, named *Shade*, also uses dynamic compilation to host instructions, but translations are cached so that each instruction is compiled only once. In this system, as each instruction is referenced for the first time it is compiled into an efficient sequence of native instructions that runs directly on the host machine. Compiled sequences of native code are recorded in a lookup table, which is checked by a core emulation loop each time it dispatches a new instruction. If a compiled translation already exists, is it found through the lookup mechanism and the code sequence need not be recompiled. Both Shadow and Shade support address-trace processing by calling user-supplied “analyzer” code after each instruction is emulated. The analyzer code is given access to the emulation state, such addresses generated by the previous instruction, so that memory simulations are possible. The slowdowns reported in Table 2.3 are for Shadow and Shade emulations with a null analyzer.

Like Shadow and Shade, *MINT* [Veenstra94] and *gsim* [Magnusson93] also dynamically compile code into the sequences of host instructions that are then saved for future re-execution. But unlike Shade, which compiles one instruction at a time, these simulators compile clusters of several source instructions together. This strategy makes it possible to construct more efficient sequences of equivalent host instructions because less emulator state must be saved and restored between each translated instruction.

Each of the emulators described up to this point rely on a core emulation loop that is executed after each emulated instruction or block of instructions executes. The *mg88* emulator avoids this overhead by combining dynamic compilation with a technique called *threaded-code execution* that directly links multiple translations together. These links, from one translation to the next, avoid the need to return to the core execution loop until a new, untranslated instruction is encountered [Bedichek94].

These last three emulators (MINT, gsim and mg88) exhibit slowdowns as high as 65 to 75. However, this reflects the additional overhead required to generate multi-processor address traces. The reported slowdowns are in some cases higher than they need to be for address-trace collection

because they include additional emulation detail. For example, the mg88 emulator models instruction-execution times and includes full simulation of caches, a memory management unit and I/O devices [Bedichek94]. The gsim slowdowns reflect a similar overhead for more complete system simulation, but the MINT slowdowns are for emulation with address generation only.

With respect to trace completeness, most of these tools collect only references from a single task and exclude kernel references. Some of these tools claim to support multi-threaded applications and emulation of operating system code, but this statement should be interpreted carefully. All of these emulators run in their own user-level task and require the full support of a host operating system. Within this task, they create a *virtual system* with varying degrees of detail. This does *not* mean that these emulators are able to generate the address references made by the actual host operating system, nor are they able to see any references made by any other user-level tasks in the host system. For example, mg88, MINT and gsim, which are all designed for multi-processor simulation, intercept thread `fork()` calls made by the application and then emulate multi-processor execution by emulating a few instructions from each thread, in turn, according to some scheduling policy. Mg88 emulates the execution of operating system code by supporting the additional instructions in the ISA that require supervisor privileges, such as memory-management functions. Mg88 even simulates hardware I/O devices to make the virtual system complete. Some of these emulators (e.g., Shadow and Shade) are able to handle dynamically-compiled and dynamically-linked code.

With respect to trace detail, instruction-set emulation naturally produces virtual addresses, but is generally unable to determine the physical addresses to which these virtual addresses correspond. Unlike the probe-based methods, instruction-set emulation can avoid time-gap and time-dilation trace distortions because all aspects of the virtual system, including I/O devices, can be stalled.

Instruction-set emulators generally share the advantages of high portability, flexibility and ease of use. Several of the emulators, such as SPIM are written entirely in C, making ports to hosts of several different ISAs possible [Larus91]. Emulators with more machine-specific components, such as mg88, have nevertheless proven to be relatively easy to port to other hosts [Bedichek94]. Shade has been designed for easy portability to other target architectures, one of which had yet to be implemented at the time the paper was written [Cmelik93; Cmelik94]. In other words, instruction-set emulators like Shade can collect address traces from machines that have not yet

been realized in hardware. Some of these emulators are very flexible in the sense that the analyzer code can specify the level of trace detail required. For example, Shade analyzers can specify that only load data addresses in a specific address range should be traced [Cmelik94]. Ease of use is enhanced by the ability of these emulators to run directly on executable images created for the target architecture, with no prior preparation or instrumentation of workloads required.

A major disadvantage of instruction-set emulators is that they build up a large amount of state. Instructions that have been translated to an intermediate representation, or to equivalent host instructions, use from 4 to 20 times as much memory as compiled code [Cmelik94; Bedichek94]. Other auxiliary data structures, such as tables that accelerate the lookup of translated instructions, boost memory usage even higher. Actual measurements of memory usage are unavailable for most of the emulators in Table 2.3, but for Shade they are reported to be in the range of 4 to 40 times the usual memory required by normal, native execution [Cmelik93; Cmelik94]. Increased memory usage means that these systems need to be equipped with additional physical memory to handle large workloads.

Code Instrumentation

The fastest instruction-set emulators *dynamically* translate instructions in the target ISA to instructions in the host ISA. This translation may involve some additional instrumentation of the host code to produce address traces. Because these emulators perform translation at run time they gain some additional functionality, such as the ability to trace dynamically-linked or dynamically-compiled code. However, this additional flexibility comes at some cost, both in overall execution slowdown and in memory usage. For the purposes of trace collection, it is often acceptable to trade some flexibility for increased speed. For example, if the target and host ISAs are the same and if dynamically-changing code is not of interest, then a workload can be instrumented statically, before run time. Code instrumentation can be performed at the source (assembly) level, the object-module level, or the executable (binary) level (see Figure 2.2 and Table 2.4), with different consequences for both implementation and the end user [Stunkel91; Wall92; Pierce94a].

The main advantage of working at the source level is ease of implementation. The task of relocating code and data of the instrumented program can be handled by the usual assembly and link phases of a compiler, and more detailed information about program structure can be used to optimize code-instrumentation points. Unfortunately, instrumentation at this level often causes

Method	Reference	Name	Slowdown	Time Dilation	Memory Dilation	Completeness		Processor	Analyzer Interface
						Multi-Task	Kernel		
Source	[Stunkel89]	TRAPEDS	20 - 30	20 - 30	8 - 10	No	No	iPSC/2	Linked Into Task
	[Eggers90]	MPtrace	1,000 +	2 - 3	4 - 6	No	No	i386	File + Post Process
	[Larus90]	AE	20 - 65	2-5	—	No	No	MIPS, SPARC	File + Post Process
Object	[Borg89]	Epoxie	8 - 12	8 - 12	5	Yes	No[1]	Titan	Buffer Processor
	[Chen93a]	Epoxie2	15	15	2	Yes	Yes	R3000	Buffer Processor
Binary	[Smith91]	Pixie	10	10	4 - 6	No	No	MIPS	File / Pipe
	[Stephens91]	Goblin	20	20	10	No	No	RS/6000	Linked Into Task
	[Pierce94a]	IDtrace	12	12	12	No	No	i486	File / Pipe
	[Larus93]	Qpt	10 - 60	2-5	3	No	No	MIPS, SPARC	File + Post Process

Table 2.4 Code-instrumentation Tools

Code-instrumentation tools add instructions to a program at the source, object or binary level to cause them to output address traces as a side effect of their execution. In the above table, *Slowdown* refers to the time it takes to both run the instrumented program and to produce the full address trace, while *Time Dilation* refers only to the time it takes to run the instrumented program. Usually these are the same, but some instrumented programs generate only a minimal trace of significant events which must be post-processed to reconstruct the full trace. *Memory dilation* refers to the additional space used by the instrumented program relative to an uninstrumented program.

Notes: [1] Kernel tracing was implemented, but was not fully debugged. [2] Data-only address traces.

problems for the end user because the complete source code for a workload of interest is often not available. This is especially true when the workload links object modules from a standard code library. An early example of code instrumentation performed at the source level is the *TRAPEDS* system [Stunkel89]. TRAPEDS adds trace-collecting code and a call to an analyzer routine at the end of each basic block in an assembly source file. The resulting program expands in size by a factor of about 8 to 10, and its execution is slowed by about 20 to 30. Other systems take better advantage of the additional information about program structure available at the source level. Both *MPtrace* and *AE* use control-flow analysis to annotate programs in a minimal way so that they only produce a trace of significant dynamic events [Eggers90; Larus90]. The resulting significant-event traces are combined with a post-processor that reconstructs the full trace given the control-flow graph of the original program. This method reduces both the size and execution time of the instrumented program. Programs instrumented by MPtrace are only about 4 to 6 times larger than usual, and exhibit slowdown of only 2 to 3, not including the time to regenerate the full trace. Eggers et al. argue that it is useful to postpone full-trace reconstruction until after the workload runs because this minimizes the effects of time dilation. It is important to include the time to regenerate the full address trace when considering the speed of these methods. In the case of AE, trace regeneration brings overall slowdowns up to about 20 to 60. Unfortunately, the trace-regeneration time is not given in terms of slowdowns for MPtrace, although Eggers et al. do report that trace regeneration is the most time-consuming step in their system, producing only 6,000 addresses per second. Assuming a processor that generates 6 million memory references per second (a conservative estimate for machine speeds at the time the paper was written), 6,000 addresses per second corresponds to a slowdown of approximately 1,000.

Performing instrumentation at the object-module level can help to simplify the preparation of a workload program. In particular, source code for library object modules is no longer needed. Wall argues that instrumenting code at this level is only slightly more difficult because data-relocation tables and symbol tables are still available [Wall92]. An early example of this form of code instrumentation is *Epoxie*, implemented for the DEC Titan [Borg89; Borg90; Mogul91], and later ported to MIPS-based DECstations [Chen93a]. In both of these systems, slowdowns for the annotated programs ranged from about 8 to 15 and code expansion ranges from 2 to 5.

Code instrumentation at the executable level provides the highest level of convenience to the end user because it is not necessary to instrument a collection of source and/or object files to produce the final, instrumented program. Instead, a single command applied to one executable

produces the desired instrumented code. Unfortunately instrumentation at this level is also the most difficult to implement because executable programs are often stripped of symbol-table information. A significant amount of analysis may be required to properly relocate code and data after trace-generating instructions have been added to the program [Pierce94a]. Despite these difficulties, there exist several program instrumentation tools that operate at the executable level. An early example is *Pixie*, which operates on MIPS executables [MIPS88; Smith91]. Pixie expands program sizes by about 4 to 6 on average and slows program execution by about a factor of 10. The popularity of Pixie has prompted the development of several similar programs that work on other instruction-set architectures. These include *Goblin* and *IDtrace* which operate on RS/6000 and i486 binaries, respectively [Stephens91; Pierce94a]. Our measurements of Goblin show that it exhibits code expansion factors of about 10 and slowdowns of about 20 for instruction traces only. This is in disagreement with values reported in [Stephens91], possibly because of a different selection of workloads. IDtrace, which generates complete instruction and data traces, expands a program by a factor of 12 over the original and execution slowdowns are also about 12. A second generation of the AE tool, called *Qpt*, can operate on both MIPS and SPARC binaries [Larus93]. Code expansion is not reported in this paper, but slowdowns are given to be 2 to 5 for generation of the significant-events trace, with an overall slowdown of 10 to 60 when including the time to regenerate full traces.

In general, these tools are not capable of instrumenting multi-task workloads or the operating system kernel, but there are some exceptions. Borg and Mogul describe modifications to the Titan operating system, Tunix, that support tracing of multiple workload tasks by Epoxie [Borg89; Borg90; Mogul91]. Tunix interleaves the traces generated by multiple tasks into a global trace buffer that is periodically emptied by a trace-processing program. These researchers also experimented with instrumenting the Tunix kernel itself, although they do not report any results obtained from these traces [Mogul91]. Chen continued this work by porting a version of epoxie to a MIPS-based DECstation running Ultrix and Mach 3.0 to produce traces from single-task workloads including the user-level X and BSD servers, and the kernel itself [Chen93a; Chen93c].

As a rule, static code instrumentation cannot handle code that is dynamically compiled at run time, for the obvious reasons. Dynamically-linked code also poses a problem although some systems, such as Chen’s, treat this problem in special cases. For example, Chen modified the BSD server to cause it to dynamically map a special instrumented version of the BSD emulation library into user-level tasks that require a BSD API.

With respect to trace detail, these methods naturally produce virtual addresses tagged by access type and size. Some of the systems that can annotate multi-task workloads are also able to tag references with a task identifier [Borg89]. However, associating a true physical address with each virtual address is essentially impossible because an instrumented program is expanded in size and therefore utilizes virtual memory very differently than an uninstrumented workload would.

The tools that include multi-task and kernel references are subject to several forms of trace distortion. Trace discontinuities occur when the trace buffer is processed or saved to disk and time-dilation distortion occurs because the instrumented programs run 10 to 30 times slower than they normally would. Chen and Borg et al. note that the effects of these distortions on clock-interrupt frequency and the CPU scheduler can be adjusted for by reprogramming the clock-generation chip [Borg89; Chen93a]. However, a solution to the problem of apparent I/O device speedup is not discussed. Borg et al. discuss a third form of trace distortion due to instrumented-code expansion called *memory dilation*. This effect can lead to increased TLB misses and paging activity. The impact of these effects can be minimized by adding additional memory to the system (to avoid paging), and to emulate, rather than instrument, the TLB miss handlers (to factor out increased TLB misses) [Borg89; Chen93a].

These tools share a number of common characteristics. First, they are on average about twice as fast as instruction-set emulation techniques. Note, however, that some of these tools are outperformed by very efficient emulators, like Shade. Second, all of these tools suffer from the disadvantage that all workload components must be prepared prior to being run. Usually this is not a major concern, but it can be a time consuming and tedious process if a workload consists of several source or object files. Even for the tools that avoid source or object-file instrumentation, it can be difficult to locate all of the executables that make up a complex multi-task workload like `sdet` from the SPEC SDM suite [SPEC93]. Portability is generally high for the source-level tools, such as AE, but decreases as code modification is postponed until later stages of the compilation process. Portability is hampered somewhat in the case of Chen's system, where several workload components in the kernel must be instrumented in assembly code by hand. Note that static instrumentation must annotate all the code in a program, whether it actually executes or not. This is not the case with the instruction-set emulators which only need to translate code that is actually used. This is an important consideration for very large executables, like X applications, that are often larger than a megabyte, but only touch a fraction of their text segment [Chen94a]. These additional memory requirements add to the expense of these methods.

Single-step Execution

Addresses can also be collected at the highest level of system abstraction shown in Figure 2.2: the operating system. Most operating systems support some form of debugging utility that enables a programmer to step through a program one instruction at a time to expose errors. This form of debugging is usually supported in hardware through a single-step execution mode, where the processor traps into the OS kernel after the execution of each instruction or basic block [Digital86; AMD91; AMD93; Motorola93; HP90; Motorola90b] or by breakpoint instructions that cause kernel traps whenever they are executed [Kane92; Intel90]. A debugger that supports single-step execution and examination of processor state, such as registers, can be modified to generate both instruction-address and data-address traces. Instruction-address traces are produced by simply recording the value of the program counter at each execution step. Data-address traces require instruction emulation to determine if the current instruction generates a memory reference and, if so, the value of that reference. Examples of studies that use traces obtained through single-stepping include [Wiecek82; Clark85; Winsor89].

The main advantages of this method are low expense, high portability, and ease of use. With the exception of debugger data structures, little additional host memory is used. Reported slowdowns for this technique vary widely from 100 [Agarwal88] to 1,000 [Flanagan92] to 10,000 [Holliday91]. High slowdowns are usually due to debugger implementations that rely on the UNIX `ptrace()` facility which, in turn, is implemented using UNIX exception-signal handlers. Recent work on tuning the exception-delivery path in UNIX-based systems suggests that these slowdowns could be cut dramatically [Thekkath94]. Assuming the support of a very efficient exception delivery mechanism, slowdowns could be brought as low as 50, but probably not much lower.

Although there is nothing inherent in this approach that limits traces to a single task, or to user-only references, debuggers typically do impose these limitations. Similarly, dynamically-compiled and dynamically-linked code is usually not supported by debuggers. Because only address-trace information is desired, a single-step trace-collection tool could, in principle, be written from scratch to avoid the overheads and single-process limitations of program debuggers. We are not aware of any existing trace-collection system that uses this approach.

Although once very popular [Holliday91], single-step execution as a method for trace collection has essentially been abandoned in recent years because of the greater efficiency of other software-based methods.

Summary of Trace-collection Methods

Table 2.1 summarizes the general characteristics of each of the trace-collection methods examined in this section. Because of the range of capabilities of tools within each category and because of the subjective nature of some of the characteristics (e.g., portability), it is difficult to accurately and fairly summarize all considerations in a single table. Nevertheless, it is worthwhile to attempt to do so, so that some general conclusions may be drawn.

For descriptions of trace quality (*completeness*, *detail* and *distortion*), a *Yes* entry means that most existing implementations of the method naturally provide trace data with the given characteristics. A *Maybe* entry means that the method does not easily provide this form of trace data, but there are nevertheless a few existing tools that overcome these limitations. A *No* entry means that there are no existing examples of a tool in the given category that provide trace data of the type in question, usually because the method makes it difficult to do so.

To make the comparisons fair, trace-collection slowdowns include any additional overhead required to produce a complete, usable address trace. This may include the time required to unload an external trace buffer (in the case of the probe-based methods), or to regenerate a complete address trace from a significant-events file (in the case of certain code-instrumentation methods). Slowdowns do not include the time required to process the trace, nor the time to save it to a secondary storage device. We give a range of slowdowns for each method, removing any excessively bad implementations in any category.

Additional *Memory* requirements refer to external trace buffers, or physical memory reserved by the host that is consumed either by trace data or by a workload expanded in size due to instrumentation. Factors that determine the *Expense* of the method include the purchase of special monitoring hardware, or any necessary modifications to the host hardware, such as changes to the motherboard to make CPU pins accessible by external probes, or the purchase of extra physical memory for the host to satisfy the memory requirements of the method. *Portability* is determined both by the ease with which the tool can be moved to other machines of the same type, and to machines that are architecturally different. Finally, *Ease-of-Use* describes the amount of effort

required of the end user to operate the tool once it has been developed. These last few characteristics require a somewhat subjective evaluation which we provide with a rough *High*, *Medium*, or *Low* ranking.

Despite all of these qualifications, it is possible to draw some general conclusions about how close actual trace-collection methods come to the ideal. A first observation is that high-quality traces are still quite difficult to obtain. Methods that by their nature produce relatively complete, detailed and undistorted traces (e.g., the probe-based or microcode-based techniques) are either very expensive, hard to port, hard to use or outdated. On the other hand, the techniques that are less expensive and easier to use and port (e.g., instruction-set emulation and code instrumentation) generally have to fight inherent limitations in the quality of traces that they can collect, particularly with respect to completeness (multi-task and kernel references). Second, none of the methods are able to collect complete traces with a slowdown less than about 10. Finally, when all the factors are considered, no single method for trace collection is a clear winner, although some, such as single-step execution, have clearly dropped out of favor. The probe-based and microcode-based methods probably produce the highest quality traces as measured by completeness, detail and distortion, but their future is in some jeopardy if designers fail to provide certain types of hardware support or greater accessibility in future machines. Code instrumentation is probably the most popular form of trace collection because of its low cost, relatively high speed, and because of recent developments that enable it to collect multi-task and kernel references. However, advances in instruction-set emulation speeds and the greater flexibility of this method may lead to the increased use of this alternative to code instrumentation.

2.2.2 Trace Reduction

Once an address trace has been collected, it is input to a trace-processing simulator or stored away on disk or tape for processing at a later time. There has been considerable interest in finding ways to reduce the enormous size of address traces in order to minimize both processing and storage requirements.[4] Because address traces exhibit high spatial and temporal locality, there are many opportunities for achieving high factors of trace reduction. In a study by Pleszkun, the information content (entropy) of address traces was shown to be very low [Pleszkun94].

4. A modern uniprocessor operating at 100 MHz can easily produce half a Gigabyte of address trace data every second.

There are several points of interest when evaluating and comparing different methods for trace reduction (see Table 2.5). The first, of course, is the factor of trace compression. The time required to reconstruct or decompress a trace is also important because it directly affects simulation times. Given that a reduced trace is eventually used to drive a memory simulator, it is important to know whether or not the resulting simulations produce results identical to simulations with a full trace. If results are not exact, the amount of error and its relationship to the parameters of the memory structure being simulated are of interest. Finally, many of the trace reduction methods make assumptions about the type of memory simulation that will be performed using the reduced trace. These assumptions usually lead to restrictions in the use of the reduced trace, and should be understood as well.

One approach to trace reduction is to apply standard data compression algorithms. For example, the UNIX compress utility, which implements the Lempel-Ziv algorithm [Ziv76], achieves a compression factor of about 3 to 5 on typical address traces [Agarwal90]. Samples has shown that much higher degrees of compression can be attained if the full address trace is first pre-processed to produce a stream of address differences [Samples89]. When the resulting *difference trace* is input to the same Lempel-Ziv compression algorithm, the compression factors increase to 10 to 20, for traces with mixed instruction and data references, and to as high as 100 for traces with only instruction references [Sample89]. This system, called *Mache*, reproduces the complete address trace, so resulting simulations are unrestricted and exact. However, because the full address trace must be reconstructed before simulation, there is a space, but not a simulation-time savings. In fact, Samples reported times imply that decompression can add a slowdown factor of as much as 200 to trace-driven simulations.

Another method for trace reduction, discussed in the previous section, is to instrument programs so that they produce traces of only significant dynamic events [Larus90; Eggers90]. The resulting significant-events file is much smaller than the complete trace, but can be post-processed, along with additional information (such as the control-flow graph), to produce a complete trace. One such tool, *AE*, achieves compression factors of 10 to 40. As with Mache, the complete trace is regenerated so simulations using these traces are unrestricted and exact, but there is no simulation-time savings. Trace reconstruction slowdowns range from 20 to 60 with AE [Larus90], and are over 1,000 with *MPtrace* [Eggers90].

Method	Reference	Reduction Factor	Decompress Time	Simulation Speedup	Exact?	Error	Restrictions
Data Compression	[Samples89]	10 - 100	100 - 200	0	Yes	N/A	None
Abstract Execution	[Larus90]	10 - 40	20 - 60	0	Yes	N/A	None
	[Eggers90]	—	1,000 +	0	Yes	N/A	None
Stack Deletion	[Smith77]	5 - 100	0	4 - 50	No	< 4 - 5%	Fully-associative Memories
Snapshot Method	[Smith77]	5 - 100	0	4 - 50	No	< 4 - 5%	Fully-associative Memories
Cache Filter	[Puzak85]	10 - 20	0	—	Yes	N/A	Fixed-line-size Caches
	[Wang90]	10 - 20	0	7 - 15	Yes	N/A	Fixed-line-size Caches
Block Filter	[Agarwal90]	50 - 100	0	—	No	< 12%	Fixed-line-size Caches
Time Sampling	[Laha88]	5 - 20	0	5 - 20	No	< 5%	Small Caches (< 128 K-byte)
	[Kessler91]	10	0	10	No	< 10%	Small Caches (< 1 M-byte)
Set Sampling	[Puzak85]	5 - 10	0	10	No	< 2%	Set Sample Not General
	[Kessler91]	10	0	10	No	< 10%	Constant-bits Set Sample

Table 2.5 Address Trace Reduction Methods

The trace reduction factor is the ratio in size between the reduced trace and the full trace. *Decompress Time* is only relevant to methods that reconstruct the full trace before it is processed. Most of these methods pass the reduced trace directly to the trace processor which is able to process this data much faster than the full trace (see *Simulation Speedup*). Usually, simulations with a reduced trace result in some simulation error and can only be performed in a restricted design space (see *Exact*, *Error* and *Restrictions*).

Often a designer has a specific purpose in mind for a given set of address traces. For example, the traces might be used only for cache simulations where the cache size is larger than some specific minimum and the line size is fixed. In a situation such as this, a full address trace can be substantially reduced in size, provided that the resulting reduced trace is only used for simulations in an appropriately-constrained design space. Smith has suggested two examples of this form of trace reduction [Smith77]. He constrained his simulation design space to fully-associative memory structures (for main-memory page-replacement or TLB simulations), and then devised two methods for trace reduction: *stack deletion* and *the snapshot method*. *Stack deletion* applies a full memory trace to a simulation of an LRU stack memory. Addresses that hit in the top D entries of the stack are discarded, while addresses that miss are concatenated to form a reduced trace. The rationale for this method is that references that hit the LRU stack are also likely to hit in any fully-associative main memory or TLB that is larger in size. The *snapshot method* constructs a reduced trace by concatenating snapshots of memory contents taken at periodic intervals separate by T, the snapshot parameter. Smith points out that such a trace could be acquired at the full speed of a real machine by periodically interrupting execution and recording the contents of page reference bits [Prieve74]. The rationale for this method is similar to that of the stack deletion method; the memory snapshots capture the most important references, while filtering out repeated references to the same location. Depending on the values of the deletion parameter, D, and the snapshot interval, T, Smith reports that trace-size reductions range from a factor of 5 to 100. When Smith used these reduced traces for the simulation of various page-replacement algorithms and compared the results against simulations with full traces, he found the relative error to be less than 5%. An advantage of these methods over those previously discussed is that the reduced trace can be used directly by the simulator. This means that there is no decompression overhead and the resulting simulations are much faster than they would be on a complete address trace. Note, however, that the simulation speedups (4x-50x) are not directly proportional to the compression factors (5x-100x). This is because simulations with the reduced trace result in more misses per trace event than with simulations on the full trace. Because processing misses usually requires more time than processing hits, simulations on the reduced trace take more time, per trace event, than they do on the full trace.

Trace stripping, first suggested by Puzak in his dissertation [Puzak85], is another method for constructing reduced traces that can only be used in a restricted design space. This method applies the full address trace to the simulation of a small, direct-mapped cache with some given line size. Only the references that miss this *filter cache* are saved to form the reduced trace. Puzak proved

that the trace of misses can be used to perform exact simulations of any cache with greater size or associativity than that of the filter cache, provided that the line size is held constant. When simulating line sizes different than that of the filter cache, Puzak points out that some simulation error does result, but it is generally less than 10% and decreases with increasing cache associativity.

Wang and Baer extended the cache filter concept to enable the simulation of write-back caches [Wang90]. Their cache filter is the same as Puzak's, but in addition to recording all read misses, their reduced trace also includes the first write to any clean cache line. With both of these methods, the trace reduction factor is equal to the inverse of the cache miss ratio. Assuming miss ratios of 0.05 to 0.10 for small direct-mapped caches, slowdowns would be in the range of 10 to 20. As with Smith's methods, the simulation speedups are not directly proportional the trace-reduction factor.

Agarwal and Huffman have pointed out that cache filters exploit only temporal, but not spatial locality in address traces [Agarwal90]. They have devised another form of trace filter, called a *block filter*, that provides an additional order-of-magnitude reduction of trace size, beyond that of a cache-filtered trace. A block filter takes as input a cache-filtered trace and two other parameters called the window size, W, and the block size, B. The filter reads a group of W references at a time and outputs only a single reference from each *spatial locality* in the window. Two addresses are defined to belong to the same spatial locality if they refer to the same block of B addresses. The rationale for constructing the reduced trace in this way is based on the theory of stratified sampling [Hodges64], where the strata correspond to spatial localities. Agarwal and Huffman show that application of the block filter can increase trace reduction factors to as high as 100, while keeping the error in simulation results under 10% to 12%.

Researchers have investigated other forms of sample-based trace reduction. One approach is to collect samples over time (*time sampling* [Laha88]), while another is to collect samples over space (*set sampling* [Puzak85]). Each is discussed in greater detail below.

Laha et al. collected different segments of a full trace that were each contiguous in time [Laha88]. Each of these trace segments was driven into a memory simulator to obtain an estimate of some performance metric, such as a miss ratio. The miss-ratio estimators from each trace segment were combined to form an average estimate of performance over the length of the entire trace. This method, called *time sampling*, must be conducted with care to avoid errors. First, the individual trace segments must be long enough to overcome a form of bias caused by starting the

simulation of each segment in a cold (empty) memory. Second, a sufficient number of trace segments must be collected (Laha et al. suggest 35) to ensure that different phases of execution are adequately represented. This study showed that time samples representing 5% to 20% of the full trace can be used to simulate caches with less than about 5% relative error. The simulations were limited to relatively small caches (< 128 K-bytes) to minimize errors due to cold-start bias.

Puzak proposed an orthogonal trace sampling approach, called *set sampling* (or congruence-class sampling) [Puzak85]. With set sampling, the reduced trace is constructed by keeping exactly those addresses that reference a certain random collection of cache sets. References to any other sets are discarded. Cache simulations are performed on each sampled set individually to obtain several estimates of some performance metric. Then, as with time sampling, the estimators are combined to form an overall estimate of cache performance. Because each set in the sample sees all of the references made to it by the full trace, this method does not suffer from cold-start bias. Puzak showed that set samples representing 10% to 20% of the full trace produce simulation results with less than 2% error with 90% confidence. He also showed that error decreases with increasing cache associativity. A disadvantage of Puzak's random set sampling is that a set sample for a given cache configuration will not necessarily be a set sample for other cache configurations. This implies that a different set sample may have to be constructed for each different cache configuration simulated.

Kessler has applied both time sampling and set sampling to the problem of simulating multi-megabyte secondary cache memories [Kessler91]. He developed and evaluated several improvements to both methods, including techniques for minimizing the effects of cold-start bias and a method for carefully constructing a single set sample for the simulation of several different cache configurations. He also compared the ability of the two different methods to meet a goal of 10% sampling that produces simulation results that are less than 10% in error, with 90% confidence. He showed that set sampling is able to satisfy this goal for large caches (greater than one megabyte), but time sampling breaks down in this range.

Some of these trace reduction methods can be combined to produce multiplicative improvements in compression factors. For example, a cache-filtered trace could also be time or set sampled. Similarly, standard data-compression algorithms can be applied to just about any trace reduced by the other methods, although the resulting compression factors are likely to be less than they would be on a full trace where the initial entropy is lower.

In summary, the most appropriate trace reduction method often depends on the questions to be answered by the simulation study. Because many of the methods restrict the way that the resulting reduced trace may be used, no single method is always best. A designer must first decide on the memory design space to be explored and then select a method depending on the simulation speed and accuracy required. If fast and exact simulation results are required, the best trace-reduction methods are limited to size-reduction factors of about 10. If speed is not a concern, but exact results are necessary, then tools like Mache or AE provide good solutions with size-reduction factors as high as 100, but trace reconstruction times can slow simulations by as much as 50 to 200. If simulation errors of 10% or less are considered acceptable, then the best methods (perhaps used in combination) can achieve space and time reduction factors of as high as 50 to 100.

2.2.3 Trace Processing

The final phase of trace-driven simulation, *trace processing*, often requires the most processing time. The objective of trace processing is to estimate the performance of a range of memory configurations by simulating their behavior in response to the memory references contained in an input trace.

This process is time consuming because a designer is typically interested in hundreds or thousands of different memory configurations in some design space. Consider that a design space for a simple cache defined by sizes ranging from 4 K-bytes to 64 K-bytes (in powers of two), line sizes ranging from 1 word to 16 words (in powers of two), and associativities ranging from 1-way to 4-way, contains 100 possible cache designs. The design space becomes very large when adding different replacement policies (LRU, FIFO, Random), different set-indexing methods (virtually- or physically-indexed) and different write policies (write-back, write-through, write-allocate). These design options are for a single cache, but actual memory systems are typically composed of multiple caches that cooperate and interact in a multi-level hierarchy. Because of these interactions and because different memory components often compete for scarce resources such as chip-die area, the different components cannot be considered in isolation. This leads to a further, combinatorial expansion of the design space.

One approach to this problem is to simulate multiple memory configurations in a single pass of the address trace (see Table 2.6). It is desirable that the simulator be able to vary parameters along several dimensions and also be able to produce any of several different metrics for

Reference	Name	Range of Parameters					Metrics	Overhead
		Sets	Line	Assoc	Write Policy	Sector		
[Mattson70]	Stack Simulation	Fixed	Fixed	Vary	None	No	Misses	—
[Hill87]	Forest Simulation	Vary	Fixed	1-way	None	No	Misses	< 5%
[Hill87]	All-Associativity	Vary	Fixed	Vary	None	No	Misses	< 30%
[Thompson89]	—	Fixed	Fixed	Vary	W-back	Yes	Misses, Write Backs	< 100%
[Wang90]	—	Vary	Fixed	Vary	W-back	No	Misses, Write Backs	< 65%
[Sugumar93]	Cheetah	Fixed	Vary	1-way	W-thru	No	Misses, WB Stalls	< 120%

Table 2.6 Multi-configuration Memory Simulators

Multi-configuration memory simulators can determine the performance for a range of memory configurations in a single pass of an address trace. However, each of these simulators is limited in the way that memory-configuration parameters can be varied (see *Range of Parameters*). Total cache size is determined by the equation: *Size = Sets * Assoc * Line.*

Overhead is the extra time it takes to perform a multi-configuration simulation relative to a single-configuration simulation (as reported by the authors of each simulator). This overhead is usually an underestimate of the true processing overhead because values reported in papers typically include the time to read input traces from a file.

performance, such as miss (hit) ratios, transfer ratios, misses per instruction (MPI) and cycles per instruction (CPI).

Mattson, Gecsei, Slutz and Traiger were the first to develop trace-driven memory simulation algorithms that are able to consider multiple configurations in a single pass of an address trace [Mattson70]. In their original paper they introduce a method, called *stack processing*, that determines the number of memory references that hit in any size of fully-associative memory that uses a *stack algorithm* for replacement. They give several examples of stack replacement algorithms, including LRU and OPT. They also note that some replacement policies, such as FIFO, are not stack algorithms. In their original paper, and in a collection of other follow-on reports (see [Sugumar93] or [Thompson89] for a more complete description), Mattson et al. described extensions to the basic stack algorithm to handle different numbers of cache sets, lines sizes and associativities.

In their early work, Mattson et al. did not report on the efficiency of actual implementations of their multi-configuration simulation algorithms. We are interested in the overhead of performing a multi-configuration simulation relative to a single-configuration simulation. This value lets us compute the average simulation speedup for a range of cache configurations relative to the time that would normally be required by several single-configurations simulations. Many researchers have advanced multi-configuration simulation by proposing various enhancements and by reporting simulation times for actual implementations of these improvements. We focus on a collection of recent papers that exhibit the range of parameters that multi-configuration methods have been extended to, and that characterize the current state-of-the-art in this form of simulation (see Table 2.6).

Hill noted that the original stack algorithm requires that the number of cache sets and the line size be fixed [Hill87]. This means that a single simulation run can only explore larger caches through higher degrees of associativity. Hill argues that designers are often more interested in fixing the cache associativity and varying the number of sets, a form of multi-configuration simulation that his *forest simulation* algorithm supports. Another algorithm studied by Hill is *all-associativity simulation* which enables both the number of sets and the associativity to be varied with just slightly more overhead than forest simulation. Thompson and Smith developed extensions that count the number of writes to main memory for different-sized caches that implement a write-back write policy [Thompson89]. They also studied multi-configuration

algorithms for sector or sub-block caches. Wang and Baer combined the work of [Mattson70], [Hill89] and [Thompson89] to compute both miss ratios and write backs in a range of caches where the both the number of sets and the associativity is varied. In his dissertation, Sugumar developed algorithms for varying line size with direct-mapped caches of a fixed size, and also for computing write-through stalls and write traffic in a cache with a coalescing write buffer [Sugumar93].

There are several points to be made about multi-configuration algorithms in general. First, for all of the examples considered, the overhead of simulating multiple configurations in one trace pass is reported to be less than 100%, which means that one multi-configuration simulation of two or more configurations would perform as well as or better than collections of two or more single-configuration simulations. However, these results should be interpreted with care because these overheads are reported relative to the time to read *and* to process traces. When the time to read an input trace is high, as is often the case when the trace comes from a file, the overhead of multi-configuration is very low. However, if the trace input times are relatively low, then the multi-configuration overheads will be much higher. This is the case with the Sugumar's *Cheetah* simulator which appears to have very high overheads relative to Hill's *Tycho* simulator [Hill87; Sugumar93] (see Table 2.6). Cheetah, however, is approximately eight times faster than Tycho because its input processing is more optimized [Sugumar93].

A second point is that even though multiple configurations can be simulated with one trace pass, it is often still necessary to re-apply multi-configuration algorithms several times to cover an entire design space. Hill gives an example design space of 24 caches, with a range of sizes, line sizes and associativities where the minimal number of trace passes required by stack simulation is 15 [Hill87]. For the same example, forest simulation still requires 3 separate passes and can only cover half of the space. All-associativity simulation also requires 3 separate passes, but covers the entire design space.

Finally, despite many advances in multi-configuration simulation, there are many types of memory systems and performance metrics that cannot be evaluated in a single trace pass. For example, most of these simulators restrict replacement policies to LRU, which is rarely implemented in actual hardware. Similarly, performance metrics that require very careful accounting of clock cycles generally cannot be computed for a range of configurations in a single simulation pass. For example, simulating contention for a second-level cache between split

Method	Buffer Capacity	Bandwidth	Slowdown	Same Task?	Typical Application
Files	G-bytes	1 MB / sec	100	No	Probe-based Microcode Modification
Pipes	K-bytes	5 MB / sec	20	No	Code Instrumentation
Memory	K-bytes	10 MB / sec	10	Yes	ISA Emulation Code Instrumentation
Procedure Call	None	—	—	Yes	ISA Emulation Code Instrumentation

Table 2.7 Some Trace Interfacing Methods

These estimates of bandwidth are based on measurements performed on a DECstation 5000/133 with a 33 MHz processor and a SCSI-connected disk. The file bandwidth is for a first read of the file from disk (as opposed to second reads which may be held in a file-block cache). Memory bandwidth was measured for a block of *uncached* reads and writes to simulate the interaction between a trace collector and trace processor that share the same address space. If these components can cooperate in such a way that the reads and writes are cached, bandwidth increases to about 40 MB/sec. Slowdowns were computed by assuming that this machine generates address references at an approximate rate of 100 MB/sec.

primary I- and D-caches requires a careful accounting of exactly when cache misses occur in each cache.

2.2.4 Trace Interfaces

Until now, we have only examined the three components of trace-driven simulation in isolation. Figure 2.1 suggests a natural composition of the three components in which they communicate through a simple linear interface of streaming addresses that may or may not include some form of buffering between the components. Because of the high data rates required, the selection of mechanisms used to communicate and buffer trace data is crucial to the overall speed of a trace-driven system. A bottleneck anywhere along the path from trace collection to trace processing can lead to high overall slowdowns. In this section we examine the pros and cons of the most-commonly-used methods (see Table 2.7).

Files

Because address traces conform to a simple linear-data-stream model, there are several options available for communicating and buffering them. The simplest and most commonly-used mechanism is the OS filesystem. *Files* provide deep buffering capability because they are implemented with secondary storage devices. As a result, files enable the postponement of trace processing as well as the ability to replay the same traces over again. This helps both to reproduce simulation results and to avoid the costs of re-collecting address traces. Files are most commonly used by probe-based trace collectors to permanently store the contents of their external trace buffers. Once a long trace of multiple samples is collected in a single file, a trace processor need not know the details of some clumsy interface to a specialized hardware trace buffer; the trace is simply accessed through a standard file read.

Files have some important disadvantages, the first of which is speed. Assuming a file-read bandwidth of 1 MB/sec and an address-generation rate of 100 MB/sec by the host (see the caption of Table 2.7), trace-driven simulation slowdowns are at least 100. Note that this still compares favorably to re-collecting traces with probe-based methods where slowdowns are in the thousands. Another disadvantage with files is that they are simply never large enough. Assuming again a host address-generation rate of 100 MB/sec, a 1.2 Giga-byte hard disk would be filled to capacity in about 12 seconds of real-time execution. This underscores the importance of the trace-reduction methods, described in Section 2.2.2, which can improve effective file capacity and bandwidth by one to two orders of magnitude.

Pipes

Pipes are another abstraction provided by the filesystem that can, under certain circumstances, overcome the limitations of files. Pipes establish a one-way channel for the flow of sequential data from one task to another. Usually only a moderate amount of memory, on the order of kilobytes, is provided to buffer the data flowing between the two tasks. This implies that both a trace collector (producer) and trace processor (consumer) must be running at the same time to prevent buffer overflow. However, in this approach, which is often called *on-the-fly simulation*, traces are discarded just after they are processed. Because traces need to be re-collected for each new simulation run, this technique is most effective when the trace collector is able to produce traces faster than can be read from a file. In the case of ISA emulators and code annotation tools, where

slowdowns range from 10 to 75, this requirement is often met. When pipes are used, trace-reduction methods are usually less attractive because they must be re-applied during each simulation run. This typically provides little or no advantage over simply processing the full address trace.

Same-task Communication

Both files and pipes are inter-process communication mechanisms provided by a filesystem. As such, their use incurs a certain amount of operating system overhead for copying or mapping data from one address space to another, and from context switching between tasks. These overheads can be avoided if a trace collector and trace processor run in the same task and arrange communication and buffering without the assistance of the OS. Several of the instruction-set emulation and code-annotation tools support trace collection and trace processing in the same task address space (see Table 2.4). In these systems, two different approaches to communicating and buffering trace data are commonly used. The first method is to make a *procedure call* to the trace processor after each memory reference. In this case, trace collection and processing are very tightly coupled and thus no trace buffering is required. A disadvantage is that procedure-call overhead, such as register saving and restoring, must be paid after each memory reference. With the second method, a region of *memory* in a task's address space is reserved to hold trace data. Execution begins in a trace-collecting mode which continues until the trace buffer fills and then switches to a trace-processing mode which runs until the trace buffer is again empty. By switching back and forth between these two modes infrequently, this method helps to amortize the cost of switches over many addresses.

2.2.5 Overall Trace-driven Simulation Slowdowns

Because of the variety of trace-driven simulation techniques and the ways to interconnect them, it is difficult to determine overall trace-driven simulation slowdowns. Unfortunately, very few papers report overall slowdowns because most tend to focus on just one component or aspect of trace-driven simulation, such as trace collection. Researchers that do assemble complete trace-driven simulation environments tend to report the results, not the speed of their simulations. There are, however, a few exceptions, which we summarize in this section and augment with our own measurements. Table 2.8 lists several complete trace-driven simulation environments that are

Name	Reference	Trace Collection	Trace Reduction	Trace Processing	Interface Method	Slowdown	Effective Slowdown
Epoxie + Panama	[Borg89]	Code Instr	None	Single Config	Memory	100	100
Meerkat	[Bedichek94]	ISA Emul	None	Single Config	Proc Call	150	150
Monster + Cheetah	—	Probe-based	Time Sample	Multi (8)	File	419	52
Pixie + Cheetah	[Sugumar93]*	Code Instr	None	Multi (44)	Pipe	183	4
Pixie + Tycho	[Gee93]	Code Instr	None	Multi (44)	Pipe	6250	142
Pixie + Cache2000	[MIPS88]*	Code Instr	None	Single Config	Pipe	60 - 80	60 - 80

Table 2.8 Overall Trace-driven Simulation Times

This table gives some typical slowdowns for a complete trace-driven simulation system. The number of configurations considered in a single pass of the trace are given under the *Trace Processing* column. *Slowdowns* are for a single simulation run, while *Effective Slowdowns* are computed by dividing by the number of configurations (given in parenthesis) simulated during that run. In each row, slowdowns were taken (or computed) directly from the referenced paper. For entries that have an asterisk by the reference, slowdowns do not come from the paper, but were determined by running the tool on a DECstation 5000/240.

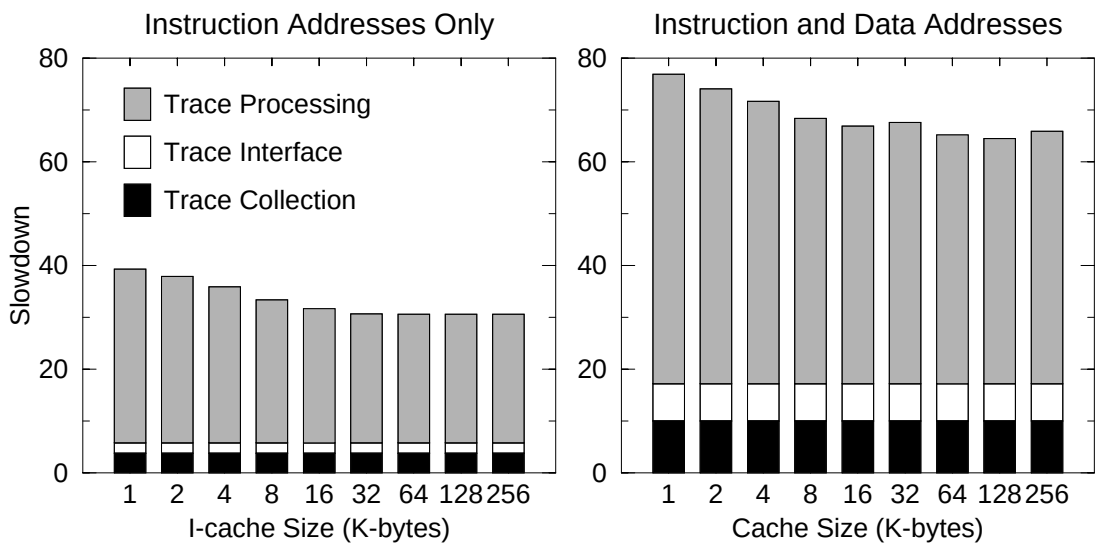


Figure 2.3 The Components of Trace-driven Slowdowns

These two plots give the components of trace-driven slowdowns for the fastest, single-configuration trace-driven simulator system found in this survey. The trace processor is Cache2000 and the trace collector is Pixie. These components were interfaced by an Ultrix pipe. The left plot shows slowdowns for I-cache simulations, while the right plot shows the slowdowns when simulating both I- and D-caches concurrently.

composed of a wide range of different trace collection and trace processing methods. As such, these systems are fairly representative of the sort of trace-driven simulation environments that can be constructed with state-of-the-art methods. We could find no examples of trace-driven simulators that exhibit overall slowdowns less than about 50 when they consider both instruction and data references. The fastest combination was Cache2000 driven by Pixie traces generated on-the-fly and passed through a pipe. Note that the overheads of the two multi-configuration simulators (Tycho and Cheetah) cause their overall slowdowns relative to single-configuration simulation (Cache2000) to be much higher than the values reported in Section 2.2.3. For Cheetah, the overheads are at least 300%, and for Tycho they are an order of magnitude higher.

To better understand the sources of trace-driven slowdown, we measured the speed of Cache2000 driven by Pixie traces over a range of instruction and data cache sizes. The results, shown in Figure 2.3, illustrate that most of the slowdowns are due to trace processing. However,

even if the trace processing time were reduced to zero, trace collection and trace communication each still slow the workload down by a factor of 5 to 15.

As Table 2.8 shows, the generation, transfer and processing of trace data for memory-system simulation is extremely challenging. With increasing processor speeds, trace-driven slowdowns are likely to increase in the future. This is a simple consequence of our definition of slowdown, which divides the overall rate at which a trace-driven simulator can collect and process addresses by the address-generation rate of the host machine (recall Eqn 2.1 - Eqn 2.3). Assuming that workloads fit reasonably well into an on-chip CPU cache, the denominator in this ratio increases in direct proportion to increasing CPU clock rates, which have improved by roughly 40% each year during the past decade [Upton94]. The numerator, however, is much more affected by bandwidth improvements all along the memory hierarchy for two reasons. First, many popular trace-collection methods, such as those based on code instrumentation or ISA emulation, cause traced workloads to bloat in size. This means that instrumented or emulated programs do not fit well into CPU caches and are therefore bound more by main-memory bandwidths. Second, trace interfaces that use buffering often rely on slower parts of the memory hierarchy. For example, files rely on disks, and pipes rely on cross-domain transfers that usually result in main-memory data copies.

In a best-case scenario, traditional trace-driven simulation is unlikely ever to exhibit slowdowns lower than about 20 to 35. This conclusion can be reached by observing three basic facts uncovered earlier in this analysis:

(1) The fastest trace collectors are based on static code instrumentation and their slowdowns are no better than about 10 to 15.

(2) The fastest trace-driven systems perform trace collection and trace processing on-the-fly. However, this technique effectively precludes the use of trace reduction methods, and imposes overheads that add a slowdown of at least 5 to 10 simply to communicate trace data from the trace collector to the trace processor.

(3) Processing an address for the simplest of memory structures, a direct-mapped cache, requires a minimal slowdown of approximately 5 to 10. Anything more complex than direct-mapped cache simulations must necessarily take more time.

The sum of these slowdowns for trace collection and trace processing are in the range of 20 to 35. Our analysis of the fastest single-configuration trace-driven simulator that we could find (Pixie

+ Cache2000) exhibits slowdowns in the range of 60 to 80, and the fastest multi-configuration simulator (Pixie + Cheetah) is at least 3 times slower, supporting the claim that trace-driven simulation is, in practice, one to two orders of magnitude slower than actual hardware.

2.3 Summary

The purpose of this chapter has been threefold: (1) to define and clarify the common issues and problems inherent in any form of memory-system simulation, (2) to evaluate and analyze the state-of-the-art in trace-driven memory simulation, and (3) to learn from trace-driven methods so that they may be applied to trap-driven simulation.

Our definitions of the issues and problems associated with memory-system simulation were given in the context of an *ideal trace-driven simulation environment*. The most important aspect of simulation is that it produce accurate and meaningful results. To assess this characteristic in less-than-ideal systems, we defined the concept of *trace quality* which relates directly to the accuracy of trace-driven simulations. Our definition of trace quality specifies what is meant by *trace completeness*, *trace detail* and *trace distortion*. We also noted the importance of simulation *speed* because it constrains the degree to which a design space can be explored. We defined a simple metric for speed, *slowdown*, which enables meaningful comparisons of simulation tools that have been implemented on host machines of differing speeds. In addition to the main issues of accuracy and speed, we have defined a handful of other considerations to use when evaluating a memory-system simulator. These include *expense*, *portability*, *ease-of-use* and, finally, the *flexibility* and *range* of simulations that can be performed.

The foregoing survey leads to several conclusions contrary to the conventional wisdom. In particular, instruction-set emulation is faster than commonly believed, probe-based trace collection is slower than commonly believed, and multi-configuration simulations include more overhead than typically reported. Most importantly, no single method is best when all points of comparison are taken into consideration; there is much variability among the methods with respect to speed, accuracy, flexibility, expense, portability and ease-of-use. Our analysis enables us to reach two basic conclusions about the current state of trace-driven simulation accuracy and speed: (1) High-quality traces are still quite difficult to obtain and even the very best traces may be incomplete, lack detail, or contain distortions that can affect simulation accuracy. (2) Because of

inherent bottlenecks, traditional trace-driven simulation will always be one to two orders of magnitude slower than actual hardware.

Our final objective for this study was to identify trace-driven simulation techniques that could be adapted to work with trap-driven simulation. As we will see in the next chapter, techniques such as cache filtering, set sampling, time sampling and multi-configuration simulation can all be implemented, often with relative ease, in a trap-driven simulator. This, combined with an ability to capture a complete and detailed picture of memory-referencing behavior and to break past inherent bottlenecks in trace-driven simulation speeds, makes trap-driven simulation a very attractive alternative for memory-system simulation. However, trap-driven simulation has certain disadvantages of its own. We begin our study of both the pros and cons of trap-driven simulation in the next chapter.

Chapter 3
Trap-driven Simulation

Strict adherence to the trace-driven simulation paradigm is likely to limit further substantial improvements in memory-simulation speeds. The primary bottleneck in trace-driven simulation comes from collecting and processing *each* memory reference made by a workload, whether or not it changes the state of a simulated memory structure. Several researchers, noting this bottleneck to trace-driven simulation, have developed innovative software-based methods for eliminating or reducing the cost of processing addresses that do not change simulation state. We begin this chapter with a brief survey of three such tools, which each use very efficient code-instrumentation techniques to improve simulation speeds by a factor of 2 to 3 relative to fast trace-driven simulators.

In the second part of this chapter, we argue that software-only simulation methods, like those described above, simply cannot provide further speed improvements. Software-only simulation methods have, after over 30 years of development, reached a speed barrier, and further advances will only be possible if simulation-host hardware is more directly involved in the task of sorting memory references that change simulated-memory state. This is where our investigation of trap-driven simulation begins. We examine early trap-driven simulators that demonstrated that simulated TLB and cache hits could be processed at the full speed of the hardware hosting a simulation.

These first-generation trap-driven simulators, while successful in demonstrating the feasibility of a new simulation method, left unanswered a number of questions regarding its overall flexibility, portability, speed, and accuracy. In the final section of this chapter, we introduce the design of a new trap-driven simulator, Tapeworm II. This design, and a prototype implementation of it, will be used in subsequent chapters to examine and answer open questions about trap-driven simulation in general.

References	Name	Overall Slowdown	Cache Configurations	Completeness
[Martonosi92;93]	MemSpy	10 - 20	D-cache (128-KB)	Single Task, No OS
[Srivastava94]	ATOM	12	D-cache (—)	Single Task, No OS*
[Lebeck94; 95]	Fast-Cache	3 - 7	D-cache (16-KB to 1-MB)	Single Task, No OS

Table 3.1 Beyond Traces: Some Recent Fast Simulators

Overall Slowdowns are highly dependent on workload and the configuration (size, associativity) of the cache being simulated, so we report slowdowns as a range of values. Wherever possible, we describe the simulation parameters in terms of *Cache Configuration* and *Completeness*.

* Srivastava et al. report that efforts to annotate the OSF/1 kernel with ATOM are in progress.

3.1 Beyond Trace-driven Simulation

Table 3.1 summarizes three recent memory-system simulators that each try to eliminate or remove the cost of processing workload memory references that do not change simulation state. These techniques, *hit bypassing*, *customized instrumentation*, and *active memory*, are each based on optimizations of standard code annotation techniques.

3.1.1 Hit Bypassing

MemSpy is a memory simulation and analysis tool built on top of the TangoLite trace-collection tool [Martonosi92]. Original implementations of MemSpy, which instrumented assembly code to call a simulation routine after each heap or static data reference,[1] exhibited typical trace-driven slowdowns in the range of 20 to 60 when performing simulations of a 128-KB, direct-mapped data cache. Each call to the MemSpy simulator incurred overheads for saving and restoring registers, simulating the cache, and updating statistics. Altogether, it cost about 230 to 275 cycles to service a simulated cache hit and 320 to 510 cycles for a cache miss. Martonosi observed that cache hits are the more frequent simulation event and can be optimized by modifying the code that instruments each memory reference with code to immediately detect

1. MemSpy does not instrument instruction references or data references to the stack segment. This means that, depending on the workload, MemSpy instruments between about 8.3% to 21% of all instructions.

simulated cache hits. When this case occurs, the MemSpy simulator code is *bypassed* and execution continues to the next instruction. This *hit-bypassing* code requires about 25 instructions, compared with the 230 to 275 cycles for a full call into the MemSpy simulator. The resulting MemSpy slowdowns, after the hit-bypassing optimization, were measured to be in the range of 10 to 20.

Because hit-bypassing is implemented in software, it limits the effectiveness of techniques such as time sampling [Laha88] and set sampling [Puzak85]. Martonosi investigates time sampling in a later paper by adding another check to the instrumented code to enable and disable monitoring at regular intervals [Martonosi93]. When enabled, instrumentation overheads are similar to those cited above, but when disabled, an instrumented reference executes only 6 extra instructions. When trapping is enabled for 10% of the entire execution time, MemSpy slowdowns drop to about 4 to 10, a factor of two improvement over simulations without sampling. This is a rather disappointing result given that 10% sampling should ideally result in a factor of 10 speedup. The bottleneck is the cost of instrumentation; Even when trapping is turned off, each instrumented memory reference still results in the execution of 6 extra instructions.

3.1.2 Customized Instrumentation

A common problem with many trace-collection tools is that they produce traces with an inflexible level of detail, which may provide too much or too little information for a given type of simulation. For example, a trace complete with both instruction and data references provides more trace information than is actually needed by an I-cache simulator which must filter away the data references. Many trace collectors are similarly rigid in the mechanism that they use to communicate addresses, typically forcing the trace through a file or pipe interface to another task containing the trace processor. Although these policies enable generic trace-driven simulators to be built, they miss certain opportunities to speed simulations.

The ATOM tracing system is designed to solve some of these problems by supporting *customized instrumentation* [Srivastava94]. With ATOM, a simulator writer defines an instrumentation specification that controls the way that a workload is annotated. To do I-cache simulation, for example, a simulator writer can specify that only instruction references be annotated, and that a specific I-cache analysis routine be called at these points. ATOM then uses an object-module editing tool, called OM, to embed calls to the I-cache analyzer routines, while using

a number of optimizations to save and restore a minimum of registers. Srivastava et al., report that cache[2] simulation can be performed on ATOM-annotated SPEC92 benchmarks with a slowdown of about 12 [Srivastava94].

3.1.3 Active Memory

Fast-cache is another tool that implements a more flexible and optimized interface between an annotated workload and simulator routines [Lebeck94; Lebeck95]. Fast-cache is based on an abstraction called *active memory.* Each block of active memory has an associated *handler* routine that is called whenever memory locations in the block are referenced. During a memory-system simulation, these handlers can be changed dynamically to implement certain optimizations. For example, a memory block that is known to be held in a simulated cache can have its handler set to point to a NULL routine. An access to this memory location results in an event that doesn't change simulated-memory state (i.e., a simulated cache hit) and because its handler is a NULL routine, the simulator is not invoked. *Fast-cache* implements active memory blocks in software by adding 9 instructions to instrument each original workload instruction that makes a memory reference (only 5 instructions execute for references that invoke the NULL handler). Fast-cache currently works on single-task, data-only simulations; No instructions or operating system references can be instrumented. Fast-cache achieves overall slowdowns in the range of 3.1 to 6.7 for the simulation of direct-mapped data caches ranging in size from 16KB to 1MB [Lebeck94].

3.1.4 Summary of New Memory Simulation Methods

The success of ATOM, MemSpy, and Fast-cache suggest that memory-simulation speeds can be improved over traditional trace-driven methods by reducing the cost of processing workload events that do not affect simulated memory state. Because all three systems are based on software-only methods, they share a number of important advantages. Namely, they are flexible, low in cost, and relatively portable because they do not rely on special hardware support. ATOM, for example, can also be used for a variety of other customized program-analysis tools, including file I/O analysis, program profiling, dynamic-memory allocation analysis, and compiler auditing [Srivastava94].

2. The parameters of the simulated cache were not reported.

Because they are based on the same basic techniques as trace collectors that use code instrumentation (see Section 2.2.1), these tools also suffer from some of the same disadvantages, such as causing program text to bloat in size by factors as high as 5 to 10 due to added instructions. Code expansion may not be a concern for applications with small text segments, but instrumenting larger, multi-task workloads along with the kernel, can cause substantial bloat. For this and other reasons, these tool are difficult to extend beyond single-task workloads. To date, all three tools exclude kernel references and dynamically-linked or dynamically-compiled code, although work is reportedly in progress to instrument the OSF/1 kernel using ATOM. Finally, as reported by Martonosi, code-instrumentation techniques have difficulty exploiting the full benefits of set and time sampling because of the base overhead they add, even when in a non-sampling mode.

3.1.5 Performance Bounds on Software-only Methods

The highly-tuned simulation systems described above improve over traditional trace-driven simulation speeds by a factor of 2 to 3, reducing slowdowns to within an order of magnitude of actual hardware. Unfortunately, there are two reasons to believe that this is the end of the road for improvements in the speed of software-only memory simulators. First, the slowdowns of these tools are bounded by the number of instructions that they add to a workload. The fastest simulator in the group, Fast-cache, must execute at least 5 extra instructions for each instrumented memory reference (and many more if the reference misses the simulated cache). Second, code that is bloated by annotation increases host I-cache misses. An instrumented version of the `tomcatv` SPEC92 workload, for example, exhibits a 20-fold increase in the host I-cache miss ratio [Lebeck95]. In other words, slowdowns are bounded not only by the number of instructions added to a workload for annotation, but also by the host I-cache, which will always be less effective when executing a bloated, annotated workload than the original, un-annotated workload.

These bounds to maximum simulation speed are evident in published results for Fast-cache and MemSpy. Fast-cache slowdowns for the simulation of a perfect D-cache that never misses are 2.5 to 4.5 [Lebeck94] and for MemSpy they are about 10 [Martonosi94]. It is important to note that both of these simulators instrument data references only, and that the above slowdowns are for single-task workloads with relatively small text segments. Complete instrumentation of both data and instruction references of larger workloads is likely to bring the best-case slowdowns into the range of 10 to 20.

3.2 Enter Trap-driven Simulation

Unlike the software-only methods described in the previous section, a trap-driven simulator relies on the underlying host hardware to detect workload memory references that change simulated-memory state. Because a trap-driven simulator adds no instructions to a workload, it avoids the bounds that limit the fastest memory simulators based on code instrumentation. In principle, trap-driven simulations can achieve near-zero slowdowns. We begin our study of trap-driven simulation with a look at first-generation trap-driven simulators (see Table 3.1).

3.2.1 Trap-driven TLB Simulation

The first-generation *Tapeworm* is an early example of trap-driven TLB simulation [Nagle93; Uhlig94b]. Tapeworm relied on the fact that all TLB misses in a MIPS-based DECstation are handled by software in the operating system kernel. The Tapeworm code was compiled into the operating system kernel and the usual TLB miss handlers were modified to call the Tapeworm code via procedural “hooks” after every miss. This mechanism passed the relevant information about all user and kernel TLB misses directly to the Tapeworm simulator. Tapeworm used this information to maintain its own data structures and to simulate other possible TLB configurations.

With Tapeworm, a simulated TLB could be either larger or smaller than the host TLB because Tapeworm ensured that the host TLB only held entries available in the simulated TLB. For example, to simulate a TLB with 128 slots using only 64 host TLB slots, Tapeworm maintained an array of 128 virtual-to-physical address mappings and checked each memory reference that missed the host TLB to determine if it would have also missed the larger, simulated TLB. Tapeworm thus maintained a strict inclusion property between the host and simulated TLBs. Tapeworm controlled the actual TLB management policies by supplying placement and replacement functions that were called by the operating system miss handlers. It simulated TLBs with fewer entries than the host TLB by providing a placement function that never utilized certain slots in the host TLB. Tapeworm used this same technique to restrict the associativity of the host TLB.[3] By combining these policy functions with adherence to the inclusion property, Tapeworm could simulate the performance of a wide range of different-sized TLBs with different degrees of associativity and a

3. The actual (host) R2000 TLB is fully-associative, but varying degrees of associativity can be emulated by using certain bits of a mapping’s virtual page number to restrict the slot (or set of slots) into which the mapping may be placed.

References	Name	Trap Method	Cycles per Trap	Overall Slowdown	Type of Simulation	Completeness
[Nagle93; Uhlig94b]	Tapeworm	TLB Miss Trap	100 - 650	0.5 - 4.5	TLB	Multi-task, With OS
[Reinhardt93]	WWT	Memory Parity (ECC)	2,500[1]	1.4 - 46[1]	MP D-cache	Single Task, No OS
[Lee94]	Tapeworm486	Invalid Page Trap	3,600 - 4,000	0 - 14	TLB	Multi-task, With OS
[Talluri94]	Foxtrot	TLB Miss Trap	1,500 - 4,000	—	TLB	Single Task, No OS
[Uhlig94]	Tapeworm II	Memory Parity (ECC) Invalid Page Trap	300 325 - 360 700	0 - 10	I-cache Instr Prefetch TLB	Multi-task, With OS

Table 3.2 Trap-driven Simulators

The code in a trap-driven memory-system simulator is only invoked when the simulated memory state changes (e.g., on cache misses). Traps into the simulator are implemented using a variety of techniques (see *Trap Method*). *Cycles per Trap* are the approximate number of host machine cycles required to handle a trap. *Overall Slowdown* are highly dependent on the configuration (size, associativity) of the cache or TLB being simulated, so we give slowdowns in ranges. *Type of Simulation* and *Completeness* summarize the range of simulation supported (MP = Multi-processor).

[1]Miss costs and slowdowns for WWT were taken from [Lebeck94].

variety of placement and replacement policies. Because all user and kernel misses were intercepted, Tapeworm was able to fully consider multi-task and OS effects for these different TLB configurations.

The principal advantage of driving Tapeworm with hardware-generated kernel traps is that non-trapping memory references proceeded at the full speed of the underlying host hardware; A simulated TLB hit resulted in no system slowdown. On the other hand, a simulated TLB miss incurred the full overhead of a kernel trap and the simulator code, which varied from 100 to 650 host cycles. Fortunately, TLB hits are far more frequent than TLB misses, outnumbering them by more than 300 to 1 in the worst case [Nagle93; Uhlig94b]. The result is that Tapeworm TLB simulation slowdowns ranged from about 0.5 to 4.5.

Recently, trap-driven TLB simulation has been implemented on other architectures with similar success. Lee has implemented a trap-driven TLB simulator on a 486-based PC running Mach 3.0 [Lee94]. Because the i486 processor implements hardware-managed TLBs, this simulator uses a different mechanism for causing TLB miss traps, one that is based on page-valid bits. By manipulating the valid bit in a page-table entry, Lee's simulator can cause TLB misses to result in kernel traps in the same way that they do in a machine with software-managed TLBs. Talluri et al. uses similar techniques in a trap-driven TLB simulator that runs on SPARC-based workstations under the *Foxtrot* operating system to study architectural support for superpages [Talluri94]. Talluri and Lee both report that the overall slowdowns for their simulators are comparable to those of Tapeworm.

3.2.2 Trap-driven Cache Simulation

A limitation with the trap-driven simulators described above is that they are not easily extended to cache simulation. This is because the mechanisms that they use to cause kernel traps operate at the granularity of a memory page. The first trap-driven simulator that overcame this limitation is the *Wisconsin Wind Tunnel* (*WWT*), which caused kernel traps by modifying the error-correcting code (ECC) check bits in a SPARC-based CM-5 [Reinhardt93]. Because each memory location has ECC bits, this method enabled traps to be set and cleared with a much finer granularity, enabling cache simulation.

As with the trap-driven TLB simulators noted above, a simulated cache hit in WWT ran at the full speed of the host machine, and for caches with low miss ratios, overall slowdowns were measured to be as low as 1.4. However, in a recent comparison with Fast-cache, Lebeck et al. reports that WWT exhibits slowdowns of greater than 30 or 40 for caches smaller than 32KB [Lebeck94]. These slowdowns are much higher than those reported for TLB simulation, both because cache misses occur much more frequently than TLB misses, and because a WWT trap requires about 2,500 cycles to service.

3.2.3 Open Questions about Trap-driven Simulation

The earliest attempts at trap-driven simulation in Tapeworm and WWT showed that near-zero slowdowns for memory simulations are possible. Unfortunately, these simulators raised more questions than they answered. These first-generation trap-driven simulators exhibited wide ranges of simulation slowdowns (see Table 3.1), making it difficult to determine when trap-driven simulation is faster than trace-driven simulation. The range of simulations that could be conducted was limited (TLB-only in Tapeworm or D-cache, single-task-only in WWT) and methods for setting traps were ad-hoc, drawing into question the flexibility and portability of the method in general. Finally, the potential sources of trap-driven simulation error were generally unstudied.

3.3 Tapeworm II

Tapeworm II is a trap-driven simulator design that we will use in subsequent chapters for answering open questions about trap-driven simulation. A prototype implementation of this design runs in the Mach 3.0 operating system kernel on a MIPS R3000-based DECstation 5000/200, and can simulate a range of TLBs, instruction caches, and instruction prefetch units. In this section, we discuss our design goals for Tapeworm II[4] and then present an overview of its organization in terms of its system-independent, OS-dependent, and hardware-dependent components.

4. Note: Tapeworm II is different from the original Tapeworm TLB simulator, but for the purposes of brevity, we often simply refer to it as Tapeworm.

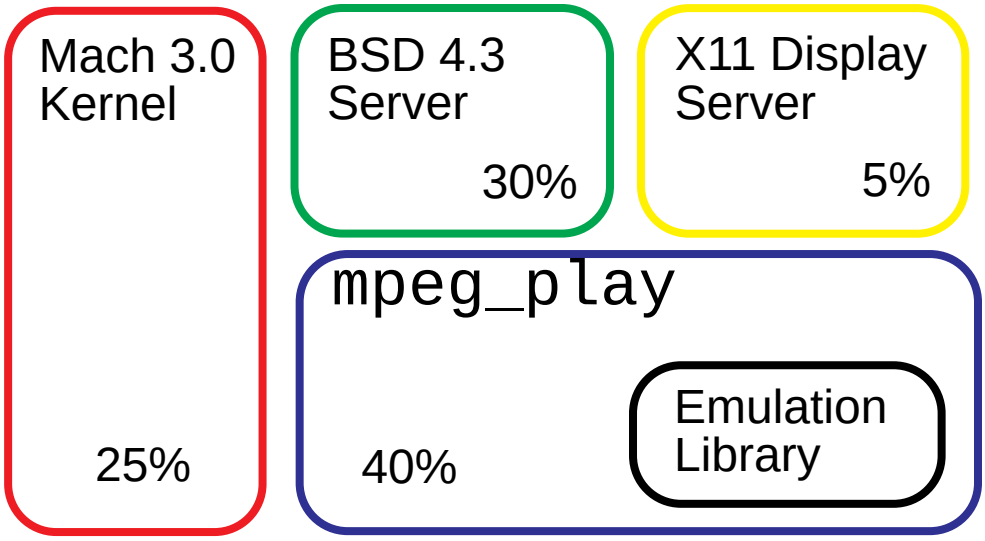


Figure 3.1 The MPEG Decoder Workload

3.3.1 Design Goals

Tapeworm was developed to enable studies of the interaction between computer architectures and operating systems. As such, our main design goal was to be able to account for the memory referencing behavior of all components of a workload, which might be spread across multiple user-level tasks and include the operating system kernel. Figure 3.1 illustrates the type of workload that Tapeworm was designed to monitor. The figure shows `mpeg_play`, an MPEG video decoder running under the Mach 3.0 operating system. This workload is difficult to monitor for a number reasons. First, a substantial fraction of its time (25%) is spent in the privileged kernel mode of execution. Second, its user-level components are spread across three different tasks: the Mach BSD server, the X11 display server and the `mpeg_play` process itself. Third, the workload includes a segment of code, called the BSD emulation library, which is dynamically-linked into the `mpeg_play` address space.

In addition to being able to monitor all workload components, we also wanted to isolate interactions between different workload components by specifying that a given component be cached or not cached during a given simulation run. By running multiple trials with different combinations of workload components cached, this capability would enable us to measure cache interference effects.

Code	Lines of Code	%of Total Code
Machine-independent Code	5652	82%
OS-dependent Code	889	13%
Hardware-dependent Code	343	5%

Table 3.3 Tapeworm Code Distribution

The majority of Tapeworm code is machine-independent, consisting of the user-level X interface `xtw` and a `twControl` task that runs on each simulation workstation. The remaining OS- and hardware-dependent code is detailed in Table 3.5 and Table 3.5.

Because Tapeworm was also intended for an exploration of the pros and cons of trap-driven simulation, it was important that its design be extensible to explore issues of flexibility, and that it be structured for portability to other machines. Finally, it was important that its trap handlers be as fast as possible so that it could exceed previously-established limits to trap-driven simulation speed.

3.3.2 Tapeworm Organization

A Tapeworm simulation system is composed of a *controlling workstation* and one or more *simulation hosts* (see Figure 3.2). The simulation host workstations each have a Tapeworm embedded in their kernels, which respond to commands from a user-level `twControl` task. The controlling workstation connects to each of the `twControl` tasks to issue simulation commands, and runs a user interface (`xtw`) and a database server to log simulation results. This division between simulation control and simulation hosts minimizes perturbation to workloads under study and enables multiple simulations to be conducted concurrently.

The Tapeworm code is divided into machine-independent, OS-dependent, and hardware-dependent sections (see Table 3.3). To explain how these different components interact, we will walk through an example Tapeworm simulation scenario.

System-independent Code

A Tapeworm simulation begins with a user-defined specification of a workload and host operating system, along with the parameters (size, associativity, replacement policies, etc.) of the

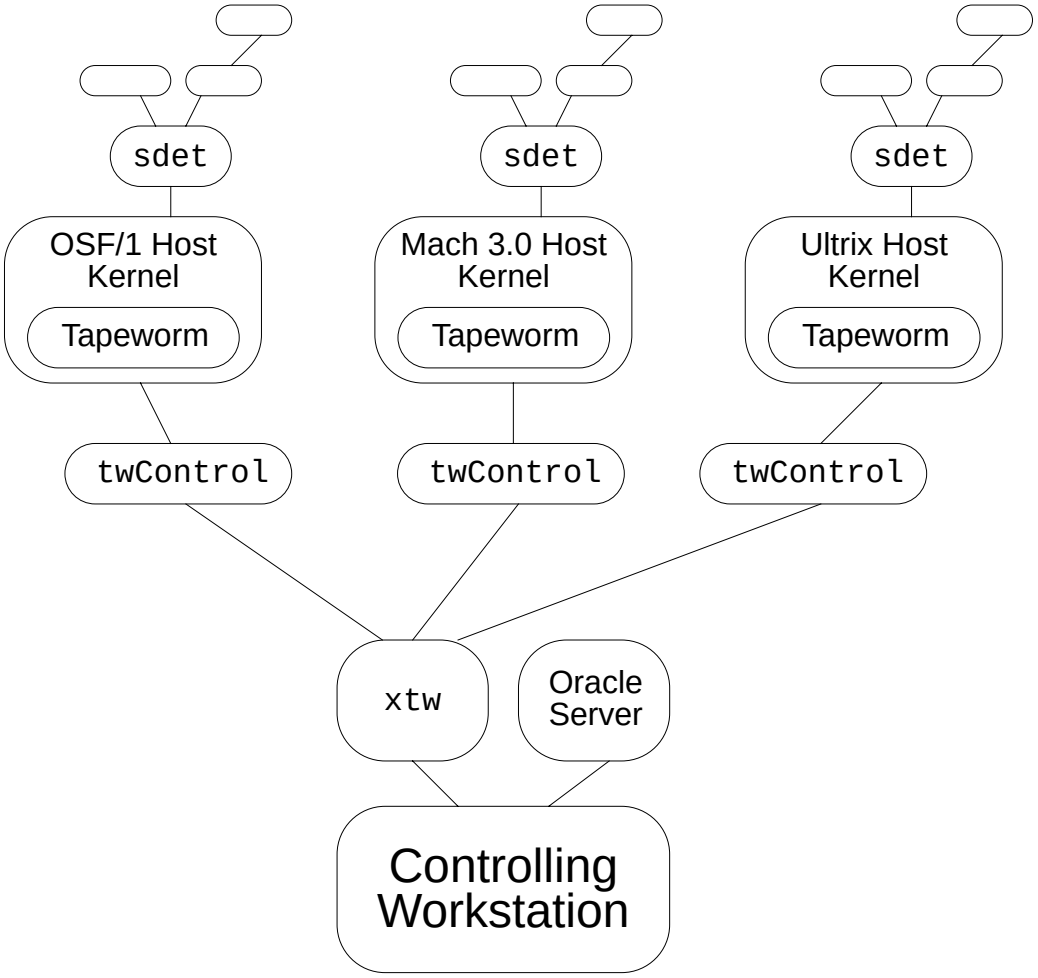


Figure 3.2 The Tapeworm Simulation System

caches or TLBs to be simulated. The user can also specify which workload components (user, kernel, servers) should be included in the simulation. These simulation parameters are controlled through `xtw`, the X interface to Tapeworm (see Figure 3.3 for an example of one dialog box in this interface). After the simulation parameters have been set, `xtw` issues a series of remote-procedure calls to the user-level `twControl` task that runs on a simulation host. `twControl`, in turn, makes kernel calls to the OS-dependent code inside the kernel of the simulation host to initialize Tapeworm, and then starts a simulation by running the specified workload.

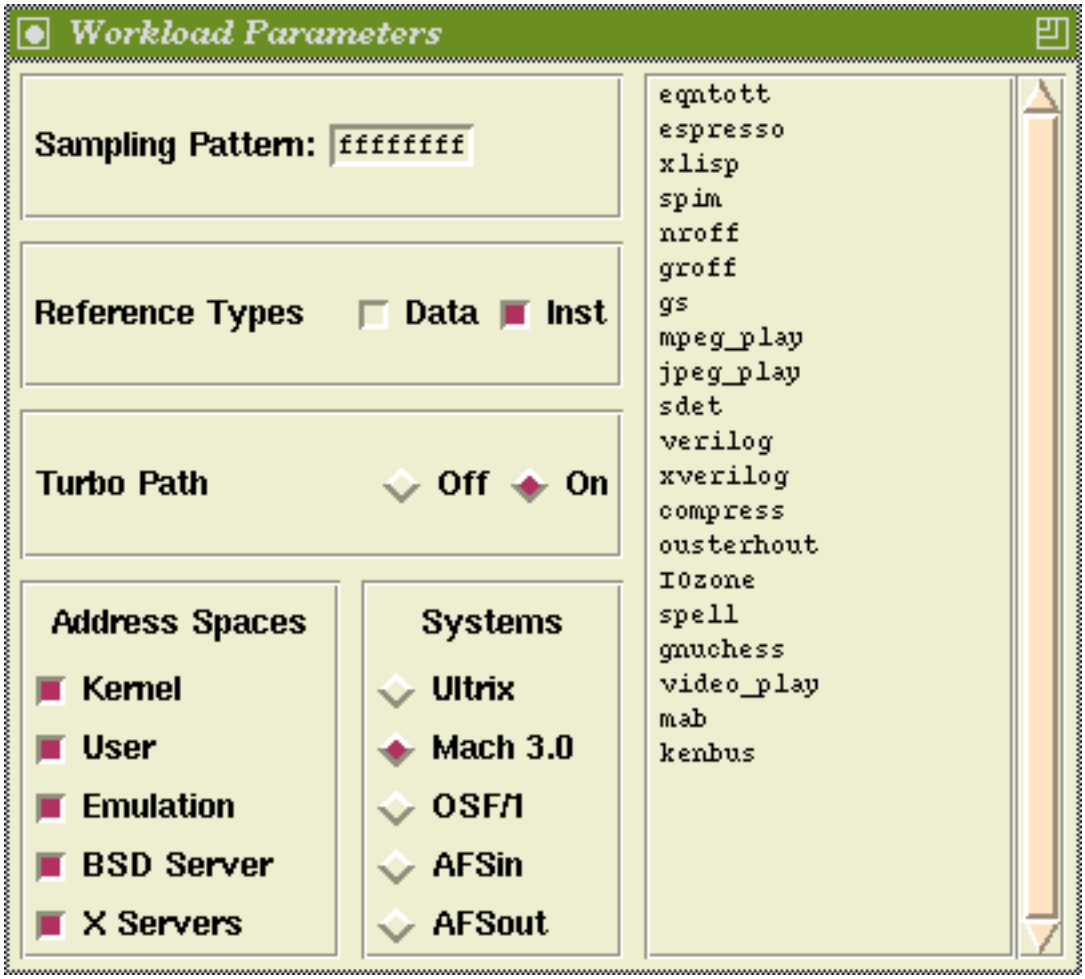


Figure 3.3 Basic Tapeworm Simulation Parameters

This figure depicts a dialog box from the X interface to Tapeworm. Before a simulation begins, a user selects a workload and operating system to run under. The user also specifies which workload components (e.g., kernel, user-task tree, BSD server, X server) are to be included in the simulation. A user can also specify a set-sampling pattern to speed simulations at the expense of some loss in accuracy.

OS-dependent Code

At the beginning of a Tapeworm simulation, the simulated cache[5] is empty, a condition that

5. For the remainder of this discussion, we will describe cache simulation, although the same basic actions apply to TLB simulation as well.

Routine	Description
`tw_add_page(tid, p, v)`	Add a page to the Tapeworm domain. The page is added by setting traps on all of its physical memory locations starting at the page address `p`. The task ID `(tid)` and the virtual-to-physical page mapping defined by `(p,v)` are recorded by Tapeworm to enable forward and reverse address translations.
`tw_remove_page(tid, p, v)`	Remove the page define by `(tid, p, v)` from the Tapeworm domain. The page is removed by flushing it from the simulated cache and by clearing all traps on its memory locations.
`tw_attributes(tid, simulate, inherit)`	Set Tapeworm attributes for the task identified by `tid`. A `tid` of zero signifies the kernel. A non-zero value of `simulate` registers a task with Tapeworm. A non-zero value of `inherit` indicates the initial value of the `simulate` attribute for children of the task.

Table 3.4 OS-dependent Tapeworm Routines

Tapeworm models by initially setting traps on all workload memory locations. To accomplish this, Tapeworm requires cooperation from the OS virtual-memory (VM) system. When a task faults on the first access to one of its pages, the VM system registers the page with Tapeworm using `tw_add_page()` (see Table 3.5), which sets traps on each memory location in the page. As the workload begins to use the new page, the first reference to each location causes a trap into the kernel, which is directed to Tapeworm. All such traps represent simulated cache misses, so the handler counts the miss and then clears the trap on the required memory location. Clearing the trap effectively caches the memory location in the simulated cache structure because subsequent references made by the workload to this location will not trap, allowing the workload to proceed uninterrupted. As the simulated cache begins to fill, incoming traps may collide with previously cached memory locations, requiring that they be displaced from the cache. The Tapeworm trap handler simulates this by setting new traps on displaced memory locations, in accordance with the cache parameters (e.g., size, associativity, indexing policy, replacement policy, etc.).

A parallel routine, `tw_remove_page()`, is used by the VM system to remove pages from the Tapeworm domain when they are unmapped due to task termination or paging to secondary storage. `tw_remove_page()` clears all traps on a page and flushes the contents of the page from the simulated cache. This mimics the same actions performed by the VM system on the host machine's real cache.

If the VM system maps more than one virtual page to a given physical page, it must still register the mapping with Tapeworm by using `tw_add_page()`. In this case, Tapeworm increments a reference count for that physical page, but does not set any new memory traps. This enables a new task to benefit from shared entries brought into the cache by another task, as would happen in a real system. Similarly, `tw_remove_page()` decrements the reference count, and flushes the page from the simulated cache only when the reference count reaches zero.

Tapeworm supports cache simulation for workloads consisting of multiple tasks. To control which tasks are included in a given simulation, each is assigned two Tapeworm attributes (`simulate` and `inherit`), which are set by calling `tw_attributes()`, and are stored in an extended version of the OS task data structure (see Figure 3.4). If `simulate` is zero (the default value), then the task runs without any intervention from Tapeworm. When non-zero, `simulate` causes all current and future pages used by the task to be added to the Tapeworm domain via a `tw_add_page()` call. A second attribute, `inherit`, defines the initial value of `simulate` for all children of the task. After a task fork, a child task inherits the Tapeworm attributes of its parent as follows:

```
child.simulate <-- parent.inherit
child.inherit <-- parent.inherit
```

Different settings of the (`simulate`, `inherit`) pair are useful for common simulation situations. For example, if the attribute pair (`simulate=0`, `inherit=1`) is set on a shell task, then any workload that is started from this shell, and all of the workload's children will be registered with Tapeworm. The shell task itself, however, is excluded from the simulation. This inheritance mechanism simplifies the simulation of workloads with complex task fork trees, such as `sdet`, `kenbus`, or a multi-stage optimizing compiler. Another common attribute pair, (`simulate=1`, `inherit=0`) is used when only the task itself, but not its children, are to be simulated. This combination is useful for registering kernel pages with Tapeworm.

Hardware-dependent Code

The hardware-dependent portion of Tapeworm provides the routines needed to control host-memory trapping mechanisms, as well as an auxiliary routine that aids in the computation of performance metrics. Table 3.5 defines the complete semantics for the primitives in this interface.

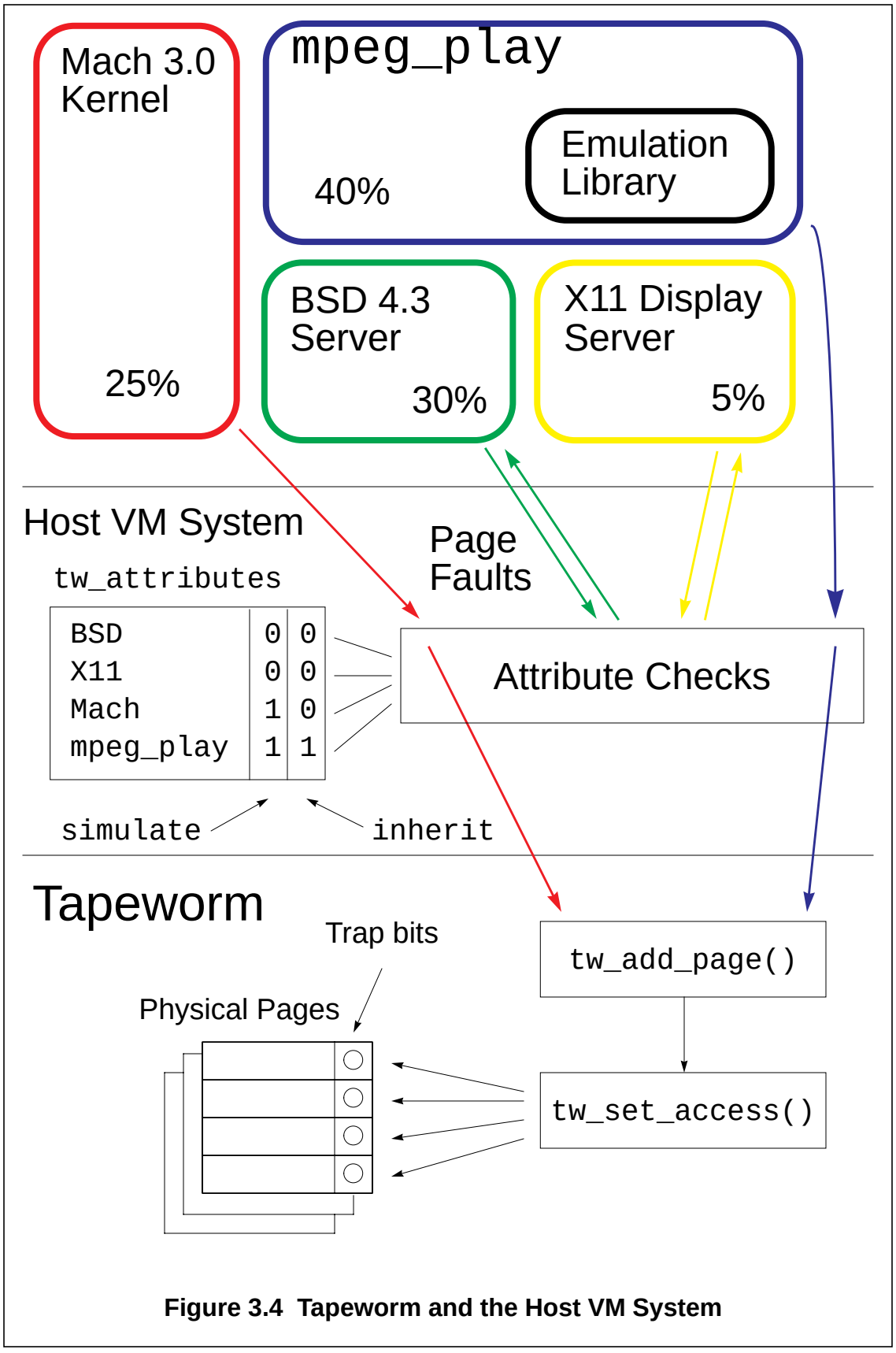


Figure 3.4 Tapeworm and the Host VM System

Routine	Description
`tw_set_access(pa, size, state)`	Set the access state of the memory region containing `pa` (a physical address). The operation is performed on memory boundaries that align to `size` bytes, starting at (`pa` div `size`) and extending for exactly `size` bytes. All subsequent references to memory locations in this range are checked against the trap `state`, which may be one of three values: `noAccess`, `readAccess` or `fullAccess`. Reads or writes to a location marked `noAccess` or writes to a location marked `readAccess` result in an OS kernel trap that passes control to Tapeworm with a call to `tw_trap()` (see below). Reads to a location marked `readAccess` or reads and writes to a location marked `fullAccess` can proceed at full hardware speeds.
`tw_get_access(pa)`	Return the access state for the memory location at physical address `pa`.
`tw_trap(pa, va, type)`	This routine, the entry point to the Tapeworm trap handler, is invoked by the host hardware whenever an access violation occurs. The host hardware should provide the physical address (`pa`) and virtual address (`va`) of the violating memory reference, and the type of memory reference (`type`), which can be a `dataLoad`, a `dataStore` or an `instrFetch`.
`tw_get_counts(type)`	Returns a count of occurrences of a given `type` of event (e.g., `dataLoad`, `dataStore`, `instrFetch`, `instrExec`). Only references to pages added to the Tapeworm domain should be counted (see `tw_add_page()` and `tw_remove_page()` in Table 3.5).

Table 3.5 Hardware-dependent Tapeworm Primitives

The first three routines, `tw_set_access()`, `tw_get_access()`, and `tw_trap()`, form the core of this interface, enabling Tapeworm to control the trapping mechanisms of the host hardware. To support the maximum flexibility in memory simulations, an implementation of these routines should support the full functionality defined in Table 3.5. In particular, the implementation of `tw_set_access()`, should support a wide a range of values in the `pa`, `size`, and `state` parameters. To enable multi-task and OS memory simulations, values of `pa` referring to any user or kernel memory location should be permitted. To enable both TLB and cache simulations, values of `size` ranging from as small as a cache line (4 or 8 words) to as large as a page (4 KB or 8 KB) should be supported. Finally, all three access (trap) states,[6] `noAccess`, `readAccess`, and `fullAccess`, should be supported to enable both I- and D-cache simulations.

The Tapeworm simulation algorithms only produce miss counts, but the final routine in the interface, `tw_get_counts()`, can be used to obtain event counts from which other performance metrics are computed. To support the computation of a range of performance metrics, this routine should report counts of memory load and store references, as well as instruction fetches and number of instructions executed.

These hardware-dependent primitives form an interface that is very similar to the memory-protection model supported by most microprocessor memory-management units. The important difference is that protection is provided at a finer granularity. Similar fine-grained access control interfaces have been proposed for systems that implement distributed shared memory [Appel91; Reinhardt94]. This interface differs slightly in its orientation to trap-driven simulation, including the addition of the event-counting routine `tw_get_counts()`.

3.4 Chapter Summary

We began this chapter with a survey of software-only memory simulators that have exhibited promising speedups by abandoning the trace-driven simulation paradigm. Because they rely on software methods only, these simulators offer important advantages, including relatively high portability and flexibility. But because they are all based on code annotation, they still suffer from performance bottlenecks, limiting them to run about an order of magnitude slower than actual hardware.

Early work in trap-driven simulation demonstrated for the first time that memory simulation slowdowns could approach zero, unbounded by any bottlenecks other than the inherent miss ratio of a simulated cache. These promising developments left unanswered, however, several questions regarding the flexibility, portability, speed and accuracy of trap-driven simulation.

The Tapeworm II design serves as a framework for exploring the bounds of trap-driven simulation in the remainder of this dissertation. In Chapter 4 we study trap-driven simulation flexibility by showing how a wide range of simulation algorithms can be implemented given the full support of the hardware-dependent Tapeworm primitives of Table 3.5. In Chapter 5 we discuss

6. *Access states* refer to those set by a trap-driven simulator using `tw_set_access()`. These levels of access are always a subset of the page-level access rights granted to a workload by the host VM system.

methods for porting these primitives to existing hardware, noting the difficulties we encountered in our prototype implementation of Tapeworm. In Chapter 6 we use this prototype implementation to show that for the smallest (1-KB) caches, trap-driven slowdowns start at the speed of the fastest trace-driven simulators, but steadily approach zero for larger caches. Finally, in Chapter 7 we use the Tapeworm prototype to study sources of trap-driven simulation error.

Chapter 4

Flexibility

The flexibility of a memory simulator can be assessed in a variety of ways. The range of memory configurations that can be modeled and the richness of the performance metrics reported by the simulator are certainly two important aspects of flexibility. If a simulator is capable of multiple-configuration simulation, then the size and dimensions of the design space covered in a single simulation pass is another measure of flexibility. Simulation flexibility is also enhanced through the ability to make speed-accuracy trade-offs. For example, it is often desirable to allow a 10% increase in simulation error in exchange for a factor of 10 increase in simulation speed; a flexible simulator enables such trade-offs.

Over the years, trace-driven memory simulators have demonstrated a high degree of flexibility in all of these respects. Practically any conceivable memory system can be simulated with trace-driven approaches and, as discussed in Chapter 2, many algorithms have been developed that enable a flexible range of multiple configurations to be simulated in a single trace pass. The survey in Chapter 2 also discussed techniques that filter or sample address traces in space or time to allow an architect to flexibly trade some degree of simulation accuracy for increased simulation speed.

Unfortunately, the flexibility of trap-driven simulation is not as well understood. The purpose of this chapter is to investigate issues of trap-driven flexibility and to answer the following questions:

- What is the range of memory configurations, policies, and performance metrics that can be simulated by a trap-driven simulator?
- Can multi-configuration simulation, and memory-reference filtering or sampling methods be adapted to trap-driven simulation?
- What are the inherent limitations of trap-driven simulation flexibility?

This chapter introduces the concept of *access constraints*, which help to answer the above questions by enabling us to reason about the feasibility of different forms of trap-driven memory simulation. Using the access-constraints model, along with a collection of diagrams, pseudo-code algorithms, and explanations, we show that a very broad range of memory configurations can be simulated with trap-driven techniques. We will also show that the trace-driven techniques of multi-configuration simulation, as well as set sampling, time sampling, and address trace filtering can be adapted to trap-driven simulation.

Many, but not all of the algorithms presented in this chapter have been implemented in the Tapeworm II simulator. At the end of this chapter, we describe our experiences with these implementations and comment on certain limitations encountered both with trap-driven simulation in general, and with Tapeworm II in particular.

4.1 Simulation Range

We define simulation range as the number and type of memory configurations that a simulator can model. Table 4.1 shows some of the cache structures, policies of operation, and performance metrics that define a simulated memory-system configuration. In this section, we will show how each of these aspects of memory-system design can be modelled with trap-driven simulation. We assume that the reader is familiar with the basic cache design options shown in the table, so we limit our discussions to the problems of simulating these design options, rather than describing their pros and cons in detail. Chapters from Hennessy's and Patterson's book on computer architecture [Hennessy90], and Smith's survey of cache memories [Smith82] give a more detailed explanation of these design options, along with advantages and disadvantages.

4.1.1 Basic Cache Structures

The purpose of a cache, of course, is to speed access to recently-used memory locations. A reference to memory that is held in the cache is processed faster than a reference to information that is only held in memory. Performance can be improved by caching many different forms of information, including instructions, data, and page-table entries. In the descriptions that follow, we discuss cache simulation in general terms, describing methods that work equally well for I-caches, D-caches, Unified caches and TLBs.

Basic Cache Structures	Cached Entity	Many forms of information benefit from being cached. We consider the caching of *instructions* in I-caches, *data* in D-caches and *page-table entries* in TLBs.
	Cache Size	The maximum amount of information (*in bytes*) that can be held in the cache memory.
	Line Size	Line size is the minimal unit of information that can be held (or mapped) by the cache structure. For I- and D-caches, this is the *line size*. For TLBs, this is the *page size*.
	Associativity	The degree to which several memory lines can co-reside in the same cache set. We consider *direct-mapped*, *set-associative* and *fully-associative* caches.
Basic Caching Policies	Replacement Policy	In an associative cache, a line refill often requires a cached line to be displaced from a set. We consider *Random*, *FIFO*, *LRU* and *NMRU* replacement policies.
	Write Policy	The method for updating main memory after a write. Options include *write-back* and *write-through* policies.
	Indexing Policy	The type of address used to index the cache. Options include indexing with a *physical* or a *virtual* address.
Forms of Cache Composition	Series	Caches are sometimes composed caches in series to form a *multi-level* hierarchy. The cache closest to the processor is usually called the first-level (L1) cache, which may be backed by a second-level (L2) and sometimes even a third-level (L3) cache.
	Parallel	Cache can also be composed in parallel. Common examples include *split* I- and D-caches, or *victim*, *assist*, and *hybrid* caches.
Cache Performance Metrics	Event Counts	The most basic performance metrics are simple counts of memory events that affect performance. Interesting events include read and write *misses*, cache-line *write-backs*, total number of read and write *references*, and number of *instructions executed*.
	Ratio Metrics	Basic event counts can be combined to form various ratio metrics, including *miss ratios*, *traffic ratios*, and *misses per instruction (MPI)*.
	Cycle and Time Metrics	Ultimately, the most important performance metric is the amount of time required to run a workload to completion. Given estimates for the cycle penalties of different memory-system events, such as *cycles per miss (CPM)*, it is possible to compute *cycles per instruction (CPI)*. This cycle metric, in turn, can be used to compute total workload run time, given a cycle time and the total number of instructions in a workload.

Table 4.1 Simulation Range: Cache Structures, Policies, and Metrics

This table shows the types of cache parameters and policies of operation that we study in this chapter. All of these aspects of caches (with the exception of a write-through write policy) can be modeled with a trap-driven simulator.

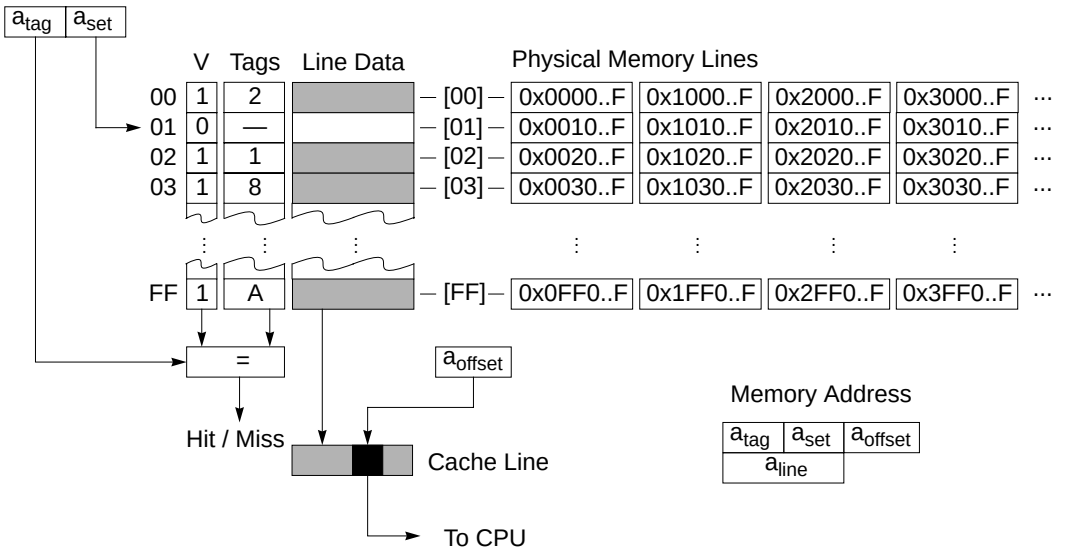


Figure 4.1 Direct-mapped Cache Hardware

For many of the examples in this chapter, we set the cache size to 4 KB and the line size to 16 bytes, thus dividing the cache into 256 sets (0x100, in hex). This implies that $||a_{offset}|| = 4$ bits and $||a_{set}|| = 8$ bits. These particular settings enable easy interpretation of addresses expressed in hex because the hex digits align to the boundaries of the three address parts. For example, address 0x2004 (tag 2, cache set 00, offset 4) would hit in the above cache, but address 0x3018 (tag 3, cache set 01, offset 8) would miss.

Simulating Direct-mapped Caches

Figure 4.1, shows the hardware for a simple *direct-mapped cache* that holds small contiguous segments (lines) of memory in its sets. The address (a) that is used to access this type of cache is divided into three parts. The middle part (a_{set}) is used to select a cache set, which consists of a valid bit (v), a tag, and a line of data. After the selected set has been read from the cache, the high-order bits of the address (a_{tag}) are compared against the cache tag. If there is a match and if the set is valid, then the low-order bits (a_{offset}) are used to select the appropriate instruction from the cache line. Notice that the combination of bits a_{tag} and a_{set} completely define the memory line. We therefore sometimes refer to the concatenation of these bits as a_{line}.

The number of bits in each part of the address (denoted by ||a||) is a function of the cache size and the line size:

$$||a_{offset}|| = \log_2(lineSize) \quad \text{(Eqn 4.1)}$$

$$||a_{set}|| = \log_2(cacheSets) \quad \text{(Eqn 4.2)}$$

$$||a_{tag}|| = ||a|| - ||a_{set}|| - ||a_{offset}|| \quad \text{(Eqn 4.3)}$$

$$||a_{line}|| = ||a_{tag}|| + ||a_{set}|| \quad \text{(Eqn 4.4)}$$

where $cacheSets = (cacheSize / lineSize)$ (Eqn 4.5)

The *memory equivalence class* of an address, denoted by [a], is:

$$[a] = \{\text{all addresses, b, such that } (a_{set} = b_{set})\} \quad \text{(Eqn 4.6)}$$

The cache in Figure 4.1 has 256 (FF in hex) memory equivalence classes, the elements of which are shown as a row of memory locations to the right of each cache set. The concept of a memory equivalence class is important because it precisely specifies the subset of memory locations that a given cache can hold. A direct-mapped cache, for example, is constrained to hold at most one line from each memory equivalence class at a time.

With these definitions in place, we can now explain how a trap-driven simulator models a direct-mapped cache. Figure 4.2, shows that a direct-mapped cache can be represented with a simple data structure (`cache[]`) that holds the starting addresses of cached memory lines, and a variable (`misses`) that counts the number of references that miss the simulated cache during the run of a workload.

The main task of a trap-driven simulator is to continually update these data structures so that they mirror the state of an actual cache running the same workload. The simulator accomplishes this by constraining access to the memory of the machine hosting the simulation in a way that causes a trap whenever the workload makes a memory reference that would result in a change of simulated cache state. In the simplest case, references that hit a cache do not change its state, but references that miss a cache do change its state because they are followed by a line refill that overwrites one of the cache sets.

Figure 4.2 shows how a trap-driven simulator can detect changes in cache state. The figure represents memory as a collection of pages, each divided into 16-byte regions on which access bits can be set. In this example, the possible access levels are *no access* and *full access*.[1] Notice that the current setting of full access on the 16-byte regions starting at 0x2000, 0x1020, 0x8030 and

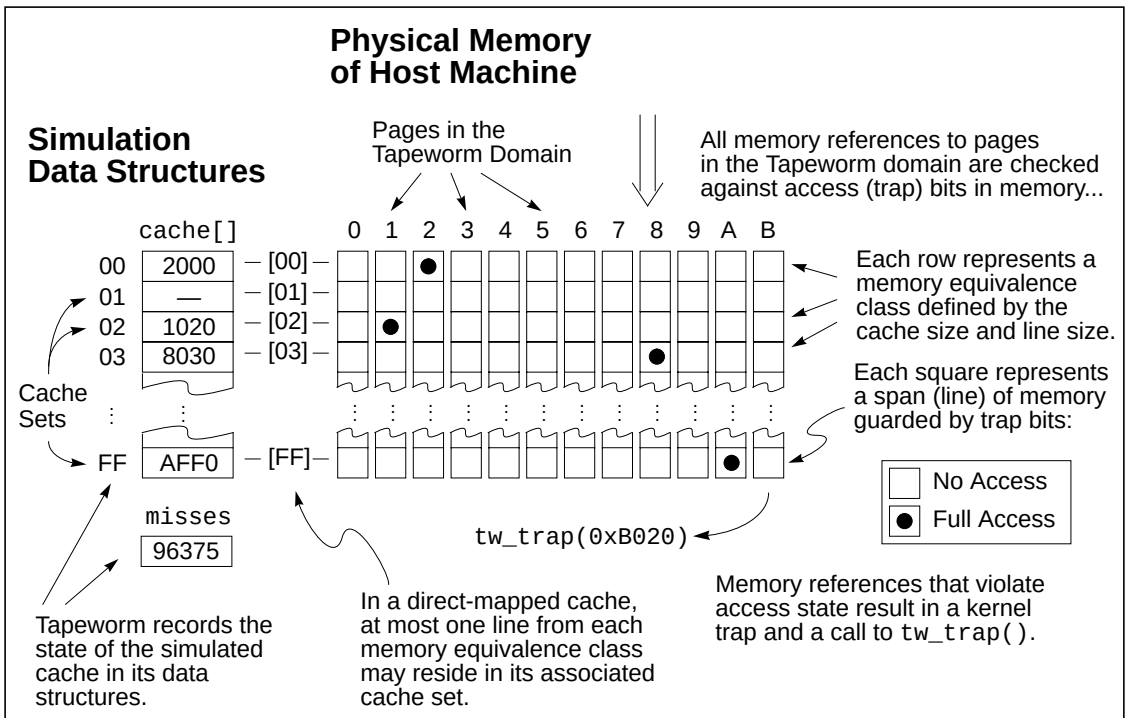


Figure 4.2 Data Structures and Host Memory for Direct-mapped Cache Simulation

This figure shows the data structures and host physical memory required for trap-driven simulation of the direct-mapped cache shown in Figure 4.1.

In a direct-mapped cache, at most one line from each main-memory equivalence class may be held in the cache at any point in time. Tapeworm emulates this constraint by permitting access to at most one line in each main-memory equivalence class.

0xAFF0, correspond exactly to the type of accesses that would result in hits for the cache shown in Figure 4.1. References that would miss the simulated cache, however, are marked as not accessible in the memory of the host machine and would cause a trap into the simulator if referenced. Notice that the simulator permits access to at most one line in each memory equivalence class, in keeping with the constrained way that a direct-mapped cache can hold memory lines.

Figure 4.2 shows the state of the simulator data structures and the host memory at one particular point in time during the run of the workload. We can see by inspection that the particular

1. Full access means the maximum access given to the page of the memory location by the host operating system. For text pages, full access is typically read-only, while for data pages it may be read-write access.

pattern of traps that have been set for this particular cache structure at this particular point in time will have the desired effect: the next reference to a memory location that is not in contained in the simulated cache will cause a trap. But what happens after the trap? That is, what actions must be taken by the trap handler to ensure that future references that change the state of the simulated cache will also cause traps? We need a more precise specification of the pattern of access rights on memory that are permitted throughout an entire workload run for a given cache configuration. To this end, we introduce the concept of *access constraints*.

We model the host physical memory system as a set of elements, P, that consists of the byte addresses of all memory locations that have been added to the simulator's domain (recall `tw_add_page()` in Chapter 3). The size of this set, denoted by |P|, is the total number of physical memory locations that are subject to the simulator's access controls. The subset $C \subset P$ represents the memory locations that may be accessed without causing a trap. We can now express the access constraints for the simulation of a direct-mapped cache as follows:

The *cache-size constraint*:

$$|C| \leq \text{cacheSize} \qquad \text{(Eqn 4.7)}$$

The *line-size constraint*:

$$\forall (a, b \in P)\ \{\ (a_{line} = b_{line}) \Rightarrow (a \in C \Leftrightarrow b \in C)\ \} \qquad \text{(Eqn 4.8)}$$

The *direct-mapping constraint*:

$$\forall (a \in P)\ \{|\,[a] \cap C\,| \leq \text{lineSize}\} \qquad \text{(Eqn 4.9)}$$

The *cache-size constraint* says that at most *cacheSize* memory locations in the domain of the trap-driven simulator can be accessed without causing a trap. The *line-size constraint* says that all the memory locations in the same memory line must all be accessible or not accessible as a group. Finally, the *direct-mapping constraint* says that at most one line from each memory equivalence class is accessible at a time. Thus, the set C is exactly the set of memory locations that can be accessed by the workload without causing a change of cache state and a corresponding kernel trap. The operation of the trap handler can now be simply stated as follows: *A trap handler maintains the validity of some set of access constraints during the run of a workload*.

The concept of access constraints is useful for a couple of reasons. First, if we can express the access constraints for a given caching structure, then we know that trap-driven simulations of such a structure are algorithmically possible. Second, access constraints help us to make qualitative

statements about the speed of a trap-driven simulation. Clearly, the larger the set C, the closer a simulation will be to a zero slowdown (if C = P, then the slowdown would be 0). Thus, if the access constraints show that trap-driven simulations are algorithmically possible and that the size of the set C is not so constrained as to excessively slow simulations, then we know that the trap-driven simulation of the given memory structure is feasible. On the other hand, an inability to define access constraints, or access constraints that make the set C too small,[2] are indications that trap-driven simulation is not possible.

We now give a detailed example of how a trap handler responds to an incoming trap in a way that satisfies the access constraints of Eqn 4.7 - Eqn 4.9. Figure 4.3 shows the trap that occurs after a reference to location 0xB024. This trap corresponds to the cache miss that would occur in an actual cache like the one shown in Figure 4.1. In an actual cache, the required line, starting at 0xB020, would be loaded from memory and inserted in cache set 02, displacing the line starting at 0x1020. The trap handler invoked by `tw_trap()` simulates this change in cache state by rearranging the access rights of the host physical memory in accordance with the access constraints and then records the new line in its cache data structures. These actions are depicted in Figure 4.3, which shows the trap handler removing access to region (0x1020 to 0x102F), the displaced cache line, and permitting access to region (0xB020 to 0xB02F), which represents the newly accessed line. The trap handler updates the cache data structure, counts the miss, and when it returns to the running workload, the access state of the host memory will be in conformance with all three access constraints. No more than 4 KB of physical memory addresses can be accessed without a kernel trap (Eqn 4.7, the cache-size constraint), addresses belonging to the same memory line all have the same access rights (Eqn 4.8, the line-size constraint), and at most one line from each memory equivalence class can be accessed (Eqn 4.9, the direct-mapping constraint).

Simulating Different Cache Sizes and Line Sizes

Until now we have made our descriptions concrete by selecting a specific cache size (4 KB) and line size (16 byte) for our examples. It is possible, however, to write a trap handler that enables

2. The meaning of “too small” depends on speed of a trap-driven simulator’s trap handlers, the miss ratio of a given workload in a given cache size, and the acceptable level of simulation speed. Trap-driven speed measurements, presented in Chapter 6, will help to make the meaning of “too small” more concrete.

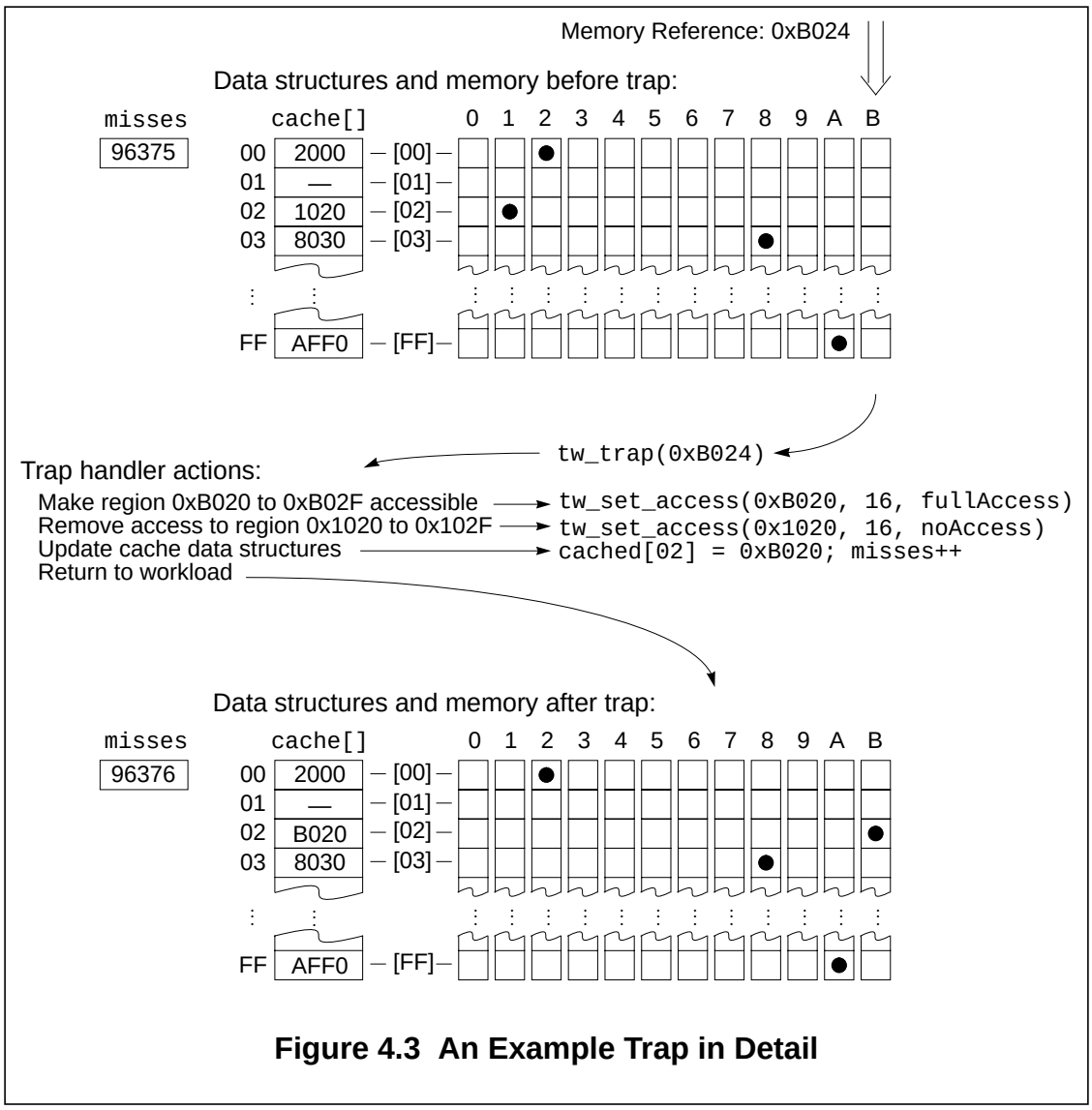


Figure 4.3 An Example Trap in Detail

cache size and line size to be freely varied. Figure 4.4 shows the pseudo code for a flexible trap-handler that works for different values of `cacheSets` and `lineSize`.

If the simulated cache size (i.e., the number of cache sets) is changed, then the size of the `cache[]` data structure changes accordingly (see Eqn 4.5) and the bits used to index this structure change as specified by Eqn 4.1 through Eqn 4.4. Memory will be re-partitioned into a different set of equivalence classes according to Eqn 4.6, and the access constraints (Eqn 4.7 - Eqn 4.9) will then define new permissible patterns of access to the host memory. The net result of these changes is that a larger simulated cache will enable more host memory locations to be accessed during the run of a workload, and less traps will occur during its run. Because our access

```
int cache[cacheSets];
int misses;

tw_trap(pa, va, type) {
   tw_set_access(pa, lineSize, fullAccess);
   if (tw_get_access(pa) != noAccess))
      tw_set_access(cache[pa_set], lineSize, noAccess);
   cache[pa_set] = pa_line;
   misses++;
}
```

Figure 4.4 A Trap Handler for Direct-mapped Cache Simulation

constraints were derived to model the caching constraints of a direct-mapped cache, this reduction in traps will exactly correspond to the reduction in misses in an actual larger cache. Similarly, if the cache data structure is reduced in size, the accessibility to host memory locations will also be constrained, which, in turn, would cause traps (and simulated misses) to increase in direct relation to the misses that an actual smaller cache would exhibit.

Changing the simulated line size also changes the equivalence-class partitioning of memory, and the re-application of the access constraints will result in a change of simulated miss counts that will correspond exactly to the misses of an actual cache with a new line size.

It is important to note that these changes in simulated cache size and line size are in no way restricted by the cache parameters of the host machine. If the full semantics of the `tw_set_access()` call are supported, then the simulated cache size and line sizes can be both smaller or larger than values of these parameters for the host machine.

Figure 4.5 depicts how changes in simulated cache parameters induce different partitions of host-memory equivalence classes. If the line size is doubled, but the cache size stays the same, then Eqn 4.5 implies that the number of cache sets (and thus the number of memory equivalence classes) will be halved (see top of Figure 4.5). The number of cache sets (and memory equivalence classes) is also halved if the size of the cache is halved, but the line size stays the same (see the middle of Figure 4.5). Although the number of equivalence classes is halved in both cases, notice that the members of these classes are not the same. In the first case, each 4-KB page is partitioned into 128 contiguous 32-byte lines that each belong to a different equivalence class. In the second

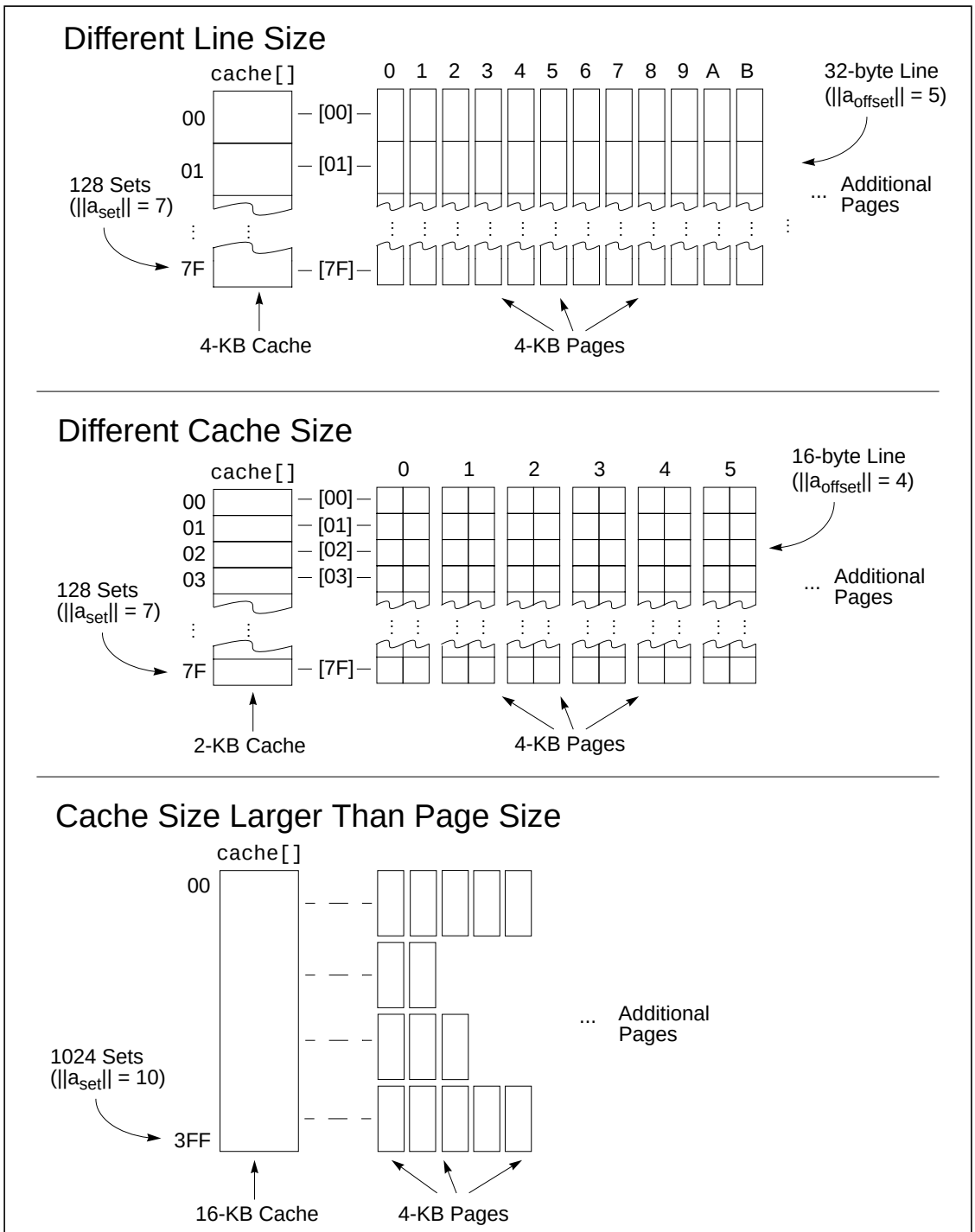


Figure 4.5 Cache Size, Line Size and Memory Equivalence Class Relationships

In the example shown above, the page size is always 4 KB, but cache sizes and line sizes vary. Changes in cache or line sizes re-partition memory into a different set of equivalence classes as given by Eqn 4.6.

case, however, each equivalence class is formed from 16-byte lines that are taken from the top and bottom half of each page.

In the cases considered so far, the cache size has been less than or equal to the host page size. If the cache is larger than the page, then the partitioning of memory into equivalence classes can depend on the allocation of virtual pages to physical pages, a situation that may vary from run to run of a workload (see bottom of Figure 4.5). Note that a trap-driven simulator is sensitive to these effects because it runs in an actual host operating system, alongside the workload being monitored. These page-allocation effects, which are a source of variability in measure cache performance, will be discussed in greater detail in Chapter 7.

Simulating Set Associativity

Direct-mapped caches sometimes exhibit high levels of conflict misses when two lines that are members of the same memory equivalence are simultaneously in use. One solution to this problem is to relax the direct-mapping constraint so that multiple lines from each memory equivalence class may be cached at the same time. Such a cache is commonly called an *n-way, set-associative cache*, and its associativity, *n*, is the number of memory lines that can be held in each cache set. Figure 4.6 shows an example 3-way, set-associative cache.

As with a direct-mapped cache, the memory address (a) used to access an *n*-way, set-associative cache is divided into three parts. The middle part (a_{set}) selects a cache set, which holds up to *n* lines from its associated memory equivalence class. The high-order bits of the address (a_{tag}) are compared against all *n* of the tags and valid bits for each line in the set. If there is a match, the appropriate line is selected and the low-order bits of the address (a_{offset}) are used to index the cache line. In an *n*-way, set-associative cache, the number of cache sets is a function of the cache size, the line size and the associativity:

$$\text{cacheSets} = (\text{cacheSize}) \,/\, (\text{lineSize} \cdot \text{cacheAssoc}) \qquad \text{(Eqn 4.10)}$$

The number of bits in each part of the address ($||a_{offset}||$, $||a_{set}||$, $||a_{tag}||$, and $||a_{line}||$) is then the same as the equations for a direct-mapped cache (Eqn 4.1 - Eqn 4.4). A set-associative cache also has the same cache-size and line-size access constraints as a direct-mapped cache (Eqn 4.7 and Eqn 4.8). It replaces the direct-mapping constraint, however, with the set-associativity constraint:

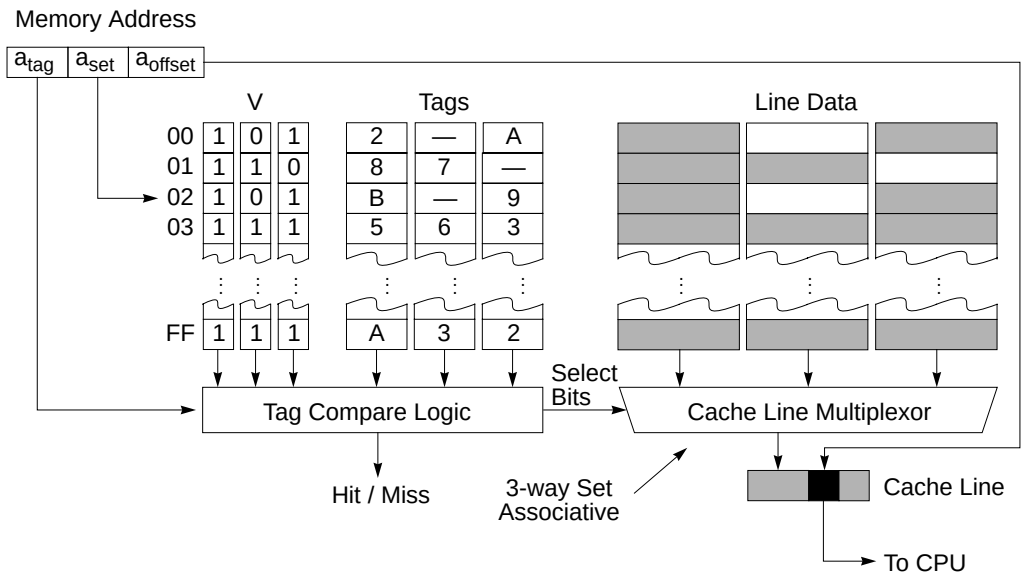


Figure 4.6 Set-associative Cache Hardware

This figure shows a 12-KB, 3-way set-associative cache, with a 16-byte line size. These parameters divide the cache into 256 sets, which implies that $||a_{offset}|| = 4$ bits and $||a_{set}|| = 8$ bits.

The *set-associativity constraint*:

$$\forall\,(a \in P)\ \{\,|\,[a] \cap C\,| \leq (\text{cacheAssoc} \cdot \text{lineSize})\,\} \qquad \text{(Eqn 4.11)}$$

Because we can express the access constraints for set-associative cache simulation, and because the size of the set C is no more constrained than it is for the simulation of a direct-mapped cache of the same size, we can conclude that trap-driven simulation of set-associative caches is just as feasible as it is for direct-mapped caches.

Figure 4.7 shows the data structures and host memory required to simulate the 3-way, set-associative cache of Figure 4.6. The left side of the figure shows a convenient data structure for set-associative cache simulation: a two-dimensional array (`cache[][]`), where the first index selects a cache set, and the second index selects a line within a given set. Depicted next to this data structure is the physical memory of the host machine. As with our previous examples, the equivalence classes that this particular cache structure induces on the host memory are shown as different rows of memory.

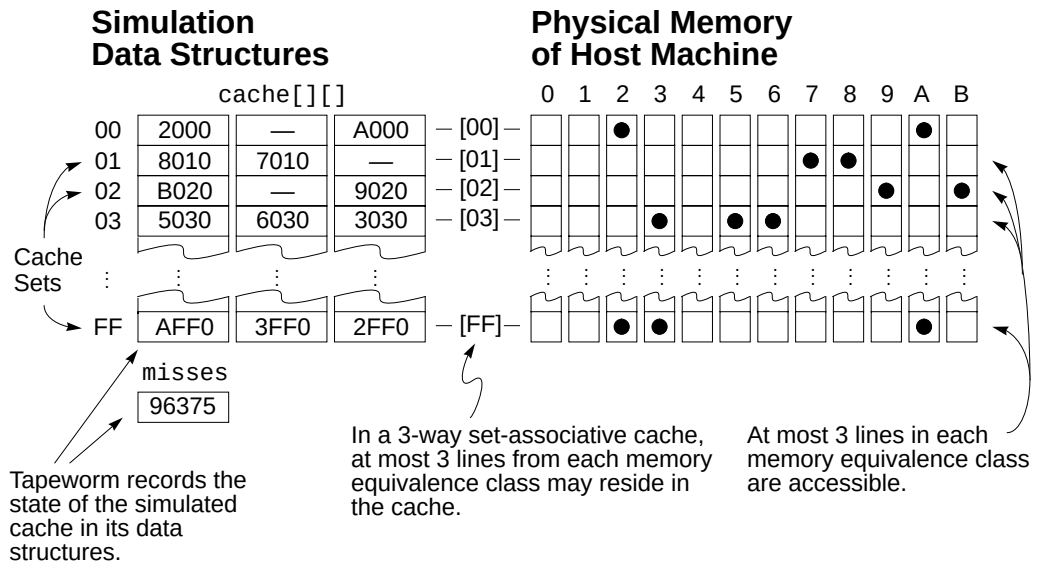


Figure 4.7 Data Structures and Host Memory for Set-associative Cache Simulation

A simple extension to Tapeworm's data structures and trap handlers enables the simulation of set-associative caches. In an *n*-way set-associative cache, up to *n* memory lines can reside in each cache set. Tapeworm's trap handlers emulate this relaxed constraint by permitting access to at most *n* lines in each memory equivalence class.

The example above shows a 3-way, 12-KB cache with 16-byte lines. Accesses to lines starting at 0x5030, 0x6030 or 0x3030 in equivalence class 03 are permitted, and correspond to simulated cache hits. Any other references to lines in this memory equivalence class, however, would result in a kernel trap and a call to `tw_trap()`. Such a trap corresponds to the cache miss that would occur in an actual 12-KB, 3-way I-cache.

Figure 4.7 is a snapshot of the simulator data structures and host memory at a specific time during the run of a workload. Notice that the access rights on the memory are in compliance with the access constraints for a 3-way, set-associative cache simulation; at most three memory lines from each equivalence class are accessible. This ensures that any memory reference that misses the simulated 3-way cache would also cause a trap to the simulator. Figure 4.8 shows an example trap handler for such an event. This trap handler makes the required line accessible to the workload and then brings the simulator data structures and host memory back into conformance with the access constraints. Notice that the handler is very similar to the direct-mapped cache handler (Figure 4.4), but differs slightly with the addition of the routine `tw_replace()`. The

```
int cache[cacheSets][cacheAssoc];
int misses;

tw_trap(pa, va, type) {
   tw_set_access(pa, lineSize, fullAccess);
   line = tw_replace(pa);
   if (tw_get_access(cache[pa_set][line]) != noAccess)
      tw_set_access(cache[pa_set][line], lineSize, noAccess);
   cache[pa_set][line] = pa_line;
   misses++;
}
```

Figure 4.8 A Trap Handler for Set-Associative Cache Simulation

purpose of this routine is to select a line to displace from the cache set. This selection was not required before because in a direct-mapped cache there is only one possible line that can be replaced. The choice of line to displace from a set-associative cache set after a miss is guided by a *replacement policy.* We will examine the simulation of several popular replacement policies in Section 4.1.2.

Simulating Full Associativity

A *fully-associative cache* is a special kind of set-associative cache with only one set that can hold any memory line. Because it has only one set, $||a_{set}|| = 0$, $a_{line} = a_{tag}$, and memory consists of only one equivalence class. A fully-associative cache has the same cache-size and line-size constraints as direct-mapped or set-associative caches (Eqn 4.7 and Eqn 4.8), but because there is only one memory equivalence class and only one cache set, there is no third caching constraint. A trap-driven simulator can thus simulate full associativity by simply ensuring that the total number of memory lines accessible to a running workload is always less than or equal to the simulated cache size. The necessary modifications to the trap handler of Figure 4.8 are minor, so we do not detail them here.

4.1.2 Basic Caching Policies

In addition to the basic cache structural parameters of size, line size and associativity, the performance of caches is also determined by their policies of operation. In this section, we

examine the trap-driven simulation of three important classes of caching policies: *replacement policies*, *write policies*, and *indexing policies*.

Simulating Random Replacement

As noted in the sections on simulating associativity, enabling more than one memory line from an equivalence class to be held in a cache set introduces the need for a replacement policy. Four of the most common replacement policies, *Random*, *First-in-first-out (FIFO)*, *Least-recently-used (LRU*) and *Not-most-recently-used (NMRU)* can all be implemented by a trap-driven simulator.

The first of these policies, Random replacement, is the easiest to implement. As its name suggests, the Random replacement policy simply picks, at random, the line to displace from the set. This is easily implemented in the trap handler of Figure 4.8 by writing the `tw_replace()` routine to return a random number in the range of 0 to N-1. Because this policy requires no additional simulation state, we need not redefine the access constraint equations.

Simulating FIFO Replacement

With a FIFO replacement policy, lines in a set are replaced in a round-robin order. This is also easily implemented in `tw_replace()` by maintaining a FIFO counter for each set in the cache. Each call to `tw_replace()` then returns the current value of the FIFO counter for the given set, and then increments the counter modulo the set size. Although the FIFO counters associated with each of the cache sets constitute part of the simulated memory state, these counters change only during a simulated cache miss. Thus, no additional access constraints are required to implement this policy in a trap handler.

Simulating LRU Replacement

The Random and FIFO replacement policies are both relatively easy to implement because they either do not add to the simulated cache state, or the added cache state only changes on cache misses. LRU replacement, on the other hand, is more difficult to implement because it depends on patterns of usage of the lines in a set. With LRU replacement, the lines in each set are continually sorted on the basis of order of last use, and on a miss the current least-recently-used line is selected for displacement.

A trap-driven simulation of these policies is only possible if we can formulate a new set of access constraints that define exactly those memory references that do not change simulation state. Notice that an access to any line other than the most-recently-used (MRU) one requires a resorting of the usage order, and thus a change in the simulated cache state. This means that only the MRU cache lines may be accessed by a running workload without causing a change in simulated cache state. If we define a subset M of the memory locations in our cache structure ($M \subset C$) to be these MRU cache lines, then the access constraints on this set are as follows:

The *cache-size constraint*:

$$|M| \leq (\text{cacheSize} / \text{cacheAssoc}) \qquad \text{(Eqn 4.12)}$$

The *line-size constraint*:

$$\forall (a, b \in P)\ \{\ (a_{line} = b_{line}) \Rightarrow (a \in M \Leftrightarrow b \in M)\ \} \qquad \text{(Eqn 4.13)}$$

The *most-recently-used constraint*:

$$\forall (a \in P)\ \{|\,[a] \cap M\,| \leq \text{lineSize}\} \qquad \text{(Eqn 4.14)}$$

Notice that these access constraints are very similar to those of a direct-mapped cache (Eqn 4.7 - Eqn 4.9). The main difference is that the total number of cache lines that are accessible at any point in time during the simulation is equal to the cache size divided by the associativity. From these access constraints, we can conclude that trap-driven simulation of an *n*-way, set-associative cache with LRU replacement is possible, and its speed will be proportional to the trap-driven simulation of a direct-mapped cache of 1/*n*-th the size. For example, the simulation of a 12-KB, 3-way, set-associative cache should be about as fast as the simulation of a 4-KB, direct-mapped cache.[3] Note, however, that simulation of a fully-associative cache with LRU replacement is probably not possible because only one line in the entire cache would be accessible at a time. Large fully-associative structures are, however, rarely implemented in hardware, so this is not a severe limitation.

Figure 4.9 shows trap handling for simulation of LRU replacement. As before, a set-associative cache can be represented as a two-dimension array, but now the lines in each row (set) are sorted according to order of use. Unlike previously simulations, in which all lines in the simulated cache are accessible in host memory, the LRU trap handler only permits access to the

3. We will verify this with empirical data in Chapter 6 on trap-driven simulation speed.

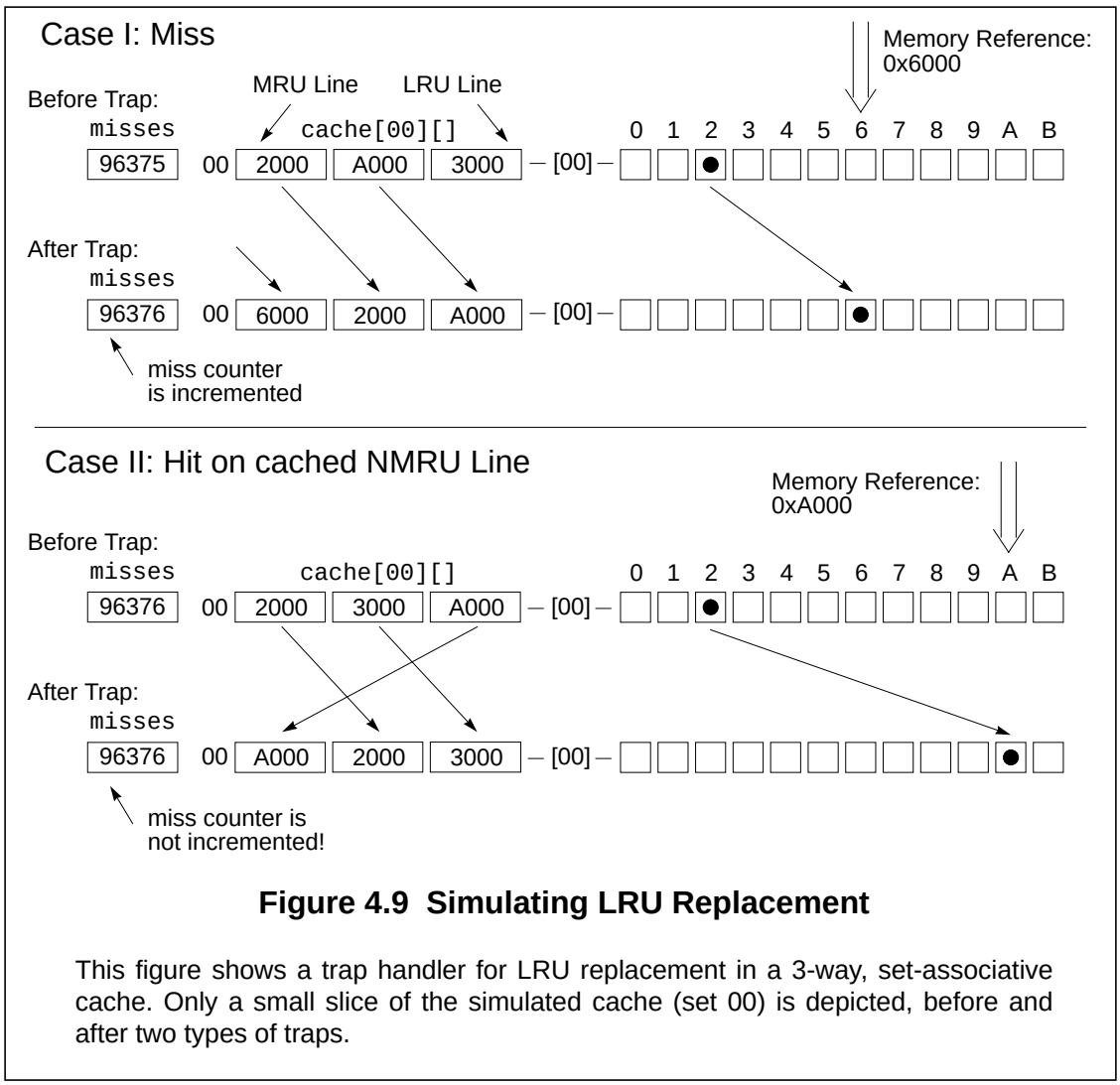


Figure 4.9 Simulating LRU Replacement

This figure shows a trap handler for LRU replacement in a 3-way, set-associative cache. Only a small slice of the simulated cache (set 00) is depicted, before and after two types of traps.

MRU line in each set, in accordance with Eqn 4.12 through Eqn 4.14. Thus, not all traps correspond to simulated cache misses; some of the traps will only require a re-sorting of the lines in a cache set to reflect current order-of-usage status. The two possible cases are shown in Figure 4.9.

For Case I, a simulated miss, the missing line (starting at 0x6000) is cached as the MRU entry, and is made accessible. The previous MRU line (starting at 0x2000) is shifted into second place and access to it is removed. The LRU line (starting at 0x3000) is displaced, and the line starting at 0xA000 becomes the new LRU line. Finally, the miss is counted.

In Case II, the trap is due to an access to a cached, but not-most-recently-used line. This is a cache hit, but the cache state must be updated to reflect the new order of line use. The figure shows how this sorting is accomplished and how access is removed from the previous LRU line (starting at 0x2000) and given to the new LRU line (starting at 0xA000). The miss counter is not incremented in this case.

Simulating NMRU Replacement

Like LRU replacement, the NMRU policy also requires maintenance of usage order, but only to a partial extent. With NMRU, it is only necessary to know that the line being replaced is not the most-recently-used line. In principle, we could just use the same access constraints implemented by the LRU trap handler, and then randomly pick from the NMRU entries in the event of a miss. However, there is an optimization that relaxes the access constraints and increases simulation speeds. This optimization works by enabling access to all lines in a cache set, except one: the candidate for NMRU replacement. We partition the cache C into two sets, N and $\overline{N}$, such that $C = N \cup \overline{N}$ and $N \cap \overline{N} = \varnothing$. The set of all NMRU candidates lines is N, and the rest of the cached lines are $\overline{N}$. Only the memory lines in $\overline{N}$ are accessible according to the following access constraints:

The *cache-size constraint*:

$$\left|\overline{N}\right| \leq (\text{cacheSize})\ (\ (\text{cacheAssoc} - 1) \not/ \text{cacheAssoc}) \qquad \text{(Eqn 4.15)}$$

The *line-size constraint*:

$$\forall (a, b \in P)\ \{\ (a_{line} = b_{line}) \Rightarrow (a \in \overline{N} \Leftrightarrow b \in \overline{N})\ \} \qquad \text{(Eqn 4.16)}$$

The *not-most-recently-used constraint*:

$$\forall (a \in P)\ \{\left|[a] \cap \overline{N}\right| \leq (\text{lineSize} \cdot (\text{cacheAssoc} - 1))\ \} \qquad \text{(Eqn 4.17)}$$

These access constraints are very similar to those of a simple set-associative cache, with the added restriction that one cached line, the NMRU candidate, may not be accessed. When a reference causes a trap to the cache set, these constraints ensure that the reference could only have been to the cached NMRU candidate, or to some other, uncached line. Figure 4.10 shows how these two cases are handled by an NMRU trap handler for a specific cache set (00). Notice, first, that in this 3-way, set-associative cache, 2 lines from the set are accessible in the host memory. Only access to the NMRU candidate (or some other uncached line) will cause a trap to the

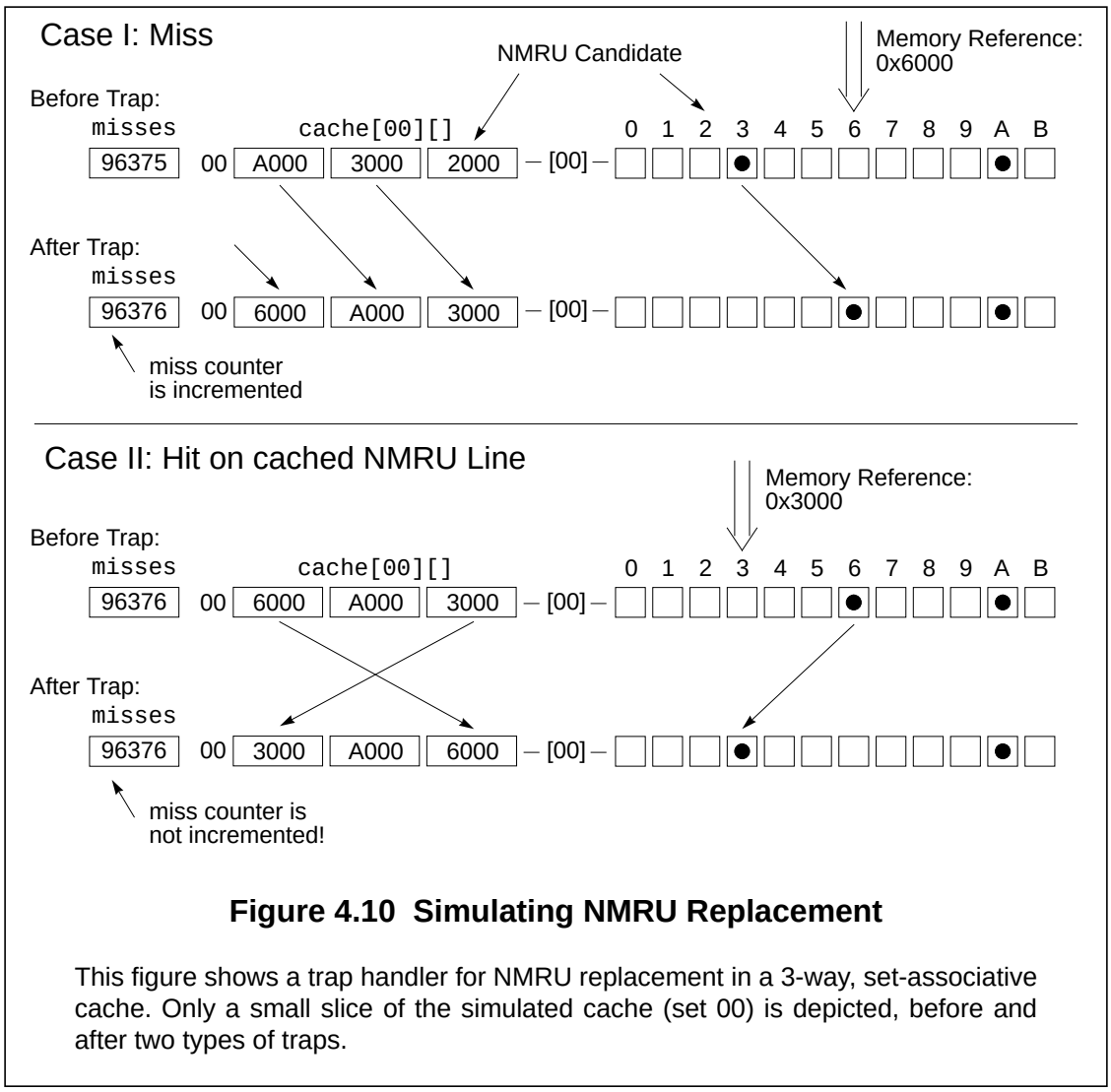


Figure 4.10 Simulating NMRU Replacement

This figure shows a trap handler for NMRU replacement in a 3-way, set-associative cache. Only a small slice of the simulated cache (set 00) is depicted, before and after two types of traps.

simulator. The array representing the cache places the NMRU candidate in array entry `cache[00][2]` (this is an element of N), and the other cache lines, in no particular access order, are held in array entries `cache[00][0..1]` (these are elements of $\overline{N}$).

For Case I, a simulated miss, the NMRU candidate is displaced, and a new NMRU candidate is selected, at random, from the set $\overline{N}$. We are guaranteed that this line is not the most-recently-used line, because the reference that caused the miss is now the MRU line. Access is removed from the new NMRU candidate, the entries in the `cache[][]` array are re-ordered, and the miss is recorded in `misses`.

In Case II, the handler detects that the trap is due to an access to the cached NMRU candidate by finding this line in the cache data structure. The NMRU candidate is no longer a candidate for replacement (because it is now the MRU line), and because the reference does not correspond to a simulated cache miss, the `misses` counter is not incremented. Instead, a new NMRU candidate is selected at random from $\overline{\text{N}}$, and exchanges position with the old NMRU candidate in the cache data structure. This swap is accompanied by a shift in host-memory access: the new NMRU candidate is made inaccessible, and the old NMRU candidate (which is now the MRU line) is made accessible.

Notice from the access-constraint equations that we would expect NMRU simulation to perform better than LRU simulation on caches of the same size. In particular, the speed of an *n*-way, set-associative simulation of the NMRU policy would be proportional to the simulation of an (*n*-1)-way, set-associative cache of (*n*-1)/*n*-th the size. For example, the simulation of a 12-KB, 3-way, set-associative cache should be about as fast as the simulation of an 8-KB, 2-way, set-associative cache. Unlike with LRU replacement, simulating fully-associative caches with NMRU replacement is possible because all but one line in the cache (the single NMRU candidate) is accessible to the running workload.

Write Policies

We have so far only considered read references to cached information. This is entirely adequate for simulating I-cache and TLBs, because instructions and page-table entries are typically cached with read-only access. A complete simulation of D-caches, however, requires a consideration of writes. Figure 4.11 depicts common hardware for supporting writes. The cache shown in the left figure supports a *write-back* write policy, while the cache depicted in the right figure supports a *write-through* write policy.

With a *write-back policy*, a write that hits in the D-cache modifies the D-cache line and marks it dirty by setting its dirty (D) bit. A write that misses the cache is typically first loaded into the cache as would occur with a read miss, and then the line is modified and the dirty bit is set. When the line is subsequently displaced by a future cache miss, the dirty line is written back to memory to maintain coherence. In a write-back cache, performance can be lost if these write-backs increase cache miss penalties.

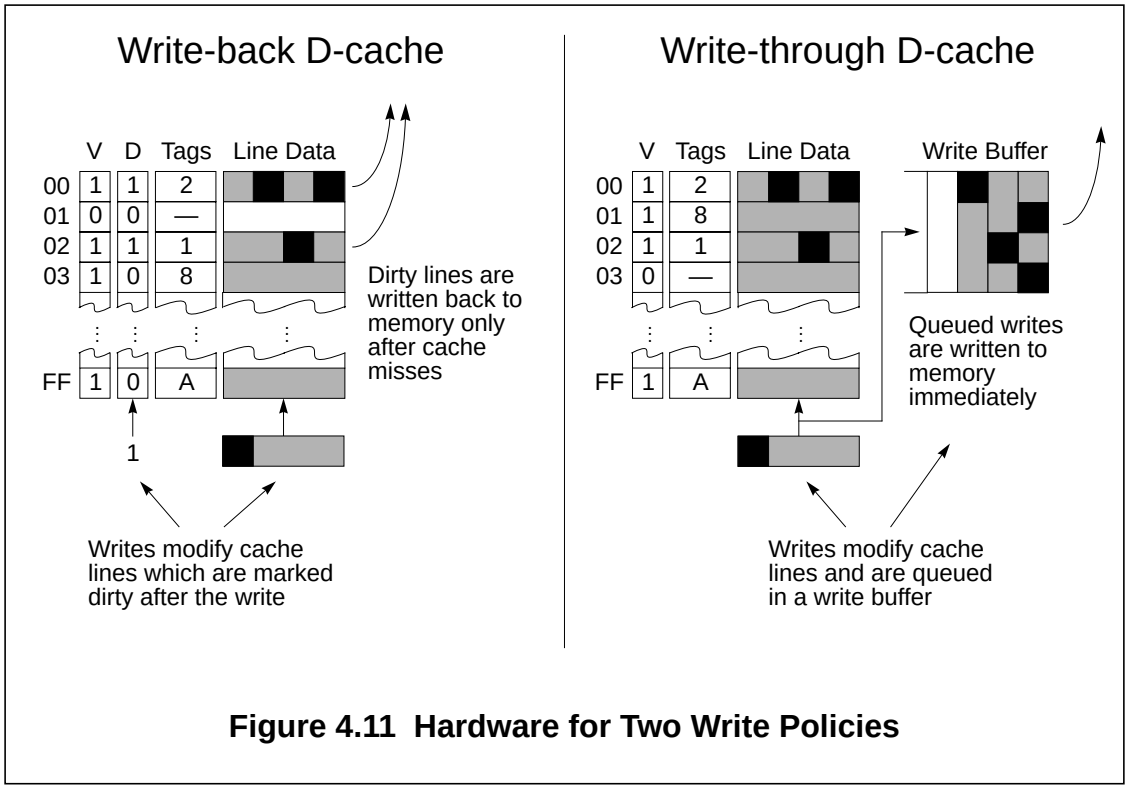


Figure 4.11 Hardware for Two Write Policies

With a *write-through policy*, writes update memory immediately. A write buffer is often used to queue writes so that the processor can continue execution. If the write hits in the cache, then the appropriate cache line is also updated, but if the write misses the cache, then there are two options. Either the line can be loaded into the cache and then modified, or no action can be taken. The first method is called a *write-allocate policy* (notice that write-back policy, as described above, is also a write-allocate policy). In a write-through cache, performance is lost if processor writes must be stalled due to a full write buffer.

In the next section we will show how trap-driven simulation can model a write-back write policy. Unfortunately, simulation of a write-through write policy appears not to be possible with trap-driven methods. The reason is due to the inherently different nature of a write buffer compared to that of a traditional cache. Caches are *memories* whose state only change in response to a read or write memory reference. Write buffers, on the other hand, are *queues* whose state gradually change over time. That is, after a write request is placed in a write buffer (thus changing its state), the request is serviced as a background operation. When the operation is completed, the request is removed from the buffer (thus changing its state again). The exact moment that these changes in write buffer state occur is dependent on many factors, such as bandwidth to main

memory, the size of the write request, and previously-queued writes. Unfortunately, these events do not fit our access-constraint model, so we are unable to design a trap handler to model them.

Simulating a Write-back Write Policy

As with all forms trap-driven simulation, a trap handler for write-back simulation must constrain access to the host memory so that a trap occurs after any memory reference that changes the simulated cache state. For a write-back cache, simulation state includes not only the starting addresses of cache lines, but also the dirty/clean status of each line. In other words, a given memory line can be in one of three states: *uncached*, *cached-clean*, or *cached-dirty.* Transitions between states are determined by the following rules:

- Case I: If a line is *uncached*, then both reads and writes to its memory locations cause cache state to change. A read reference makes the line cached-clean, and a write reference makes the line cached-dirty.

- Case II: If the line is *cached-clean*, then changes in cache state depend on the type of memory reference. If it is a read, then cache state doesn't change, but if it is a write, the dirty bit if the line must be set to change its state to cached-dirty.

- Case III: If a line is *cached-dirty*, then both reads and writes cause no change to cache state at all. That is, both reads and writes to the line are cache hits.

We now formalize these into a new set of access constraints suitable for trap-driven simulation. We first divide the set of memory locations representing the simulated cache, C, into two non-overlapping sets, R and W (s.t. $C = R \cup W$ and $R \cap W = \varnothing$), such that R represents all host memory locations that can be accessed with read references, without causing a trap to the simulator, while W represents all host memory locations that can be read or written without a trap. Our access constraints for a direct-mapped write-back cache are then:

The *cache-size constraint*:

$$|R \cup W| \leq \text{cacheSize} \qquad \text{(Eqn 4.18)}$$

The *line-size constraint*:

$$\forall (a, b \in P)\ \{\ (a_{line} = b_{line}) \Rightarrow (a \in R \Leftrightarrow b \in R)\ \}$$
$$\forall (a, b \in P)\ \{\ (a_{line} = b_{line}) \Rightarrow (a \in W \Leftrightarrow b \in W)\ \} \qquad \text{(Eqn 4.19)}$$

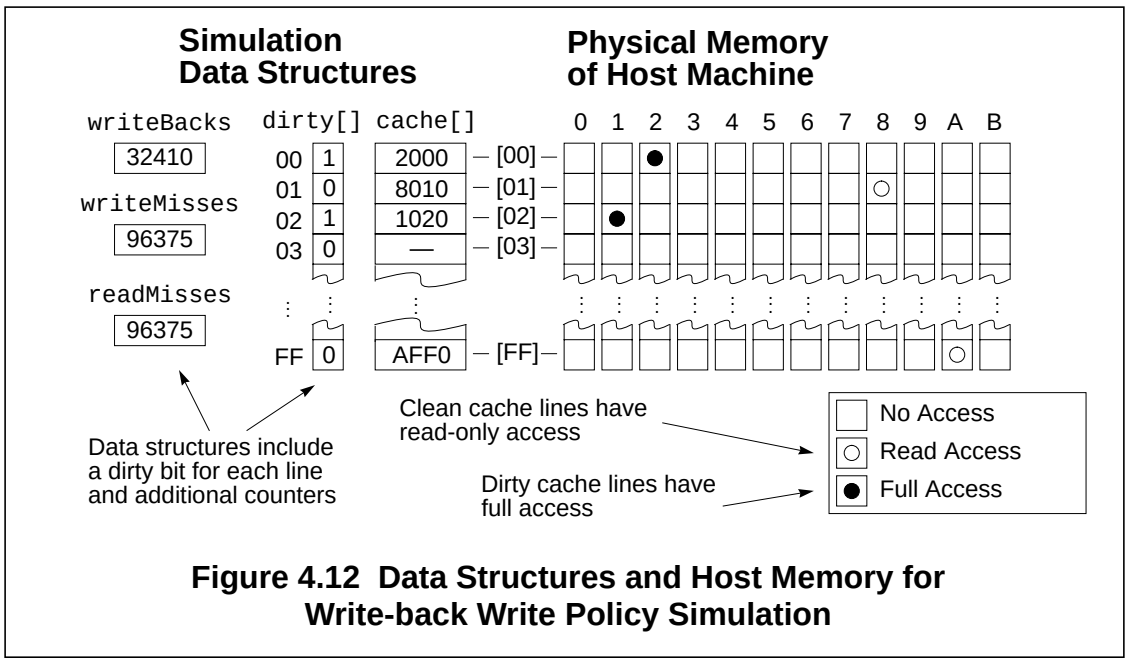


Figure 4.12 Data Structures and Host Memory for Write-back Write Policy Simulation

The *direct-mapping constraint*:

$$\forall (a \in P)\ \{\,|\,[a] \cap (R \cup W)\,| \leq \text{lineSize}\} \qquad \text{(Eqn 4.20)}$$

The cache-size constraint says that the total number of memory lines with any type of access (read-access or full-access) cannot exceed the size of the simulated cache. The line-size constraint says all parts of a given memory line must have the same access state (no-access, read-access, or full-access). Finally, the direct-mapping constraint says that at most one line from each memory equivalence class is accessible (either read-access or full-access) at a time.

The existence of these access constraints implies that trap-driven simulation of write-back, direct-mapped caches is possible. A minor modification to the direct-mapping constraint shows that write-back, *n*-way, set-associative cache simulation is also possible:

The *set-associativity constraint*:

$$\forall (a \in P)\ \{\,|\,[a] \cap (R \cup W)\,| \leq (\text{cacheAssoc} \cdot \text{lineSize})\ \} \qquad \text{(Eqn 4.21)}$$

Figure 4.12 shows data structures for trap-driven write-back simulation, alongside a snapshot of host memory access state that complies with the access constraints for direct-mapped, write-back cache simulation (Eqn 4.18 - Eqn 4.20). The main additions to the simulator data structures are an array (`dirty[]`) that marks the clean/dirty state of cache lines, and three new variables

```
int dirty[cacheSets], cache[cacheSets];
int writeBacks, writeMisses, readMisses;

tw_trap(pa, va, type) {

   if (cache[pa_set] == pa) {
      tw_set_access(pa, lineSize, fullAccess);
      dirty[pa_set] = 1;
   }

   else {
      if (dirty[pa_set])
         writeBacks++;

      switch (type)
      {
         case dataLoad:
            tw_set_access(pa, lineSize, readAccess);
            dirty[pa_set] = 0;
            readMisses++;

         case dataStore:
            tw_set_access(pa, lineSize, fullAccess);
            dirty[pa_set] = 1;
            writeMisses++;
      }

      if (tw_get_access(pa) != noAccess) {
         tw_set_access(cache[pa_set], lineSize, noAccess);
         cache[pa_set] = pa_line;
      }
   }
}
```

Figure 4.13 A Trap Handler for Write-back Write Policy Simulation

`writeBacks`, `writeMisses`, and `readMisses`) that count the number of dirty cache lines written back to memory, and the two types of cache misses.

Figure 4.13 shows the trap handler that maintains these simulator data structures under the access constraints. The handler first checks if the line is cached, a situation that can only occur if a write reference is made to a clean, cached line. In this case, the handler gives full access to the line, marks it dirty, and immediately returns to the running workload. If the line is not cached, then a genuine miss occurred, and the handler counts a write-back if the line is dirty. Then, the handler adjusts the members of the sets R and W by setting the appropriate level of access for the line

(read-access or full-access) in accordance with the access constraints. Finally, the handler sets the dirty status of the line, counts the type of miss, removes access to the displaced cache line, and updates the `cache[]` structure.

Simulating Virtual and Physical Indexing Policies

The trap handlers that we have considered so far have all used the physical address, `pa`, to index the cache data structure, and therefore simulate physically-indexed caches. A common alternative to this cache indexing policy is virtual indexing.

Because the trap handlers of a trap-driven simulator are designed to run in the OS kernel of a host machine, they have full access to both virtual and physical address, and can therefore easily switch between both indexing policies. This ability makes it possible to explore the effects of an operating system's virtual-to-physical page allocation policy on cache performance. As we will see in Chapter 7, these page-allocation policies can affect caches because different allocations of virtual pages to physical page frames from run to run of a given workload can cause its memory to reside in different cache locations.

Most of the trace-driven simulators discussed in Chapter 2 are unable to be used in such experiments because they are limited to either virtual addresses or physical addresses, or can't easily collect multiple samples of the virtual-to-physical mappings from run to run of a workload.

4.1.3 Complex Memory Systems

In the previous sections, we have shown how to simulate a range of basic cache structures with different policies of operation. In this section, we discuss how these "building block" cache structures can be composed to simulate more complex memory systems. For the purposes of trap-driven simulation, the composition of caches can be divided into two basic categories. Caches can either be composed in series, forming a memory hierarchy, or they can be composed in parallel. There are many examples of both forms of cache composition and as we will see in the next two sections, all can be simulated with a trap-driven simulator

Simulating Caches in Series

The increasing gap between processor and main memory speeds is making it increasingly difficult to adequately service memory requests with just a single level of cache placed between the processor and main memory. Many memory systems, therefore, link a series of successively larger (but slower) cache memories into a memory hierarchy [Short88, Baer87, Baer88, Przybylski89, Przybylski90, Happel92, Kessler91, Olukotun91, Jouppi94, Wang89]. In such a system, the processor first references the fastest cache in the hierarchy, and on a miss tries to find the requested data by referencing each of successively larger caches in the hierarchy. The overall performance of such a *multi-level cache hierarchy* depends on the number of misses and the miss penalty in each cache in the hierarchy.

The key to simulating the caches in a multi-level hierarchy is to note that they typically exhibit an *inclusion property*, meaning that each cache in the hierarchy is a proper subset of the next largest cache in the hierarchy.[4] To be more precise, we define a collection of sets C_1, C_2, ..., C_n, consisting of host memory locations that represent the caches in the memory hierarchy to be simulated. Each of these sets has an associated collection of access constraints derived from the parameters of the cache that they represent. Following the inclusion property of multi-level caches, these sets $C_1 \subset C_2 \subset ... \subset C_n$, satisfy the following additional properties:

- Host memory locations in C_1 can be accessed without causing any change in simulated cache state all along the cache hierarchy.
- Accesses to host memory locations in C_2 may change the state of the simulated first-level cache, but all other caches in the hierarchy will be unchanged.
- Accesses to host memory locations in C_n may change the state of the simulated first-level to (n-1)-th level caches, but the n-th level cache will be unchanged.

Given these conditions, a multi-level cache can be simulated by a trap handler that implements the access constraints for just the smallest set, C_1. Because of the inclusion relationship between each of these sets, any memory references that cause a change of state anywhere higher in the hierarchy must also result in a trap. Given this, the C_1 trap handler can maintain a data structure for the entire multi-level cache hierarchy, and count the number of misses at each level.

4. This property is sometimes slightly relaxed, as in the case of write-back caches.

An alternative method is to simulate the different levels of the memory hierarchy in multiple simulation passes. For example, if a designer wishes to simulate a two-level cache hierarchy, then the misses from the first-level cache can be counted in one simulation pass where the trap handlers implement access constraints for the C_1 set. Then, a second simulation pass can be performed in which the trap handlers implement the access constraints for the C_2 set. Because the set C_2 is larger than C_1, the second pass will run more quickly than the first.

The advantage of the second method is increased *overall* simulation speeds. To understand why this is so, consider the typical design constraints that a memory-system designer must contend with. Often, because of access time requirements, a first-level cache is highly constrained; its size is typically small and it has low associativity. A second-level cache, on the other hand, is usually much less constrained because it is not part of the critical execution path of a processor. A designer faced with these constraints typically only needs to simulate only a small set of first-level cache configurations, but is free to explore a much wider range of second-level configurations. Because the simulation of a multi-level cache can be separated into two phases as described above, the simulation task can be divided into two parts: (1) slow simulations of a small set of first-level cache configurations, and (2) fast simulations of a large set of second-level cache configurations. The miss counts from these two sets of simulation passes can then be arithmetically combined to obtain an overall estimate of simulation performance for the complete memory hierarchy. Performing the multi-level simulations in this manner is much faster than if each possible combination of first-level and second-level cache were simulated at speeds determined by the access constraints for C_1.

Simulating Caches in Parallel

Ideally, a cache is large, highly associative, fast, and supports multiple access ports. It is, of course, difficult to realize such a cache in actual hardware, but it is sometimes possible to combine two different cache structures into a single hybrid structure that exhibits the strengths of both. For example, a relatively large and fast direct-mapped cache can be accessed in parallel with a relatively small and fast fully-associative cache. The resulting hybrid structure benefits from the size of the direct-mapped cache, and the associativity of the smaller fully-associative cache. Such cache structures have been variously called *victim caches* [Jouppi90], *assist caches* [Gwennap94] and *hybrid caches (or TLBs)* [Jacob95]. Another common example of caches that are accessed in parallel are *split* I- and D-caches, which are an alternative to *unified* caches. By splitting a unified

cache into two caches, the number of access ports can be doubled and parameters of the two caches can be independently tuned.

Two caches accessed in parallel are implemented in a trap-driven simulator by maintaining a separate data structure for each cache and then enforcing two different sets of access constraints, one for each cache. Because the caches would be accessed in parallel in a real implementation, it is possible to permit the memory locations of both caches to be accessible simultaneously during the trap-driven simulation. This is ideal for a trap-driven simulator, because an increase in the number of host memory locations that are simultaneously accessible increases simulation speeds. If it makes sense to do so, a variety of policies for moving cached information between the two caches (as in a hybrid cache) can also be implemented, provided that the policy is invoked only on cache misses.

4.1.4 Metrics

All of the algorithms that we have presented so far provide the same type of performance metric, a count of some event that affects memory-system performance. We have shown that trap-driven simulation can count *readMisses*, *writeMisses*, and *writeBacks*, which can be used directly to compare the relative performance of two different cache structures. A cache that exhibits fewer misses or write-backs than another cache, when running a given workload, clearly provides better performance.

Unfortunately, simple event counts in isolation can’t be used to compare the performance of different workloads running in the same cache because each may make a different number of total references. Another problem with simple event counts is that they do not provide a good measure of the relative importance of the memory system on overall system performance. It is usually necessary to combine these counts with other information to obtain a more complete picture of overall performance. In this section, we examine how other common performance metrics can be obtained through trap-driven simulation.

Miss Ratios

A common and intuitive measure of cache performance is the *miss ratio (MR)*, which is the number of cache misses divided by the number of times the cache is referenced. Some example miss ratio computations include:

For I-caches:

$$MR_{inst} = (instMisses) / (instRefs) \quad \text{(Eqn 4.22)}$$

For D-caches:

$$MR_{data} = (readMisses + writeMisses) / (dataRefs) \quad \text{(Eqn 4.23)}$$

For TLBs:

$$MR_{tlb} = (instMisses + dataMisses) / (totalRefs) \quad \text{(Eqn 4.24)}$$

where totalRefs = (instRefs + dataRefs). To compute these ratios in a trap-driven simulation environment, we use the `tw_get_counts()` call (see Chapter 3), which returns the number of instruction or data references made to host memory pages registered with the simulator. The advantage of miss ratios over simple event counts is that they can be used to compare the performance of two different workloads running in the same cache because the different number of references made by each workload is factored into the computation.

Traffic Ratios

In multiprocessor systems that share a common memory-system bus, an important metric is the *traffic ratio (TR)*, which measures the affect of a cache on memory-bus references. The traffic ratio is the number of bytes transferred to and from main memory with a cache, divided by the number of bytes transferred to and from main memory without a cache. If each cache miss or write-back transfers *lineSize* bytes of data to or from memory, and if *refSize* is the average number of bytes in a simple instruction or data memory reference to main memory, then traffic ratio can be computed from the miss ratio as follows:

$$TR = MR \cdot (blockSize / refSize) \quad \text{(Eqn 4.25)}$$

Because *lineSize* and *refSize* are typically constant and fixed values for a given memory system, the computation of traffic ratios poses no problems for a trap-driven simulator.

Misses per Instruction

The miss ratios of different caches in a memory system (I-cache, D-cache, TLB, second-level cache, etc.) are not directly comparable because they do not use a common denominator in their computation. For example, if an I-cache and D-cache both exhibit a 0.10 miss ratio, this does not mean that they have an equal impact on overall performance; the I-cache will exhibit many more total misses because I-caches are typically accessed much more frequently than D-caches. This problem can be avoided by reporting misses for different caches against a common denominator, the total number of instructions executed. Some example computations of this metric, *misses per instruction (MPI)*, are shown below:

For I-caches:

$$MPI_{inst} = (instMisses) / (instExec) \quad \text{(Eqn 4.26)}$$

For D-caches:

$$MPI_{data} = (readMisses + writeMisses) / (instExec) \quad \text{(Eqn 4.27)}$$

For TLBs:

$$MPI_{tlb} = (instMisses + dataMisses) / (instExec) \quad \text{(Eqn 4.28)}$$

As with the computation of miss ratios, a trap-driven simulator relies on the `tw_get_counts()` call to obtain the value of *instExec*, the total number of instructions executed by the workload from pages in the simulator's domain.

Cycles per Instruction

Although very useful for comparing the relative performance of different workloads in different types of caches, the MPI metric still omits an important piece of information, the amount of *time* a workload spends waiting for a memory system. To compute this value, the average number of *cycles per miss (CPM)* must be known. Given this value, *cycles per instruction (CPI)* for a memory system component can be computed as follows:

$$CPI = CPM \cdot MPI \quad \text{(Eqn 4.29)}$$

Obtaining an accurate estimate of cycles per miss depends very much on the features of the memory system being simulated. In a simple machine that completely stalls on a cache miss, and

waits until the entire line has been loaded into the cache, the average CPM is simply the number of cycles required to load a cache line. This, in turn, is a function of the cache line size (in bytes), the fetch latency (in cycles), and the refill bandwidth (in bytes per cycle) of the next level in the memory hierarchy:

$$\text{CPM} = \text{latency} + (\text{lineSize} / \text{bandwidth}) \qquad \text{(Eqn 4.30)}$$

In a more sophisticated memory system, estimates of CPM can be more difficult. For example, if the cache implements a *bypass* mechanism that enables the processor to resume execution after the required bytes have been loaded into the cache, then the average CPM value must be computed as a weighted average of miss penalties:

$$\text{CPM} = \text{latency} + \sum_{i=1}^{\text{lineSize}} p_{\text{access}}(i) \cdot \left\lceil \frac{i}{\text{bandwidth}} \right\rceil \qquad \text{(Eqn 4.31)}$$

where $p_{access}(i)$ is the probability that the missing access is to the i-th byte of a cache line and:

$$\sum_{i=1}^{\text{lineSize}} p_{\text{access}}(i) = 1 \qquad \text{(Eqn 4.32)}$$

A trap-driven simulator can estimate this probability density function by maintaining a histogram of miss counts. A counter can be associated with each byte (or word) on a line, and after each miss, the counter that corresponds to the part of the line that is required first is incremented.

Another refill policy is *critical-word-first*, in which the cache controller fetches the required part of the line first. Assuming that the bandwidth of the refill is high enough to load the remainder of the line into the cache before its other parts are accessed, the CPM for this refill policy can be approximated as:

$$\text{CPM} = \text{latency} \qquad \text{(Eqn 4.33)}$$

As memory systems become more complex, it becomes increasingly difficult to accurately estimate values for CPM. For example, a memory system can *prefetch* instructions or data [Farrens89, Hill87, Smith78, Smith92, Pierce95], caches can be designed to *pipeline* accesses [Jouppi90, Olukotun92, Palcharla94], and process multiple outstanding misses in parallel (a *non-blocking* or *lockup-free* cache), and dirty write-backs can be queued and processed as a background operation [Smith82, Kessler91]. Each of these memory-system optimizations can reduce the average value of CPM in a complex way that is dependent on many factors, including

the memory-access patterns of a given workload, and the execution engine of the CPU. It is difficult to accurately account for these effects in a trap-driven simulator.

4.2 Adapting Methods of Trace-driven Simulation

In the survey of Chapter 2, we discussed several trace-driven simulation methods that enable a memory-system designer to explore a design space more quickly. In this section we show how four of these methods, *set sampling*, *time sampling*, *multi-configuration simulation*, and *address filtering* can be adapted to trap-driven simulation.

4.2.1 Set Sampling

With set sampling, only a portion of the sets in a cache are simulated and the performance in these sets is used as an estimate of the performance of the entire cache.

With trace-driven simulation, set sampling is implemented by filtering a trace of all memory references that do not access the sampled cache sets. The resulting filtered trace must hold exactly the memory references that access the sampled cache sets, no more and no less, and the remaining addresses must also be in the same order as in the original trace [Puzak85; Kessler91]. Filtering a trace to obtain a set-sampled trace is a *pre-processing* step that itself takes time. The advantage of the set-sampled trace, once obtained, is that it can be used repeatedly in subsequent simulations that will complete more quickly because there are less total addresses to process.

An operation equivalent to set sampling can be performed by a trap-driven simulator *during* a simulation run. This is accomplished by subjecting only a subset of host-memory equivalence classes to the access constraints of a given cache structure. Figure 4.14 shows how this is implemented for the example of a direct-mapped cache simulation. In the example shown, the even memory equivalence classes [00], [02], ..., [FE] conform to the usual direct-mapping constraint (at most one memory line accessible per cache set), but in the odd equivalence classes [01], [03], ..., [FF], all memory locations are fully accessible. This relaxed pattern of access right guarantees that the trap handlers will only see misses from the sampled, even sets. In this way, a trap-driven simulator can automatically filter references to the unsampled sets at the full speed of

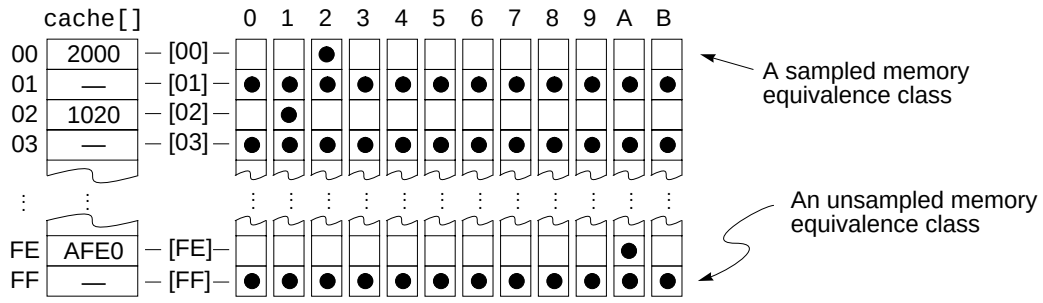


Figure 4.14 Set Sampling in a Trap-driven Simulator

the host hardware. As a result, the trap-driven simulation speeds increase in inverse proportion to the fraction of sets sampled.

With both trace-driven and trap-driven set sampling, the miss counts from the simulated sets can be used to compute estimates of overall performance. An estimated value of overall MPI, for example, would be computed as follows:

$$MPI_i = misses_i / instExec_i \qquad \text{(Eqn 4.34)}$$

where MPI_i is the MPI estimate for some set sample, $misses_i$ is the number of misses exhibited by the cache sets in the sample, and $instExec_i$ is the fraction of total instructions used to compute the MPI estimate. In his dissertation, Kessler suggests several options for computing the fraction $instExec_i$, and shows that simply multiplying the total number of instructions, instExec, by the fraction of sets sampled is the superior method [Kessler91]. This is convenient for trap-driven simulation, because it requires no special host hardware support beyond what we have already defined in `tw_get_counts()`.

4.2.2 Time Sampling

Set sampling selects memory references based on which cache sets they map into. Set sampling, therefore, samples addresses in space. Another common sampling option is to sample addresses on the basis of which region in time they appear [Laha88; Kessler91].

In a trace-driven simulator, time sampling is implemented by a different form of trace filtering. A complete, contiguous set of N addresses is selected from a complete trace to form a time sample, and then this trace sample is fed to a trace-driven simulator to obtain an estimate of MPI over the sampled region of program execution. Multiple sets of time samples, each containing all the references in a window of program execution can be collected, but each sample is fed into an initialized simulator that assumes the cache is empty at the beginning of the sample. The resulting simulation speeds will be faster by a factor determined by the number of sampled memory references relative to the total number of workload memory references.

In a trap-driven simulator, time sampling can be implemented by simply activating and deactivating the enforcement of access constraints. By monitoring the number of clock interrupts or the number of memory references (through the `tw_get_counts()` call) made by a workload during its execution, the simulator can turn access constraints on and off at pre-specified intervals. Before turning access constraints on after some period of non-sampling, the data structure representing a simulated cache can be flushed so that it will be empty at the beginning of a sample, as is done with trace-driven simulation. During periods when access constraints are not enforced (and cache misses are not counted), workload execution will proceed at full host hardware speeds. Thus, trap-driven simulation is able to benefit from time sampling in the same way that trace-driven simulation can.

4.2.3 Multi-configuration Simulation

In this section, we briefly describe a very basic stack algorithm for multi-configuration simulation based on the original ideas of Mattson et al [Mattson70]. We will then discuss a transformation of this algorithm that is suitable for implementation in a trap-driven simulator. Finally, using a simple argument, we show that many, more complex, multi-configuration algorithms can be implemented in a trap-driven simulator.

A Basic Stack Algorithm

The technique of stack analysis relies on the property of *inclusion* that is exhibited by certain classes of caches with certain replacement policies. Mattson et al. show, for example, that an *n*-

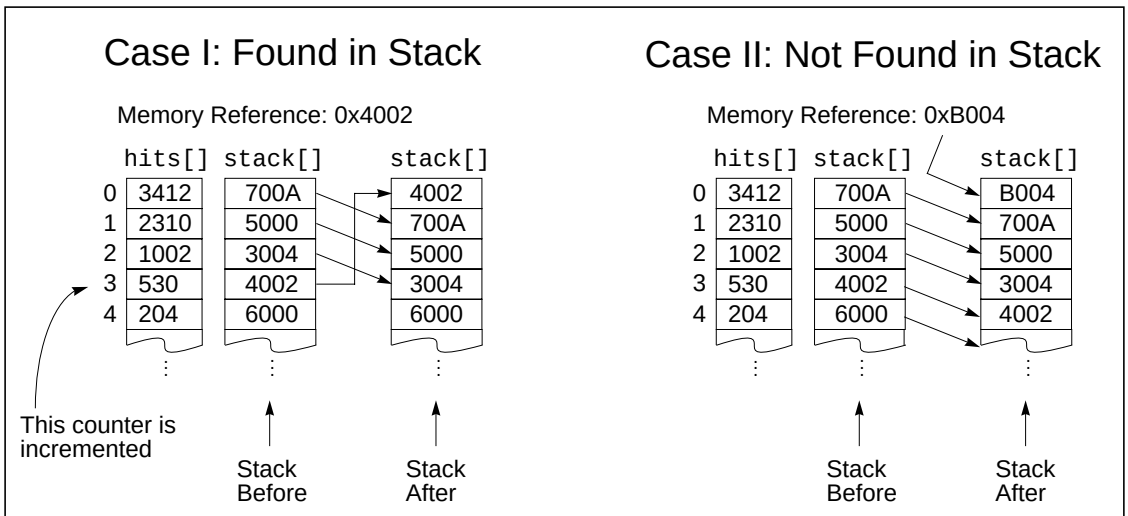


Figure 4.15 Data Structures for Stack Simulation

In Case I, the address is found at stack depth 3, so the hits[3] counter is incremented, and the entry at this depth is pulled to the top of the stack. In Case II, the address is not in the stack, so it is pushed onto the top, and no counter is incremented.

entry, fully-associative cache that implements an LRU replacement policy includes all of the contents of a similar cache with only (*n*-1) entries.

When inclusion holds, a range of different-sized, fully-associative caches can be represented as a stack as shown in Figure 4.15. The figure shows that a one-entry cache holds the memory line starting at 0x700A, a two-entry cache holds the lines starting at 0x700A and 0x5000, and so on. Trace addresses are processed, one at a time, by searching the stack. Either the address is found in the stack (Case I) at some *stack depth*, or it is not found (Case II). In the first case, the entry is pulled from the middle of the stack, and pushed onto the top to become the most-recently-used entry; other entries are shifted down until the vacant slot in the middle of the stack is filled. In the second case, the missing address is pushed onto the top of the stack and all other entries are shifted down.

To record the performance of different cache sizes, the simulator maintains an array that counts the number of hits at each stack depth. At the end of the simulation run, the number of hits in a fully-associative cache of size *n* ($hits_n$) can be computed from this array by adding all the hit counts up to a stack depth of (*n*-1) as follows:

```
int stack[cacheSets][maxAssoc];
int hits[maxAssoc];

while (pa = trace(file)) {
   stackDepth = search(stack[pa_set][], pa);

   if (stackDepth > 0)
      adjust(stackDepth, stack[pa_set][]);

   if (stackDepth != Infinity)
      hits[depth]++;
}
```

Figure 4.16 A Trace-driven Implementation of the Stack Algorithm

By maintaining a separate stack for each cache set, caches ranging in associativity from direct-mapped to n-way set-associative can be simulated in one trace pass. The pseudo-code above shows the essentials of such a stack algorithm. Notice that a stack is only adjusted when the workload accesses memory location that is not at the top of any stack.

$$\text{hits}_n = \sum_{i=0}^{n-1} \text{hits}[i] \qquad \text{(Eqn 4.35)}$$

Further metrics, such the number of misses, the miss ratio, or the MPI in a cache of size *n* can then be computed as follows:

$$\text{misses}_n = \text{totalRefs} - \text{hits}_n \qquad \text{(Eqn 4.36)}$$

$$\text{missRatio}_n = \text{misses}_n / \text{totalRefs} \qquad \text{(Eqn 4.37)}$$

$$\text{MPI}_n = \text{misses}_n / \text{instExec} \qquad \text{(Eqn 4.38)}$$

A Trace-driven Implementation of the Stack Algorithm

The essential parts of the trace-driven stack algorithm of set-associative caches is shown in Figure 4.16. This algorithm implements one stack for each cache set so that a range of direct-mapped to n-way set-associativities can be simulated in a single simulation pass.

The algorithm reads addresses from an input trace one at a time, indexes a cache set, and then searches the corresponding stack with the `search()` routine. The search returns the stack depth

```
int stack[cacheSets][maxAssoc];
int hits[maxAssoc], coldMisses;

tw_trap(pa, va, type) {
   stackDepth = search(stack[pa_set][], pa);

   adjust(stackDepth, stack[pa_set][]);

   if (stackDepth == Infinity)
      coldMisses++;
   else
      hits[stackDepth]++;
}
```

Figure 4.17 A Trap-driven Implementation of the Stack Algorithm

This figure shows a trap-driven version of an algorithm equivalent in function to that of Figure 4.16. The stack is *always* adjusted in this version because traps only occur when the workload references memory not at the top of a stack.

of the address, or `Infinity` if the address is not located in the stack. If the address is not at the top of the stack, then the stack is adjusted with `adjust()` as shown in the two cases of Figure 4.15. If the stack depth is not infinity, then the appropriate hit counter is incremented.

A Trap-driven Implementation of the Stack Algorithm

Implementing the stack algorithm in a trap-driven simulator poses two problems. First, any memory reference that requires a change in state of the array of stacks must be detected. Second, the array of hit counters must be maintained for each stack and each stack depth. The trap-driven algorithm shown in Figure 4.17 shows how these problems are solved.

The first problem can be solved by noticing that in the trace-driven implementation, the array of stacks is only updated when the stack depth is greater than zero. In other words, any reference to the top of a stack in a given set does not change the state of the stack data structure. By restricting access to only the most-recently-used line in each set, a trap-driven simulator can force traps to occur whenever stack state needs to be updated. The access constraints that accomplish this are very similar to those used for the simulation of an LRU replacement policy (Eqn 4.12 - Eqn 4.14).

The second problem, updating the array of hit counts for each stack depth, is easily solved for stack depths other than 0. The trap handlers can simply increment the appropriate hit counter before returning to the running workload. It is not possible, however, to continually update the hit counters that correspond to stack depth zero without trapping after each memory reference. However, if we know the total number of references made by the workload, as well as the number of misses that occur at a stack depth of infinity (cold-start misses), then from the following equality:

$$\text{hits}[0] = \text{totalRefs} - \text{coldMisses} - \sum_{i=1}^{\text{maxAssoc}} \text{hits}[i] \qquad \text{(Eqn 4.39)}$$

we can compute the total number of hits to stack depth 0 in all stacks. Once this value is known, performance metrics can be computed for caches of varying associativities (or set sizes) and can be computed using the same equations as given in Eqn 4.35 - Eqn 4.38.

Other Multi-configuration Algorithms

As noted in Chapter 2, Mattson et al. and others have show that the simple stack algorithm can be extended to a variety of cache structures with different sizes, line sizes, associativities, and write policies. Can these algorithms be adapted to trap-driven simulation? The answer is yes, but an exhaustive transformation of each of the numerous published multi-configuration algorithms would not be particularly enlightening. Instead, we provide a simple, intuitive argument for why transformations are possible.

All multi-configuration algorithms rely, in some way, on the similarity of state among a collection of cache structures. This similarity is usually described in terms of an *inclusion relationship* among all of the cache structures simultaneously under consideration. There is typically some *base configuration* that is smaller, less associative, or more constrained in some other way, than any of the other caches considered, and the state of this base configuration is included in all of the other caches that are simulated. If a trap-driven simulator defines access constraints for the base configuration, then a trap will occur for all references that cause a change in state in *any* of the multiple configurations considered. It is possible, then, to write a trap handler that implements the desired multi-configuration simulation algorithm.

In support of the above argument, we cite the work of Wang and Baer [Wang90] who give single-pass, multi-configuration simulation algorithms for a variety of cache sizes, associativities,

and performance metrics (misses and write-backs). All of their algorithms are all driven by filtered traces that are obtained by removing all trace addresses that hit in a small direct-mapped cache (a base configuration), or that are not first-time writes to a clean line. A trap-driven simulator can provide a series of traps that correspond exactly to the addresses in Wang's form of filtered trace by implementing access constraints similar to those of Eqn 4.18 - Eqn 4.20. The existence of Wang's algorithms and the ability of a trap-driven simulator to produce a reduced sequence of addresses on-the-fly, shows that a range of multi-configuration trap-driven simulations are possible.

4.2.4 Real-time Filtered Trace Generation

Each of the methods described in the previous three sections is based on some form of *filtering* of a complete sequence of memory references. Set sampling filters out addresses not belonging to the sampled sets, time sampling filters addresses not belonging to sampled time windows, and multi-configuration simulation uses a base cache configuration to filter the references presented to several other, less-constrained configurations. We showed that the trap-driven implementation each of these methods is possible by using access constraints to define different types of memory-reference filters. This observation leads to a different way of viewing a trap-driven simulator: It is, in essence, *an address generator that can produce a filtered stream of memory references, on-the-fly, in a flexible manner.* A filtered address stream obtained from a trap-driven simulator can be used in much the same way as any other filtered address trace can.

4.3 Flexibility and Tapeworm II

We have discussed in detail the forms of memory simulation that can, in principle, be implemented by trap-driven simulation. Most, but not all of these algorithms have been implemented in Tapeworm II.

Figure 4.18 shows two dialog boxes for the X interface to Tapeworm that illustrate some of the flexibility in its simulation parameters. As the figure shows, Tapeworm can simulate I-caches and TLBs with a range of sizes, line sizes, associativities, replacement policies, and indexing policies. Not shown in the figure, but also supported by Tapeworm is the simulation of multi-level cache hierarchies. Jacob added the ability to simulate hybrid TLBs to an earlier version of Tapeworm

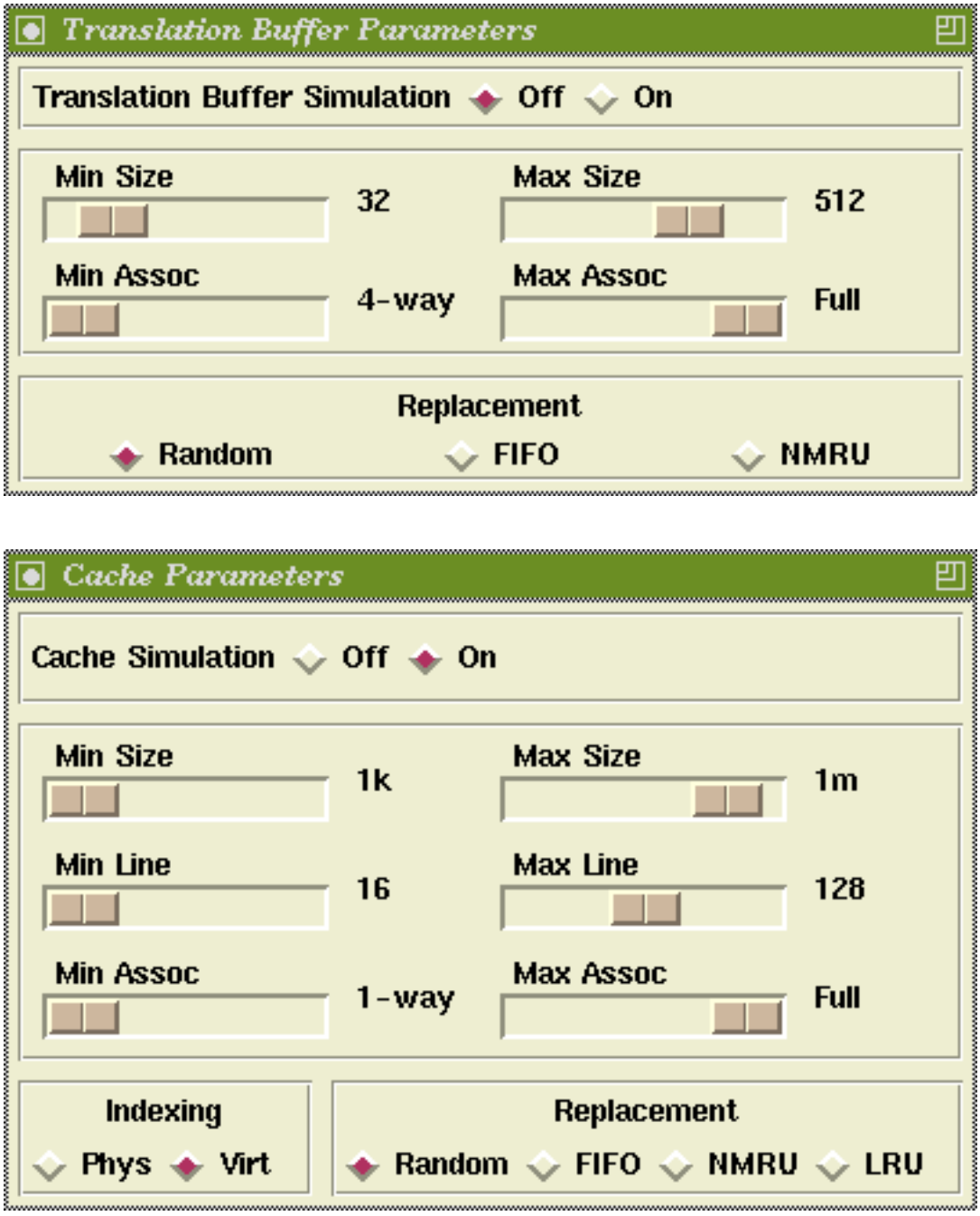


Figure 4.18 Some Tapeworm Cache and TLB Simulation Parameters

This figure shows two dialog boxes from the Tapeworm X interface, one for setting TLB parameters, the other for I-cache parameters. TLB size can range from 16 to 1K entries, and associativity can be from 4-way to fully-associative. I-cache size can range from 1K to 1M, line size from 16 bytes to 1K bytes, and associativity from direct-mapped to fully-associative.

[Jacob95]. Tapeworm can report simulation results in terms of all of the metrics discussed in Section 4.1.4. Finally, Tapeworm supports address filtering through set sampling.

The primary omission of functionality is D-cache simulation,[5] which was not possible due to insufficient support from the `tw_set_access()` primitive in the Tapeworm II prototype. The reasons behind this will be discussed in Chapter 5 on Portability. Because D-cache simulation was not implemented in Tapeworm, write policies could also not be implemented. Tapeworm does not implement time sampling, primarily because set sampling, a functionally equivalent method, has been shown to be generally superior to time sampling [Kessler91]. Earlier versions of Tapeworm implemented multi-configuration simulation algorithms for TLB simulation, but were later abandoned because of the high speed of single-configuration simulations made the more complex and slower multi-configuration algorithms less attractive than they normally are in a trace-driven environment.

4.4 Flexibility Summary

The goal of this chapter has been to develop a better understanding of what can and cannot be modeled by a trap-driven simulator. Our primary tool for reasoning about flexibility has been the concept of *access constraints*, which are a precise definition of exactly those memory references that a running workload can make without causing a change in simulated cache state. A trap-driven simulator is, in essence, a program that maintains a set of access constraints throughout the run of a workload.

Using access-constraint definitions, we have shown that a broad range of cache structures and policies of operation can be implemented with the small set of primitive routines defined in Table 3.5 of Chapter 3. We showed that the simple operations of series and parallel composition can combine these basic cache structures into trap-driven simulations of arbitrarily complex memory systems. In a similar manner, we showed that the basic event counts available at the end of a trap-driven simulation can be combined to compute all of the common memory-performance metrics. Access constraints essentially define a filter for memory references. This observation is

5. A collection of case studies performed by our group [Nagle93, Nagle94, Uhlig94b, Uhlig95] has shown that changes in operating-system structure tend to have a greater impact on TLB and I-cache performance. Tapeworm's inability to consider D-caches was therefore not a major hinderance to these studies.

the basis of trap-driven adaptations of set sampling, time sampling and multi-configuration simulation.

The access-constraint concept has also proven useful for exposing inherent limitations in trap-driven simulation. In particular, hardware structures that continually change state in parallel with program execution present difficulties to a trap-driven simulator. Examples of such structures include write buffers, prefetching units, and non-blocking caches. Although some approximation of the performance of such hardware structures may be possible, trap-driven simulation cannot completely and accurately account for their complete behavior. It should be noted, however, that when very high levels of simulation accuracy are required, trace-driven simulation may also have difficulty in these respects. For example, new processors that perform speculative execution or that execute instructions out of order, require a more detailed processor simulator, and cannot be simulated with simple static traces.

In summary, although trap-driven simulation does not exhibit the full flexibility of traditional trace-driven simulation, it comes very close. In future chapters we will see that the high speed of trap-driven simulations, in addition to its ability to easily consider multi-task and operating-system effects, more than compensates for small limitations in simulation range.

Chapter 5

Portability

Trace-driven simulation methods have been implemented on a broad range of host computer systems. Some form of trace collection, the hardware-dependent component of trace-driven simulation, exists for nearly every modern instruction-set architecture. This high degree of *portability* has been an important factor in the acceptance of trace-driven simulation methods.

If trap-driven simulation is to become widely accepted, it must exhibit a similar level of portability. Unfortunately, there are only a few trap-driven simulators presently in existence, suggesting that the method is difficult to adapt to a range of machines. Trap-driven simulation is, on the other hand, a relatively new method, so a lack of implementations does not imply that the method is inherently non-portable. The purpose of this chapter is to develop a clearer picture of trap-driven simulation portability, with the ultimate goal of answering the following questions:

- How and to what extent can trap-driven simulation be implemented on existing, unmodified computer systems?
- How can next-generation systems be modified to better support trap-driven simulation and how expensive would such modifications be?

We examine portability in three steps, beginning with a description of several general methods for implementing trap-driven simulation primitives on unmodified hardware. We will see that trap-driven TLB and I-cache simulation can be performed on most existing machines, but sometimes D-cache simulation is not possible. Next, we examine our specific implementations of trap-driven simulation primitives in Tapeworm and Tapeworm II, noting the various difficulties and problems that we encountered. Finally, armed with this experience, we suggest inexpensive hardware modifications that could substantially improve the portability and speed of a trap-driven simulator. This final section serves as a guide to architects who wish to provide support for trap-driven simulation in future computer systems.

Method	References	Required Hardware Support	Simulation Type
TLB Miss Redirection	[Nagle93] [Uhlig94b] [Talluri94]	Software-managed TLB	TLB
Page Table Shadowing	[Lee94] [Uhlig94b]	Standard memory-management hardware	TLB
Instruction Shadowing	—	Kernel trap for breakpoint instructions	I-cache
Instruction Recoding	—	Kernel trap for un-implemented instructions	I-cache
Tagged Memory Trap Redirection	—	Memory tag bits that support run-time type checking, debugging, or synchronization.	D-cache I-cache
Memory Parity Recoding	[Reinhardt93] [Uhlig94b]	Memory with parity or error-correcting code (ECC) bits with a diagnostic mode to modify these bits.	D-cache I-cache

Table 5.1 Methods for Implementing Access-control Primitives

This table summarizes some forms of host hardware support that are useful for implementing the `tw_set_access()` primitive. Many of these are common features of existing architectures (see Table 5.2). References are given for known existing implementations of a given method in an actual trap-driven simulator. *Simulation Type* specifies the most suitable application of a given method.

5.1 Porting to Existing Hardware

An important underlying assumption of the trap-driven algorithms presented in the previous chapter is that the host hardware provides full support for the primitive routines defined in Table 3.5. Ideally, we would like to implement these primitives on unmodified, existing hardware to keep implementation costs to a minimum. The following sections describe several different methods for achieving this goal.

5.1.1 Implementing Access Control

The memory-access primitives, `tw_set_access()` and `tw_get_access()`, are the basis for the realization of access constraints equations, and are thus key to any trap-driven simulation algorithm. Table 5.1 summarizes several methods for implementing these primitives on existing hardware, each of which is based in some way on the unconventional use of certain privileged machine operations. We describe each in greater detail below.

TLB Miss Redirection

In a machine with software-managed TLBs, access to a page-sized region of memory can be controlled by inserting or removing a mapping to the region in the TLB. Because TLB misses are handled in software on such a machine, a memory reference that violates the access rights defined by the TLB contents will cause a trap that can be forwarded to a trap-driven simulator. With this technique, it is possible that only a subset of all of the accessible host memory regions can be held in the TLB at any given point in time. In this case, the simulator can record access rights in an auxiliary data structure that can be queried after each TLB miss trap. Only true violations are forwarded to the simulator. The original Tapeworm TLB simulator used this method to implement trap-driven TLB simulation [Nagle93; Uhlig94b], as does the TLB simulator by Talluri [Talluri94].

A disadvantage of this method is that if the host TLB is small, then there will be frequent "false" traps that must be checked as described above. Another limitation is that access can only be granted on regions of memory that are the size of a page, limiting simulations to TLB structures. Additionally, host machines that handle TLB misses in hardware are unable to use this method because they do not trap to the OS kernel after each TLB miss.

Page Table Shadowing

Some of the limitations of the method described above can be avoided by a related method that makes a *shadow* copy of the actual page tables of the host VM system. The shadow copy holds the access rights defined by the trap-driven simulator, which are always a subset of the actual access rights defined by the host VM system. By inserting this page-table shadow in place of the actual page tables, the memory-management hardware will trap on a more constrained set of access rights. If a resulting trap is due to a true page fault or access violation, it is forwarded to the host VM system to handle in its normal way. Otherwise, it is passed to the trap-driven simulator. This method was implemented by Lee in his port of the original Tapeworm simulator to a 486-based Gateway PC [Lee94] and is also used in Tapeworm II.

The main advantage of page-table shadowing is that it works with both hardware- and software-managed TLBs. A further advantage is that no checking is required for false traps as with the redirection of software-managed TLB miss traps. A disadvantage is that a more complex set of

data structures (the page-table shadows) must be maintained, and access control is still limited to page-sized regions of host memory.

Instruction Shadowing

Another form of shadowing can be used to control access on smaller regions of memory consisting of instructions. This method works by inserting a shadow page full of breakpoint instructions in place of each text page that is in the domain of the trap-driven simulator. A workload that accesses (executes) memory locations on the shadow page will be suspended by an instruction-breakpoint trap to the kernel. Read-only access can be granted to the shadow-page memory location by copying a valid instruction from the original text page in place of the breakpoint instruction. Similarly, access can later be removed by copying a new breakpoint in place of the valid instruction.

Instruction shadowing can be implemented on any machine with a breakpoint instruction (i.e., most modern microprocessors), but the method is easier to apply to RISC machines, which have regular 32-bit instruction codings. The method allows read-only access to be controlled on memory regions as small as a single word, but it can only be applied to instruction memory locations, limiting its applicability to I-cache simulation. Because each shadowed instruction page uses twice as much physical memory as it normally would, another disadvantage of instruction shadowing is increased memory usage.

Instruction Recoding

Most instruction sets have a number of unused instruction codes that cause an undefined-instruction trap if encountered by the processor. If the number of unused instruction opcodes is less than the number of legal ones, as in the MIPS-I instruction set [Kane92], then it is possible to recode each legal opcode, in a one-to-one manner, to a corresponding and unique un-implemented opcode. Read-only access to instruction memory locations can then be granted or removed by converting an instruction to and from its legal and un-implemented counterpart.

The advantage of this method over instruction shadowing is that the recoding of instructions can be done *in-place*, so that no extra memory for shadow pages is required. With the exception of these memory savings, instruction recoding shares the same advantages and disadvantages of

instruction shadowing. Namely, it allows access to be controlled on small memory regions, but it it's applicability is limited to I-cache simulation.

Tagged Memory Trap Redirection

Some machines associate tag bits with each memory location so that they can perform an access test on references to small regions of memory. Tagged memories have been proposed and implemented for a variety of purposes, including distributed shared memory, debugging, synchronization, and run-time type checking [Reinhardt94; Alverson90; Taylor86].

A machine that supports tagged-memory checks provides ideal support for a trap-driven simulator because tag states can often be set in such a way to cause kernel traps. Unfortunately, few machines implement tagged memories, so applicability of this method is limited.

Memory Parity Recoding

Although most machines do not provide memory tags, many *do* add extra parity or error-correcting-code (ECC) bits to each memory location to detect memory errors. These machines typically support diagnostic test modes that enable the parity or ECC bits to be recoded to an invalid state and then stored back to memory. Subsequent accesses to memory locations with recoded parity or ECC bits cause a kernel trap that can be forwarded to a trap-driven simulator. As with instruction recoding, it may be possible to change access states in-place, avoiding the need for shadow copies of original data.

The recoding of memory parity bits can be performed on small regions of both instruction and data memory, making the method suitable for both trap-driven I-cache and D-cache simulation. Unfortunately, implementing this method can be difficult because it requires a knowledge of the diagnostic test modes of a system, information that is often not publicly available. Another limitation is that it is usually only possible to implement two access states, (no-access and read-only) or (no-access and read-write).[1] This method was first proposed and implementing in the WWT simulator on a SPARC-based CM-5 [Reinhardt93], and is also used in Tapeworm II.

1. If only two access states can be implemented, most forms of trap-driven simulation are still possible. Three states are only required when simulating a D-cache where write-back counts are desired.

5.1.2 Implementing Traps

Methods for controlling access cause host hardware to trap to the OS kernel, invoking some code inserted at a trap-vector address. Normally, the code at the vector address contains a handler for some normal system function, like servicing a page fault, a debug trap, or a memory-parity error. This code vector must be replaced by new code that invokes the `tw_trap()` call. This replacement must be done in a way that satisfies two requirements. First, normal system functions must still be supported. For example, an actual page fault or ECC error must still be serviced. Second, the three arguments `pa`, `va` and `type` must be computed so that they can be passed in the invocation of `tw_trap()`.

The first requirement can be satisfied by checking for conditions that identify the cause of the trap so that it can be properly directed to either the host operating system or to the trap-driven simulator via a `tw_trap()` call. How this is implemented depends on the trapping mechanism, but it is typically not difficult to accomplish. For example, if page-table shadowing is used, a real page fault can be detected by consulting the host VM page tables. Similarly, if a specific and consistent method for recoding ECC bits is used, then real ECC errors can be detected with high probability.

The second requirement can usually be satisfied by reading status registers provided by the host, which will typically report the virtual or physical address responsible for a trap, along with the address of the faulting instruction. If only a virtual or physical address is reported, but the opposite form is required, then the page tables of the host can be used to translate or reverse translate the address. The type of memory reference (`dataLoad`, `dataStore`, or `instrFetch`), if not directly reported by the host hardware, can be deduced by examining the faulting instruction.

5.1.3 Implementing Event Counting

Many new microprocessors support performance counters that can be used to determine the number of loads, stores and instruction fetches that are required to implement the `tw_get_counts()` primitive. To support the full semantics of this call, counting should only be performed while executing tasks that are in the trap-driven simulator’s domain. This can be implemented by enabling or disabling counting, as appropriate, during task switches. For

processors that do not provide performance counters it may be necessary to use brute-force methods, such as counting memory references with an external hardware logic analyzer.

5.1.4 Summary of General Implementation Methods

We have presented several methods for implementing trap-driven simulator primitives on existing hardware. No single method is superior in all respects, with the selection of the best method depending on the type of memory simulation to be performed and on the particular support provided by the host hardware. Fortunately, many existing machines provide at least some form of support for a trap-driven simulator (see Table 5.2).

TLB-miss redirection and *page-table shadowing* are most effective for TLB simulation because they allow access to be changed quickly on large memory regions and because any microprocessor with an MMU provides the necessary support. Wide support for instruction breakpoints makes *instruction shadowing* the best choice for I-cache simulation, with *instruction recoding* being an attractive optimization that saves memory, provided that there are enough un-implemented instruction codings available to permit its implementation. While not as widely supported, *tagged-memory-trap redirection* and *memory-parity recoding* provide the advantage of supporting both I-cache and D-cache simulation, but they may not be able to implement all three access states.

5.2 Portability and Tapeworm II

To demonstrate their feasibility, several of the methods for implementing trap-driven simulator primitives on existing hardware have been tested in the Tapeworm and Tapeworm II prototypes. We now describe and comment on the various difficulties encountered during the implementation of these prototype primitives.

5.2.1 Implementation of Access Control

The Tapeworm and Tapeworm II prototypes experimented with three of the methods for implementing the `tw_set_access()` primitive: TLB-miss redirection, page-table shadowing, and memory-parity recoding.

Privileged Operation	MIPS R3000	MIPS R4000	SPARC	DEC Alpha	Tera	Intel i486	Intel Pentium	AMD 29050	HP PA-RISC	Power PC
Software-managed TLB	Yes	Yes	Yes	Yes	—	No	No	Yes	Yes	—
Memory Management Hardware	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Instruction Breakpoint	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Invalid Instruction Trap	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Memory Tag Bits	No	No	No	No	Yes	No	No	No	No	No
Memory Parity or ECC Traps	Yes	Yes	Yes	Yes	Yes	—	Yes	—	—	—
Performance Counters	No	No	—	Yes	—	No	Yes	No	—	Yes

Table 5.2 Privileged Operations on Modern Microprocessors

The entries in this table were taken from a variety of sources including data books, text books and Microprocessor Report [MReport92; MReport93; MReport94]. A given entry may not be true for every implementation of a given processor. Some features, such as memory-parity-error traps are actually system-implementation dependent. For these features, an affirmative entry means that we found at least one system with the given microprocessor that implements the feature. A blank entry means that insufficient data was available.

TLB-miss redirection in the first version of Tapeworm on a DECstation 3100 proved successful, although maximum speed when simulating large TLBs was limited due to a large number of false traps. This limitation was avoided in the port to the 486-based Gateway, and in Tapeworm II on a DECstation 5000/200, both of which successfully implement page-table shadowing.

The implementation of access control with ECC bits in Tapeworm II was far more difficult. Recoding ECC bits on a DECstation 5000/200 proved to be a time-consuming operation, requiring a long and convoluted sequence of control instructions to the memory-controller ASIC that implements its ECC logic. To ensure that a trap would occur on a future reference, it was also necessary to flush the recoded memory from the processor cache,[2] an operation that also required a complex and time-consuming sequence of special instructions.

A more serious problem was caused by writes to memory locations marked non-accessible by recoded ECC bits. These writes resulted in new (valid) ECC bits being recomputed and stored to memory without checking the old (invalid) ECC bit values. This behavior effectively changed a memory location's access state from no-access to full-access outside of Tapeworm's control. Fortunately, the method could still be used on read-only text pages, but this limited simulations to I-caches with two access states, no-access and read-only. As noted by Reinhardt, it is possible to avoid this problem on a machine with an *allocate-on-write* policy[3] by flushing memory locations from the cache when setting their state to no-access [Reinhardt93]. In such a system, a write to the uncached location causes the data to first be read (allocated) from main memory into the cache before the write completes. The ECC bits of this allocate operation will be checked in the same way as any other read to main memory, thus forcing a trap to occur. Although this solution enables D-cache simulation, it still only supports two access states: no-access and full-access.

Other problems with ECC caused difficulties when porting Tapeworm II to other machines. For example, our port of Tapeworm from a DECstation 5000/200 to a DECstation 5000/240 was hindered due to differences between the way that DMA is implemented on the two machines. Another minor limitation is that ECC bits are checked on 4-word cache-line refills, effectively limiting the simulation of cache line sizes to multiples of 4 words.

2. On the DECstation 5000/200, ECC is only checked on cache-line refills after a cache miss.

3. The DECstation 5000/200 uses a write-though policy with no allocate-on-write.

5.2.2 Implementation of Traps

Implementation of the `tw_trap()` call was non-eventful for the TLB miss redirection and page-table shadowing methods. All the necessary information, `pa`, `va` and `type`, was easily accessible and available in hardware registers, enabling efficient trap handlers to be written.

An efficient implementation `tw_trap()` was much more difficult in the case of the ECC recoding method. First, this trap was routed to a generic exception vector and had to be identified from among many other sources of traps, interrupts and exceptions. A clumsy interface to the memory-control ASIC required a dozen load, shift, add and mask instructions to piece together the memory address of an ECC error (i.e., the `pa` value of the `tw_trap()` call). Once this value was obtained, the corresponding virtual address had to be found by searching an inverted page table. These code sequences, along with the complex sequence described in the previous section for recoding ECC bits, required several working registers. This, in turn, required the saving and restoring of several workload registers before and after each trap, further increasing the trap-handling time.

In retrospect, given the problems implementing `tw_set_access()` and `tw_trap()` with ECC bits, it would have been more sensible to use instruction shadowing or instruction recoding in Tapeworm II. Implementation would have been eased because instruction breakpoints are far easier to set and clear, and the breakpoint traps report the faulting instruction address in an easily accessible hardware register. The resulting primitives would have supported the same two access states (no-access and read-only), and would likely have resulted in a faster, less problematic implementation. Unfortunately, these methods were devised after work on this dissertation began, but work is currently underway to implement and evaluate them.

5.2.3 Implementation of Event Counts

Tapeworm was implemented on machines that do not support performance counters. We were forced to use the brute-force method of obtaining memory-reference counts and instruction counts using a logic analyzer. Special marker instructions were inserted in the kernel to mark address-space switches. This was required to enable and disable counting so that only instructions and memory references made by tasks in the Tapeworm domain would be counted.

Primitive	Hardware Support	Relative Cost
`tw_set_access()`	MMU Modifications to support small, variable-sized pages with sub-page access bits	High
	Memory tag bits for coding access states	Medium
	Trap on cache miss or first write to a clean line with per-page enable bit	Low
`tw_trap()`	Dedicated trap vector	Low
	Extra working registers for trap handler	Medium
	Easy access to `va`, `pa` and `type`	Low
`tw_get_counts()`	Instruction and memory-reference counters with per-page enable bit	Medium

Table 5.3 Suggested Hardware Support for Trap-driven Primitives

This table summarizes some forms of inexpensive hardware support that would ease the implementation of the trap-driven primitives. The relative cost of these methods is based on the approximate degree to which they could affect the design of the host machine.

5.2.4 Summary of Tapeworm II Portability

Despite several implementation problems, particularly with ECC recoding, we were able to implement enough of the Tapeworm primitive operations to construct a usable trap-driven simulator prototype. Although limited to TLBs and I-caches, this prototype has enabled us to evaluate the feasibility of trap-driven simulation without resorting to hardware modifications. Despite being hindered by these limitations, we will see in future chapters that the speed and accuracy of this prototype provides significant improvements over trace-driven simulation. These promising results in a prototype implementation justify the investigation of special hardware support for trap-driven simulation that could provide an even greater boost in memory-simulation speeds.

5.3 Trap-driven Simulation on Next-generation Hardware

During the implementation of the Tapeworm primitives, it became clear that certain minor hardware modifications could substantially improve their efficiency. Table 5.3 lists some suggestions for designers who wish to provide more support for trap-driven simulation in their designs of future machines. We now describe these hardware modifications in greater detail.

5.3.1 Support for Access Control

We propose three different hardware modifications that would each help to implement the `tw_set_access()` primitive. These modifications are roughly ordered according to decreasing cost, with a corresponding decrease in flexibility and speed. The first method, which modifies MMU hardware, provides the most flexibility, but is also likely to be the most costly. The second method, based on tagging memory with access bits, is somewhat less flexible but would probably have less of an impact on processor cycle time. The final method, involving a simple modification to a system's cache controllers, would cost very little, but would also be the least flexible and could provide worse performance.

Memory Management Unit Modifications

The access controls required by a trap-driven simulator are very similar to those already supported by standard MMU hardware. This suggests that MMUs are a natural starting place for hardware modifications that support access control on smaller regions of memory. Two modifications to standard MMUs, used in combination, would enable them to provide the desired support. The first modification is to support small, variable-sized pages, and the second is to implement sub-page protection.

Many recent microprocessors support variable-size pages by using a variable number of bits from the virtual address in their TLB lookups (the larger the page, the smaller the virtual page number and the corresponding TLB tag). For example, the MIPS R4000-series support 7 page sizes ranging from 4K to 16MB pages [Kane92]. An extension of this concept is to enable *smaller* variable-sized pages by supporting slightly larger tags. For example, adding 8 bits to the maximum TLB tag size would enable the minimum page size to be reduced from 4K to 16 bytes. Although this method enables access control on smaller regions of memory, it would dramatically increase TLB miss rates. For example, a 64-entry TLB holding mappings for 16-byte regions would cover a total of only 1 KB of memory, an effective capacity less than even a very small cache.

A different extension to MMU hardware is *sub-page protection*, which enables memory locations on the same page to have different levels of access. This is supported by extending page-table entries to contain extra access bits for each sub-page region. For example, a 4-KB page could be divided into 16 sub-page regions of 256 bytes apiece. Access to each region could then be guarded by 2 access bits that encode the three access states. The number of sub-page regions is

limited by the number of access bits that can be held by the TLB. In the example above, full support of three access states would require 16 x 2 = 32 bits, using a simple coding scheme.[4] Support similar to this is provided in actual machines, such as the RS/6000, which provides a lock bit for each 128-byte region on a page [Schonias94], and the R4000, which maps 8K virtual regions in one TLB entry, but permits the two halves of the 8K region to have different access rights (and physical page frames) [Kane92]. Although sub-page protection improves the granularity of protection, 256-byte or 128-byte regions are still too large for cache simulation, where line sizes are typically in the range of 16 to 64 bytes.

An MMU that combines the above two methods could overcome their individual limitations. If, for example, a machine with variable-sized pages set the page size to 512 bytes, and divided it into 16 sub-page regions, then a 64-entry TLB could map a total 32 KB of memory and access could be controlled on 32-byte regions. A strong point of this method is that it flexibly supports a full range of access sizes, ranging from cache-line sizes to larger, traditional-sized pages.

The implementation costs of such support are difficult to assess because of their affect on TLBs, which are typically highly-associative structures that are sensitive to increases in the size of their tags and content bits. We note, however, that many processors now support 64-bit virtual addressing and steadily increasing physical memory sizes, both of which require similar increases in TLB tag and content sizes.

Memory Access Tags

Access control can be supported simply and directly by tagging memory locations with bits that code the three possible access states (no-access, read-access, and full-access). Such support would essentially be the same as that provided by tagged memory systems described in previous sections [Reinhardt94; Alverson90; Taylor86]. The host hardware should support instructions that enable the access-state tags to be changed quickly, and each memory access should be checked against these tags. This could be implemented by minor modifications to cache-control logic. The access bits, for example, would be checked on cache refills, and then concatenated to cache tags so that they could be checked as part of the usual cache hit-detection logic on future references.

4. More compact codings are possible, at the expense of increased complexity.

Such support can increase the cost of a design in two ways. First, extra memory is required to store the access tags (both in memory and in caches). Second, extra logic is required to perform the access checks. The amount of extra memory required is small compared to the extra bits already stored by many machines to support parity or ECC bits. For example, a parity bit is often assigned to each byte or word of memory, and 7 ECC bits are required for each 32 bits of data to implement a single-error correcting, double-error detecting code. These ratios of data to redundant bits are roughly 4:1 to 32:1. On the other hand, if two access bits are assigned to 32-byte blocks of memory, then the ratio of data to extra bits is only 128:1. Schoinas points out that if a system already supports ECC, it is possible to avoid extra memory costs altogether by coding the access states with unused ECC values [Schoinas94].

The amount of extra logic required to implement memory-access tags would not substantially complicate cache-control logic because only two extra access-state bits must be examined. This is unlikely to add to the critical path of a processor that must already check page-level access (through a TLB lookup) and perform cache-tag comparisons on 20- to 30-bit values. This method is somewhat less flexible than the MMU modifications described in the previous section because the regions tagged by access bits are of a fixed (small) sized.

Trap on Cache Miss

The *TLB miss redirection method*, described in Section 5.1.1, relies on host hardware that traps to the kernel whenever a TLB miss occurs. Unfortunately, this same method cannot be applied to cache simulation because host cache misses are typically serviced entirely by hardware. Cache-control logic could, however, be modified to support a trap-on-cache-miss execution mode. When the processor is in this mode, a *cache miss*, or the *first write* to a clean cache line would cause a trap to a software cache-miss handler. This form of support is not quite the same as actual fine-grain access control, but equivalent functionality can be achieved, provided that the host cache uses a write-back, allocate-on-write policy, and supports instructions to manipulate the state of the cache. Given such a machine, the three access states can be synthesized as follows:

- Flushing a memory location from the cache corresponds to setting its access state to *no-access* because a future read or write to the memory location will cause a cache miss and cache-line refill, resulting in a kernel trap.

- Loading a memory location into the cache and clearing the dirty bit on the line corresponds to setting the access state to *read-only* because a future write to the line will cause a *first-write* trap.

- Loading a memory location into the cache and setting the dirty bit on the line corresponds to setting the access state to *full-access* because both reads and writes to the cache line will proceed uninterrupted, as long as the line is not displaced from the cache.

As with TLB miss redirection, a trap does not necessarily imply that access was violated; traps can occur when a line that was marked accessible is displaced from the cache and then later re-accessed. A data structure that records the access state of each memory location in the simulator's domain (much like the page-table shadow) could be used to check each trap for actual access violations, and false access violations would be handled by simply reloading the desired memory location back into the cache.

To avoid interfering with access to pages outside the domain of the simulator, it should be possible to disable the trap-on-cache-miss execution mode on a per-page basis in much the same way that many processors support a per-page cacheable bit. In a machine that supports multi-level caches, it would be best to implement the trap-on-cache-miss mode in the largest cache, so that false access violations could be kept to a minimum.

This method is inexpensive to implement because it uses the existing resources of a standard write-back cache (its tags and dirty bits), while implementing everything else in software. The only required changes are a minor modification to the cache-control logic, and the addition of a trap-on-cache-miss bit to the page-table entry format.

5.3.2 Support for Traps

Trap logic would be best improved by treating access violations as a common, rather than an exceptional event. Three things, in particular, would improve the performance of a `tw_trap()` call. First, the trap should invoke code at a special vector address reserved for holding the simulator trap handler. A dedicated vector removes the cost of determining if the trap is due to the simulator or a normal system event. Second, a small set of working registers reserved for the simulator trap handler would help to avoid the cost of saving and restoring workload registers.

Finally, access to information about the reference causing the trap (`pa`, `va` and `type`) should be made easy for the handler to access quickly.

A good example of ideal support is exhibited by many processors that handle TLB misses in software. The MIPS R-series of processors [Kane92], for example, support a dedicated trap vector for TLB misses, although all other traps are sent to a generic trap vector. Two working registers are always available to the TLB miss handler, so no workload registers must be saved, and the faulting address is easily accessible in a hardware register. This simple support enables a software TLB miss handler of less than a dozen instructions to execute in about 20 cycles [Nagle93]. Other machines, such as certain versions of the HP-PA and the SPARC architectures, provide similarly effective and streamlined trapping support [HP90; Huck93; Sun94]. We believe that a cleaner trap interface could reduce the total miss-handling time in Tapeworm II from 300 cycles in its current implementation, down to about 50 cycles.

5.3.3 Support for Event Counting

Many new processors already provide performance counters that report the total number of memory references and instruction executed by a workload. A minor enhancement to such counters is to support the enabling and disabling of counting on a per-page basis so that a counter is incremented only when executing from text pages that have a count bit set. Such support would make it easier to determine the exact number of references and instructions executed only by the workload pages that have been added to the Tapeworm domain via the `tw_add_page()` call.

5.4 Portability Summary

Trap-driven simulators are clearly more difficult to port to existing hardware than are trace-driven simulators. The purpose of this chapter has been to alleviate this problem by describing methods for implementing trap-driven primitives on existing hardware, and by suggesting low-cost ways to support trap-driven simulation on future machines. Several of the methods discussed for implementing the primitives on existing machines were validated in Tapeworm and Tapeworm II, and the success of these prototypes justifies the consideration of more special hardware support for trap-driven simulation in future machines.

We have shown that virtually any modern microprocessor supports trap-driven TLB and I-cache simulation, although trap-driven D-cache simulation may not always be possible. Difficulties implementing the Tapeworm prototypes usually stemmed from awkward hardware interfaces that appear to have been designed under the assumption that traps are infrequent events. Minor changes to these trap interfaces would substantially ease the implementation and improve the performance of trap-driven simulators on future machines. Simple memory-reference and instruction counters would have been very helpful, allowing us to avoid the use of a logic analyzer; Every new processor should support at least an instruction counter.

Simply stated, the hardware support that a trap-driven simulator requires most is fine-grain access control. Most of the problems with the trap-driven portability will vanish if future systems provide such support. We have suggested three different methods for implementing fine-grain access control, each with a different level of cost and flexibility, giving designers several possible starting points. The suggested changes are by no means excessively costly, and many similar forms of support have been implemented in past system and in many experimental machines.

It should be noted that many other applications would also benefit from fine-grained access control. Program debugging, garbage collection, persistent storage, and distributed shared memory could all be made faster and more efficient [Appel91; Reinhardt94; Schoinas94]. These applications, and the promise of very fast trap-driven memory simulation, suggest that architects should give more serious consideration to supporting fine-grained access control on future processors.

Chapter 6

Speed

The main motivation for our study of trap-driven simulation has been its potential for improving simulation speeds. We have shown that despite several advances, trace-driven simulation is still at least an order of magnitude slower than actual hardware, and is hindered by inherent bottlenecks that limit further substantial improvements in its speed. Trap-driven simulation speeds, on the other hand, are limited only by the frequency of changes in simulated cache state. Slowdowns in a trap-driven simulator, for example, approach zero as cache sizes increase. Although this is true in principle, it of no use in practice if trap-driven slowdowns are excessively high for all but the largest caches. Other cache parameters, such as cache associativity, line size and replacement policy, also affect the frequency of cache-state changes, as well as the time to process these changes. Early trap-driven simulators, in fact, exhibited wide variations in simulation speeds (see Chapter 3). These issues raise several questions:

- How do the speeds of actual trace-driven and trap-driven simulators compare?
- How do slowdowns of an actual trap-driven simulator vary with the parameters of a simulated cache?
- For what range of cache parameters are trap-driven simulation slowdowns acceptable?

In this chapter, we answer these questions with measurements of the Tapeworm II prototype. We begin by using the *slowdown* metric to compare the speed of Cache2000 simulations driven by Pixie traces and Tapeworm simulations driven by kernel traps. We use a simple model for simulation overheads to explain these results, and then apply the model in further experiments to explore the relationship between slowdowns and simulated cache size, associativity, line size, replacement policy, cache indexing and set sampling.

6.1 Trap-driven versus Trace-driven Simulation Speed

We compare the speed of simulators using the slowdown metric. Recall that *Slowdown* is the ratio of simulation overhead to the run time of an uninstrumented workload. Depending on the simulator, we compute slowdown as follows:

Slowdown = (Tapeworm Overhead) / (Normal Workload Run Time) (Eqn 6.1)

Slowdown = (Cache2000 Overhead) / (Normal Workload Run Time) (Eqn 6.2)

where *Overhead* is the time added to a workload run by Tapeworm or Cache2000. In the case of Cache2000 simulators, this overhead includes the time to generate addresses from a pixie-annotated workload. *Normal Workload Run Time* is for an unmodified workload running on a host. machine.

Figure 6.1 plots Tapeworm and Cache2000 slowdowns against cache size for the `mpeg_play` workload. Because the Pixie and Cache2000 combination can measure only a single-task workload, Tapeworm was configured to set traps only on memory locations in the `mpeg_play` task to enable a fair comparison.[1] For both simulators, slowdowns decrease as cache sizes increase. Cache2000 slowdowns are approximately 30 for the smallest caches and decrease to just under 25 for the largest caches, while Tapeworm slowdowns start at about 3-4 for small caches and decrease to 0 as cache size is increased. To understand this behavior, consider the following expression for the overhead of the Cache2000 simulations:

$$\text{Cache2000 Overhead} = (\text{Miss}_{\text{count}})\,(\text{Miss}_{\text{time}}) + (\text{Hit}_{\text{count}})\,(\text{Hit}_{\text{time}}) \qquad \text{(Eqn 6.3)}$$

where $\text{Miss}_{\text{count}}$ and Hit_{time} represents the number of simulated cache misses and hits, while $\text{Miss}_{\text{time}}$ and Hit_{time} are the average amount of time required to process simulated cache hits and misses, respectively. These processing times are different because a simulated cache hit requires only an address generation and search operation (about 60 cycles in Cache2000), but a simulated cache miss also requires data structures to be updated with the missing cache line (about 260 cycles in Cache2000). This explains why the Cache2000 slowdowns decrease with increasing cache size; larger caches exhibit more hits than misses, and hits require less time to process.

1. The plot also shows Tapeworm slowdowns when all workload components are monitored. The resulting slowdowns are about 2 to 2.5 times greater.

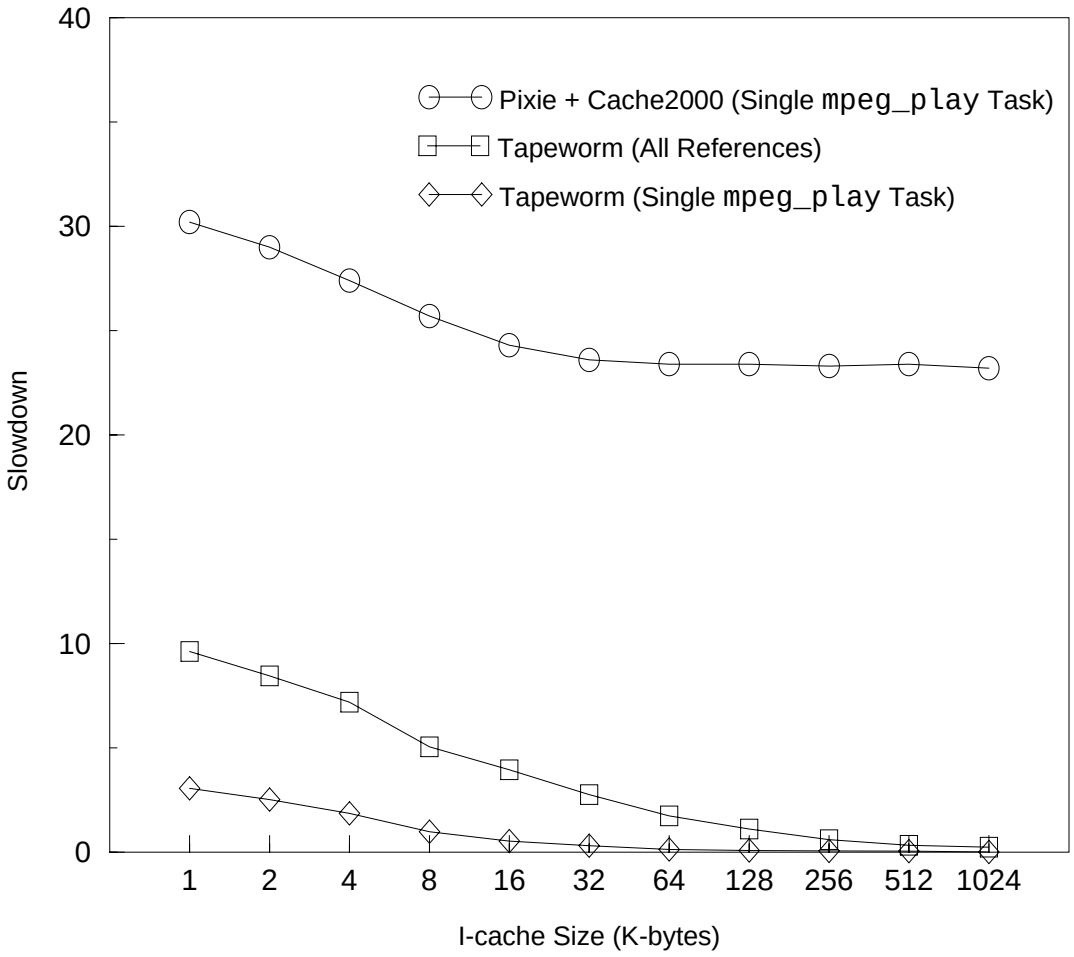


Figure 6.1 Comparison of Trace-driven and Tapeworm Slowdowns

Tapeworm slowdowns compared with slowdowns of a Cache2000 simulation driven by Pixie-generated instruction address traces. The simulation is of `mpeg_play` for different sizes of direct-mapped instruction caches with 4-word lines (4 bytes/word). Two different Tapeworm simulations are shown: one with user-only references from just the `mpeg_play` task and another with references from all workload components, including the kernel and user-level servers (BSD and X). The Pixie + Cache2000 combination can measure only a single-task workload. In all cases, slowdowns were computed relative to the total wall-clock run time for all workload components.

The Pixie + Cache2000 simulations were performed under Ultrix 4.1 on a DECstation 5000/133. The Tapeworm simulation were performed under Mach 3.0 on a DECstation 5000/200. Slowdowns in each case were computed relative to the respective host machine to make them comparable.

Note: Because not all workload components are instrumented with the User Only Cache2000 simulations, the slowdowns shown here are lower than the slowdowns for Pixie + Cache2000 simulation given in Chapter 2.

Task	Instructions
Kernel Entry and Exit	59
Obtain Faulting Address	33
Direct-mapped Cache Simulation	45
Set Trap	46
Clear Trap	5
Total	188

Table 6.1 Tapeworm Miss Handling Time

This table shows the instructions required to handle different components of a Tapeworm trap for the simulation of direct-mapped caches with 4-word line sizes. A 25-MHz DECstation 5000/200 required 299 cycles to execute the 188 instructions in the handler.

Tapeworm adds overhead only when executing its trap handler:

$$\text{Tapeworm Overhead} = (\text{Trap}_{\text{count}})\,(\text{Trap}_{\text{time}}) \qquad \text{(Eqn 6.4)}$$

where $\text{Trap}_{\text{count}}$ is the number of Tapeworm traps and $\text{Trap}_{\text{time}}$ is the average time to process a single trap. The Tapeworm trap handler, can displace workload instructions from the host I-cache, thus increasing the number of workload I-cache misses. We include the cost of host I-cache pollution as part of the average time to handle a Tapeworm trap. Pollution of the host D-cache is also included as part of the average trap-handling time, but this effect is minor.

The original implementation of the Tapeworm miss handler was written entirely in C and required over 2,000 cycles per miss to execute, similar to the 2,500 cycles required for the same operation in the Wisconsin Wind Tunnel Simulator [Lebeck94]. This cost was so high in comparison with the trace-driven hit and miss times that Tapeworm slowdowns were comparable to Cache2000 slowdowns when simulating small cache structures that frequently trapped.

To improve performance, the handler was optimized by re-writing it entirely in assembly code and by bypassing the usual kernel entry and exit code. The new code uses no execution stack and saves fewer registers, requiring approximately 300 cycles to handle simulated misses in direct-mapped caches with 4-word line sizes (see Table 6.1 for the components of this time).

The expression for Tapeworm overhead explains the shape of the Tapeworm slowdown curves shown in Figure 6.1. Small caches frequently miss, resulting in a change of cache state and a Tapeworm trap. The resulting overall slowdowns for a 1-KB cache are about 3 to 4. As the number of misses decreases for larger caches, the number of traps also decrease to negligible amounts, and slowdowns approach zero for caches as small as 8-KB to 16-KB.

Large fractions of the time in the Tapeworm trap handler could be further reduced with the help of better host hardware support. The 59 instructions required by the kernel entry and exit consist mostly of instructions that save and restore registers and that redirect a trap from the general-exception vector to the Tapeworm trap handler. This cost could be reduced if the host hardware supported a dedicated vector for access-fault traps. The 33 instructions required to obtain a faulting address and the 46 instructions required to set an access trap are due mostly to an awkward interface to the ECC diagnostic logic on the DECstation 5000/200 (see Section 5.2.1 and Section 5.2.2), and could be reduced substantially with a cleaner design.[2] An additional benefit of a cleaner design is that it would reduce the number of working registers required by the trap handler, thus further reducing the cost of kernel entry and exit.

6.2 Speed and Tapeworm II

In the previous section, we introduced a simple model to explain Cache2000 and Tapeworm slowdowns. In this section, we use this model to explain Tapeworm slowdowns in greater detail over a broader range of simulated cache and TLB configurations. Because the following sections do not include comparisons with Cache2000, Tapeworm slowdowns for the remainder of this chapter include all system activity (the `mpeg_play` task, the Mach 3.0 kernel and the user-level BSD and X servers).

6.2.1 Line Size and Slowdown

The slowdowns reported in Section 6.1 were for the simulation of a direct-mapped cache with a 4-word line size. Simulating larger line sizes increases the amount of time in

2. An implementation with instruction breakpoints (see Section 5.1.1) would probably also require less instructions.

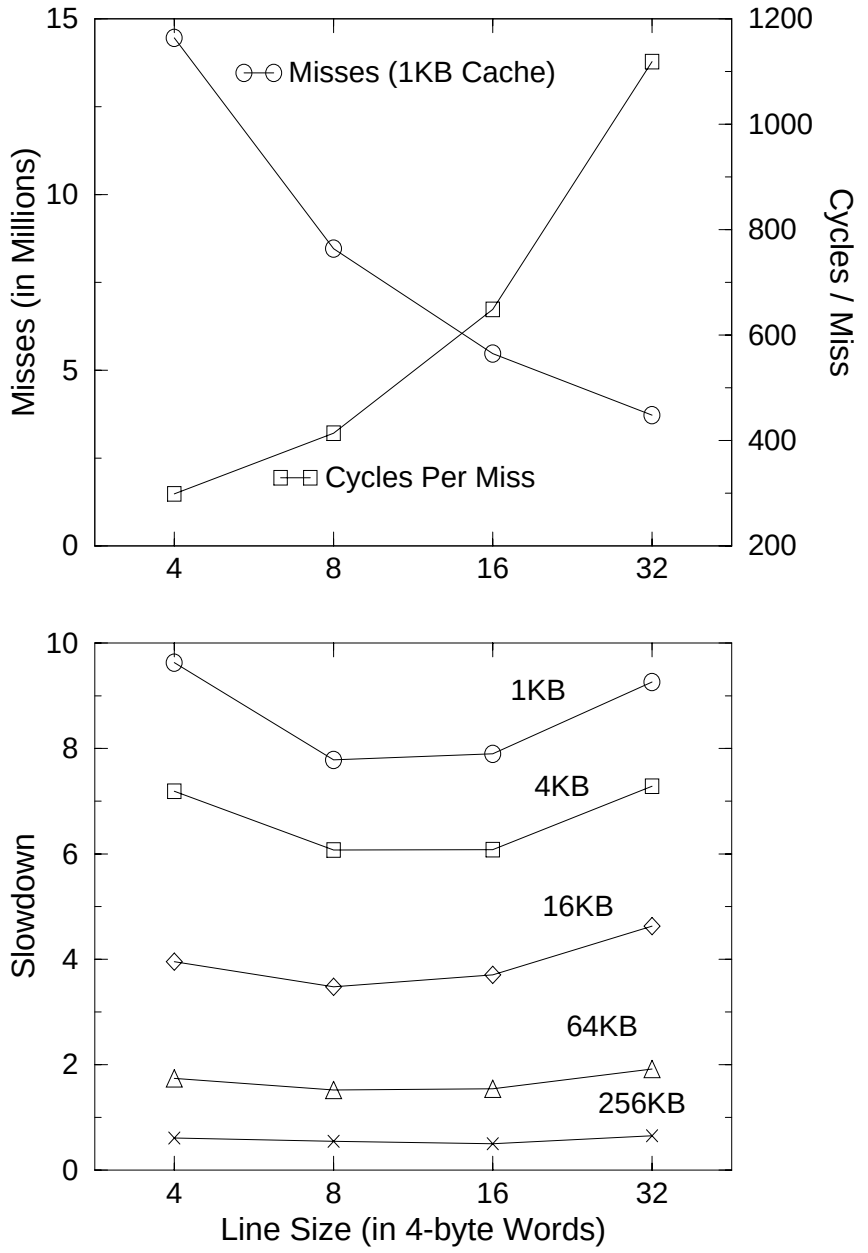


Figure 6.2 Line Size and Slowdown

As the simulated line size increases, the Tapeworm miss handler requires more time to set and clear memory traps. However, as line sizes increase, the total number of simulated misses (and traps) decreases. These two effects are shown in the top plot for a 1KB, direct-mapped I-cache. The bottom plot shows the combined effect of these two factors on overall simulation slowdowns over a range of direct-mapped I-caches and line sizes.

`tw_set_access()` because traps must be set and cleared on larger clusters of memory. On the other hand, increasing the line size decreases the number of cache misses because larger lines better exploit the temporal and spatial locality in memory-reference streams. These two opposing effects are shown at the top of Figure 6.2, which shows that each doubling of the line size reduces the number of cache misses by 30% to 45%, with diminishing reductions in misses as the line size increases. On the other hand, each doubling of line size increases the miss-handling time by 25% to 80%, with larger relative increases in time as line size increases. Setting and clearing memory traps on a cluster of 4 words requires about 100 cycles. For small line sizes, this is a relatively small component of the miss-handling time, which is dominated by the kernel entry and exit code. However, as line sizes grow large, the fraction of miss-handling time spent setting and clear traps begins to dominate, and each doubling of the line size nearly doubles the time to handle a miss.

Recall that the overall Tapeworm overhead is the product of the number of traps and the time required to handle each trap. For direct-mapped caches, traps occur if and only if a reference misses the simulated cache. The resulting slowdowns are shown in Figure 6.2, which shows that initially, increasing the line size reduces overall simulation slowdowns because the number of misses is substantially reduced, but the increase in miss handling times is relatively small. However, for the largest line sizes, slowdowns begin to increase because the relative reduction in misses begins to diminish, while the cost of handling a miss increases geometrically. The “U-shape” of these Tapeworm slowdowns versus line size is very similar to those of the performance of actual hardware caches that exhibit cache pollution due to large lines [Przybylski90].

6.2.2 Associativity and Slowdown

For a very simple replacement policy, such as Random, the simulation of cache associativity does not appreciably change trap-handling times in the Tapeworm II prototype. The MIPS R3000 processor of the Tapeworm host machine has a Random register that is used by the software TLB miss handlers. Tapeworm uses this same register to implement a fast version of random replacement for the simulation of associative caches. Although the trap times do not increase when simulating associative caches with Random replacement, caches with higher degrees of associativity typically exhibit fewer cache misses, resulting in overall decreases in simulation slowdowns.

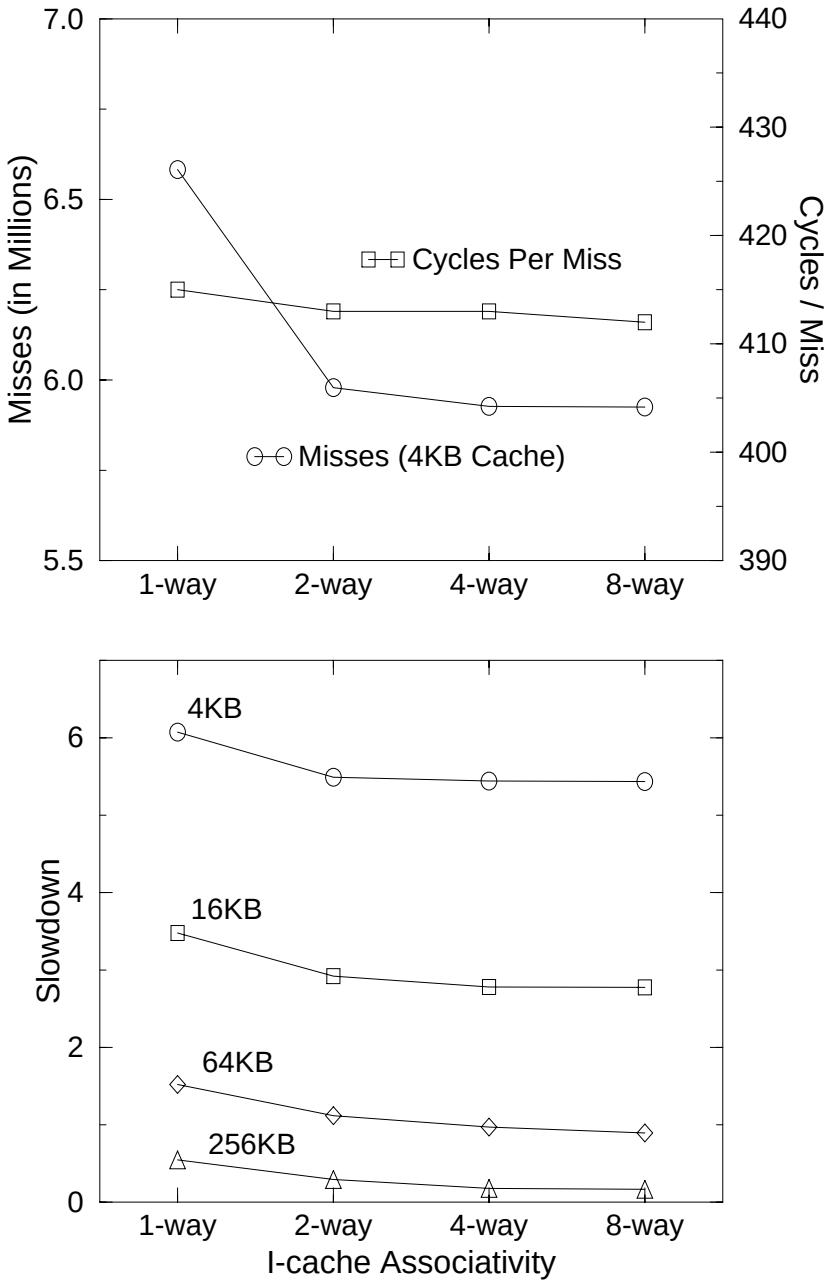


Figure 6.3 Associativity and Slowdown

Simulating associative caches with Random replacement does not appreciably affect the miss handling time. However, greater degrees of associativity can reduce misses. These two effects are shown in the top plot for a 4-KB I-cache with an 8-word line. The bottom plot shows the combined effect of these two factors on overall simulation slowdowns over a range of I-cache sizes and associativities.

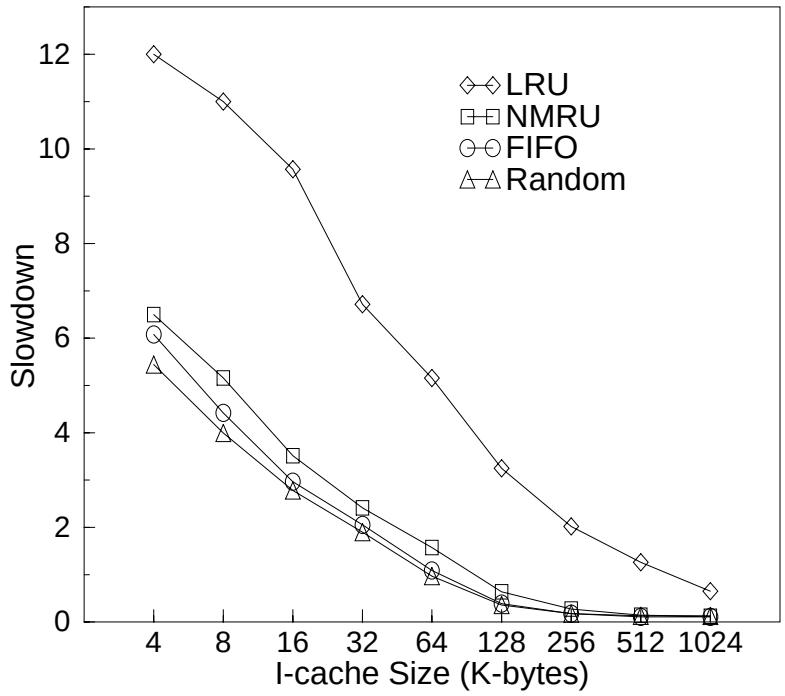


Figure 6.4 Replacement Policy and Slowdowns

This figure shows slowdowns over a range of 4-way set-associative I-caches with an 8-word line size. Miss counts for these policies differ by less than 10%, but slowdowns vary by as much as a factor of two.

Trap-handling times and number of traps (misses) are plotted at the top of Figure 6.3 for caches ranging in associativity from 1-way (direct-mapped) to 8-way. The product of these two terms, plotted at the bottom of the figure, show that trap-driven slowdowns decrease with increasing simulated associativity. Because the greatest reductions in miss counts come from 2-way, set-associativity, overall slowdowns do not decrease substantially for associativities of 4-way or greater.

6.2.3 Replacement Policy and Slowdown

Simulating set-associative caches with replacement policies other than Random can add to the time spent in the trap handler. When simulating a first-in first-out (FIFO) replacement policy, for example, trap handling is slightly more expensive than Random because a wrap-around counter

must be maintained for each set. Figure 6.4 shows the cost of maintaining the FIFO counter by plotting slowdowns for Random replacement and FIFO replacement over a range of cache sizes.

Other replacement policies that depend on order of use, such as not-most-recently-used (NMRU) and least-recently-used (LRU) increase both the trap-handling times and the total number of simulator traps. The additional traps help Tapeworm to order memory locations according to use within each set (recall Section 4.1.2). To fully sort the members of a cache set according to use, our trap-driven algorithm for LRU simulation permits only the most-recently-used memory location in each set to be accessible. This causes the simulation of a 4-way, set-associative cache to trap about as frequently as the simulation of a direct-mapped cache of 1/4-th the size. Figure 6.4 shows that the resulting slowdowns for LRU simulation are about two times as great as they are for the other replacement policies.

The trap-driven algorithm for NMRU replacement is more efficient than LRU because it restricts access to only one memory location in each cache set, the NMRU candidate. Thus, in the 4-way, set-associative cache, 3 of the 4 memory locations in each set are always accessible. Figure 6.4 shows that the resulting simulation speeds for NMRU replacement are only slightly worse than the FIFO simulation speeds.

6.2.4 Cache Indexing Policy and Slowdown

When a cache-miss trap occurs in the current implementation of Tapeworm, the host hardware supplies only the missing physical address to the trap handler. This is all that is required when simulating physically-indexed caches. To simulate virtually-indexed caches, however, a reverse translation must be performed. Tapeworm implements an inverted-page table structure to perform this operation, increasing the cost of trap handling. The plot in Figure 6.5 shows the resulting increase in simulation slowdowns over a range of cache sizes.

6.2.5 Data-cache Simulation and Slowdown

The current implementation of Tapeworm cannot simulate data caches, so we cannot report actual slowdowns for this form of simulation. However, given knowledge of trace-driven

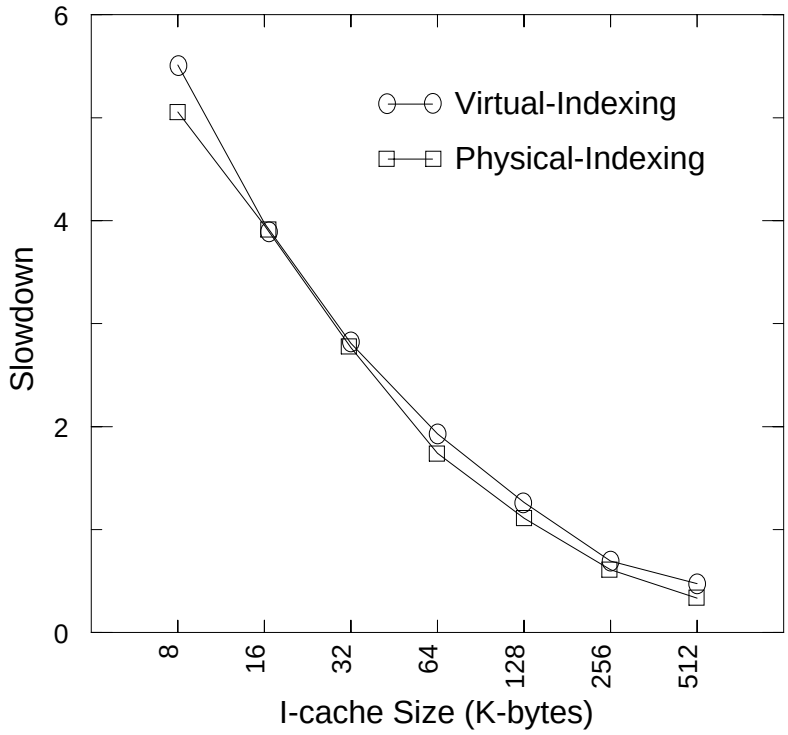


Figure 6.5 Cache Indexing Policy and Slowdowns

Simulating a virtually-indexed cache increases the processing time for each trap because the physical addresses given by the hardware must be translated to a virtual address. This plot shows the difference in slowdown between virtual- and physical-indexing in a direct-mapped I-cache with a 4-word line. Because virtual indexing and physical indexing are the same for caches that are 4KB or smaller, only caches greater than 8-KB are shown.

simulation slowdowns for D-cache simulation, we can speculate on the degree of slowdown that Tapeworm would exhibit if it *could* perform D-cache simulation.

At first, it may seem that simulation of data caches would exhibit inherently larger slowdowns because miss ratios for data caches are typically higher than they are for instruction caches [Gee93]. Although this is sometimes true, Tapeworm overheads are determined not by miss ratios, but by the total number of cache misses. Because data references typically occur at 1/4 to 1/3 of the frequency of instruction references in RISC processors [Hennessy90], I-cache misses can actually outnumber data cache misses even when instruction-miss ratios are lower. In fact, it is often the case that I-cache miss counts are higher than D-cache miss counts for many workloads

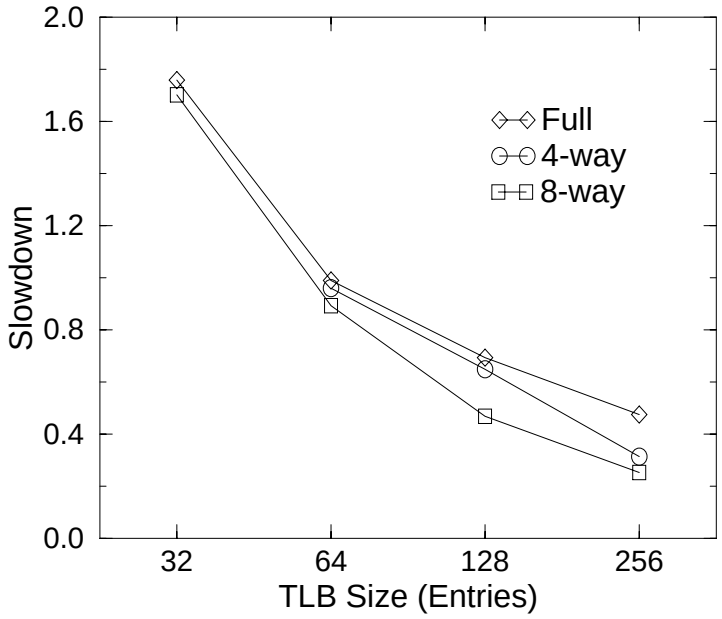


Figure 6.6 TLB Slowdowns

The cost of handling a Tapeworm TLB miss trap is about 700 to 1,000 cycles. TLB misses are, however, infrequent, resulting in low overall simulation slowdowns. This plot overall slowdowns for TLBs with different associativities and sizes (assuming a 4-KB page)

[Nagle94; Maynard94]. Therefore, it is not reasonable to conclude that trap-driven slowdowns of D-caches would be inherently slower than those of I-caches. This assertion is supported by our experiment in Chapter 2, which showed that the trace-driven simulations of both D-cache and I-cache is only about twice as slow as the simulation of I-cache only (see Figure 2.3). Of course, the relative number of instruction and data misses is highly dependent on workload characteristics, so actual slowdowns for I-cache and D-cache simulation ultimately depend on the workload being simulated.

6.2.6 TLB Simulation and Slowdown

The time to handle a simulated TLB miss is much higher than the time for a simulated cache miss (about 700 to 1,000 cycles), because the Tapeworm TLB miss handlers were never optimized by re-writing them in assembly language as was done with the cache-miss handlers. This was

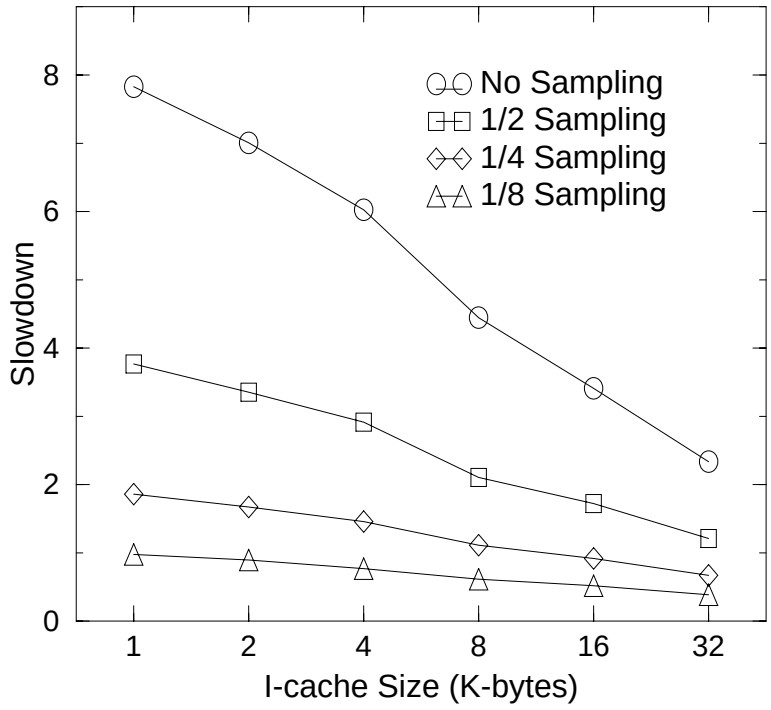


Figure 6.7 Set Sampling and Slowdowns

When a smaller fraction of cache sets is sampled, the number of Tapeworm traps decreases. This results in a decrease in simulation slowdowns that is proportion to the fraction of sets sampled. This figure plots slowdowns for different degrees of sampling over a range of small, direct-mapped I-caches with a line size of 8 words.

deemed unnecessary because TLB misses are far less frequent than cache misses, even for small TLBs. Figure 6.6 shows that TLB simulation slowdowns are, in fact, less than 2 for even the smallest TLBs.

6.2.7 Set Sampling and Slowdown

All slowdowns reported so far are for simulations without sampling. Recall that set sampling uses only a subset of cache lines, causing slowdowns to decrease in direct proportion to the fraction of sets sampled. Figure 6.7 illustrates the benefits of set sampling, showing that sampling 1/*n*-th of the cache sets reduces slowdowns by a factor of *n*. Although set sampling improves

simulation speeds, it also increases the amount of measurement variance. We examine this effect in greater detail in later sections on simulation accuracy.

When 1/8-th set sampling is used, Tapeworm overheads for even the smallest 1-KB I-caches result in less than a doubling of workload run times. Larger caches (> 32-KB) add less 20% to 30% to run times. Slowdowns that are this low make it possible to monitor cache performance while the host workstation is in actual use, opening up new possibilities for real-time memory-system analysis.

6.3 Speed Summary

Early trap-driven simulators exhibited wide variations in their slowdowns, which were highly dependent on the cost and the number of simulator traps. Through a variety of optimizations, the Tapeworm II prototype improves the cost of trap handling over early trap-driven simulators by nearly an order of magnitude. Using Tapeworm's optimized trap handlers, we showed that a trap-driven simulation slowdowns of an I-cache start at about 10 for a direct-mapped, 1-KB cache, and approach zero for modestly larger caches (> 16-KB). TLB slowdowns are similarly low, starting at under 2 for a small 32-entry TLB, and quickly approach zero for larger TLBs. Even in the worst-case, these slowdowns are better than the fastest trace-driven simulators studied in Chapter 2.

Frequently, cache configurations that perform better in actual hardware (e.g., associative caches or caches with tuned line sizes) also exhibit lower trap-driven simulation slowdowns. Other simulation parameters, such as replacement policy or indexing policy, minimally increase overhead, and we argued that D-cache simulation slowdowns are likely to be similar to I-cache slowdowns. When implementing set sampling, Tapeworm simulation slowdowns decrease in direct proportion to the fraction of sets sampled, reducing slowdowns to a range that makes real-time monitoring of cache performance feasible.

The speed of the Tapeworm II prototype is very encouraging, given that its trap handlers are hindered in a variety of ways by the host hardware. Very minimal changes to the hardware of the host machine, such as described in Chapter 5, would enable even faster trap handlers. With the appropriate support, trap-handling times in the range of 50 to 100 cycles are quite feasible, and would reduce Tapeworm slowdowns by a further factor of 3 to 6.

Chapter 7

Accuracy

Measurements of performance delivered by a memory-system simulator are typically subject to two basic types of error, variance and bias. *Variance* refers to differences in measured performance over multiple trial runs of the same workload with a given memory-system configuration, while *Bias* refers to consistent, systematic over- or under-estimates of true performance during multiple experimental trials. Memory-system simulator are subject to many sources of measurement variation and bias, some of which are due to natural effects occurring in real systems, while others are induced by the method of instrumentation and simulation itself. An ideal memory-system simulator is sensitive to the real, naturally-occurring effects, but avoids the induced, artificial sources of measurement error.

The survey in Chapter 2 discusses many of the known sources of trace-driven simulation error, but trap-driven simulation errors have generally gone unstudied. Unanswered questions include:

- How can the different sources of error be isolated and quantified?
- What are the sources of trap-driven simulation bias and variance?
- Which errors are due to naturally-occurring system effects, and which are induced by the simulator itself?
- How do trap-driven simulation errors compare with those of trace-driven simulation?

We will show that trap-driven simulation is not inherently any more or less accurate than trace-driven simulation, but it is more sensitive to certain real-system effects that can cause performance variations.

Workload	Description
mpeg_play	mpeg_play (version 2.0) from the Berkeley Plateau Research Group. Displays 85 frames from a *compressed* video file [Patel92].
jpeg_play	The xloadimage (version 3.0) program written by Jim Frost. Displays two JPEG images.
gs	Ghostscript (version 2.4.1) from the Free Software Foundation. Renders and displays a single postscript page with text and graphics in an X window.
verilog	Verilog-XL (version 1.6b) simulating an experimental GaAs microprocessor.
gcc	The GNU C compiler (version 2.6)
spim	The SPIM MIPS emulator written by James Larus [Larus91]. The input is the SPEC92 espresso program.
sdet	A multiprocess, system performance benchmark which includes programs that test CPU performance, OS performance and I/O performance. From the SPEC SDM benchmark suite.
ousterhout	John Ousterhout's benchmark suite from [Ousterhout89].
xlisp	Lisp interpreter written in C. Configured to solve the 8-queens problem. A SPEC92 benchmark.
espresso	Boolean function minimization. A SPEC92 benchmark.
eqntott	Translates logical representation of boolean equation to a truth table. A SPEC92 benchmark.
kenbus	Simulates user activity in a research-oriented, software development environment. From the SPEC SDM benchmark suite.

Table 7.1 Workload Summary

7.1 Accuracy and Tapeworm II

Our examination of trap-driven simulator errors is based on the Tapeworm II prototype and uses the workloads listed in Table 7.1. Tapeworm includes several features for isolating and measuring the degree of different sources of error. Our study of trap-driven simulator errors will use Tapeworm features such as its ability to control the caching of different workload components, to switch between physical- and virtual-indexing policies, and to enable or disable set sampling.

7.1.1 Sources of Measurement Variation

With trace-driven simulations, the same trace from a given workload is typically used repeatedly to obtain performance measurements for different memory configurations. As a result, trace-driven simulations exhibit no variance if the simulation for a given memory configuration is repeated. The precise sequence of traps that drive a Tapeworm simulation, however, are

impossible to reproduce from run to run because of dynamic system effects. For example, the distributions of physical page frames allocated to a task are different from run to run, which changes the sequence of traps to the simulator. This is precisely the same effect that causes performance variations in actual, physically-indexed caches [Kessler92, Sites88]. Measurement variance can also be caused by Tapeworm itself when it employs set sampling. Cache-miss estimates vary depending on the number and selection of cache sets that are included in a given sample.

Table 7.2 shows the combined effect of page allocation and set sampling on the measured performance of selected workloads from Table 7.1 The table summarizes measurements from 16 trial simulation runs of a 16 K-byte, physically-indexed cache when sampling 1/8th of the cache sets. Standard deviations of the different measurement trials are rather large, ranging from about 10% to as high as 70% of the mean values. In some cases, minimum and maximum values differ from the mean by as much as a factor of two.

Variation due to Set Sampling

To isolate the measurement variation caused by set sampling, we removed page-allocation effects by simulating a virtually-indexed, rather than a physically-indexed cache. The memory references applied to a virtually-indexed cache from run to run of the same workload are unaffected by virtual-to-physical page allocation.

After removing variation due to page allocation, new trials were performed with and without set sampling. The results are shown in Figure 7.1 for `espresso`. Results without sampling show zero variance over multiple trials of the experiment. Notice that results without sampling consistently predict slightly higher miss counts than those with sampling. This measurement bias, discussed more completely in the next section, is due to an increased time dilation effect from the higher slowdown of the non-sampled experiments.

Variation due to Page Allocation

Figure 7.2 shows how page allocation, working in isolation, can vary cache performance. We removed sampling variation and then simulated the same workload (`mpeg_play` in this example) in both a physically-indexed and a virtually-indexed cache. Simulations of the virtually-indexed

Workload	Misses ($\bar{x}$) (x 10^6)	s (x 10^6)		Minimum (x 10^6)		Maximum (x 10^6)		Range (x 10^6)	
eqntott	4.42	2.53	(57%)	3.25	(26%)	13.13	(197%)	9.88	(223%)
espresso	4.91	2.93	(60%)	3.45	(30%)	13.72	(180%)	10.28	(209%)
jpeg_play	18.58	1.34	(7%)	16.26	(13%)	21.96	(18%)	5.71	(31%)
kenbus	20.89	5.30	(25%)	17.10	(18%)	36.37	(74%)	19.27	(92%)
mpeg_play	58.48	7.01	(12%)	47.34	(19%)	68.95	(18%)	21.61	(37%)
ousterhout	31.50	2.61	(8%)	27.09	(14%)	35.03	(11%)	7.94	(25%)
sdet	41.28	8.77	(21%)	32.58	(21%)	63.48	(54%)	30.90	(75%)
xlisp	41.55	31.78	(76%)	15.16	(64%)	104.48	(151%)	89.32	(215%)

Table 7.2 Variation in Measured Memory System Performance

The statistics shown above represent 16 trial runs of each workload in a 16 K-byte, 4-word line, direct-mapped, physically-indexed cache, using 1/8 set sampling. All workload components, including the kernel and servers, were cached. $\bar{x}$ is the mean number of misses and s is the standard deviation of the trial set. Numbers in parenthesis are the percent of the mean value for s and *Range*, and the percent difference from the mean value for *Minimum* and *Maximum*.

Size (KB)	Misses (x 10^6)		
	$\bar{x}$	s	%
With Sampling			
4	1.72	0.13	8%
8	0.89	0.09	10%
16	0.38	0.02	6%
32	0.13	0.01	11%
64	0.02	0.00	3%
128	0.02	0.00	5%
Without Sampling			
4	1.80	0.00	0%
8	0.93	0.00	0%
16	0.41	0.00	0%
32	0.14	0.00	0%
64	0.03	0.00	0%
128	0.02	0.00	1%

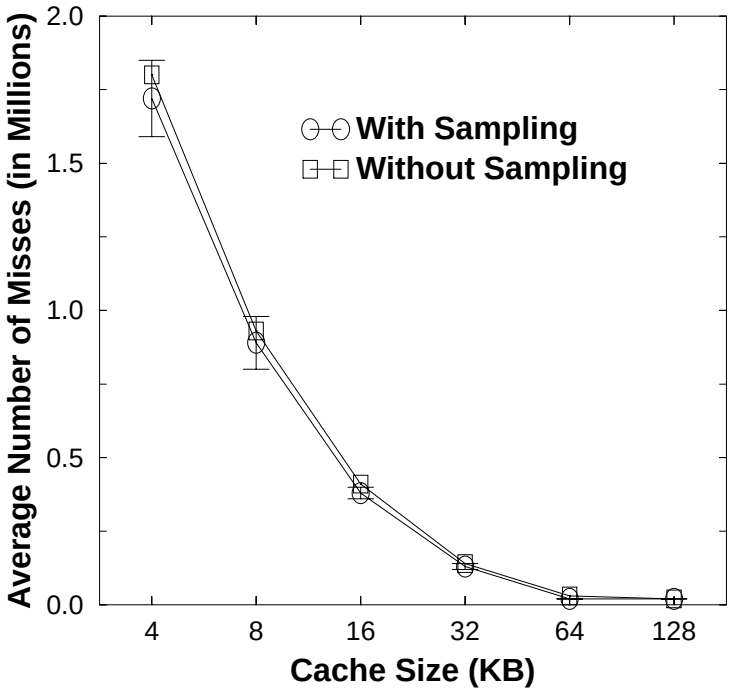


Figure 7.1 Variation due to Set Sampling

This table isolates measurement variance due to set sampling. Tapeworm removed all other sources of variation by considering only activity from the `espresso` process (no kernel or servers) and by simulating virtually-indexed caches (4-word line, direct-mapped). The two sets of data points are for measurements with and without sampling and consist of 16 trials each. The error bars on the plot represent one standard deviation.

cache exhibited zero variation because the sequence of references to the cache is independent of the distribution of physical page frames assigned by the OS from run to run. This is essentially the assumption made by most trace-driven cache simulators. Note that the 4 K-byte, physically-indexed cache simulation results do not vary. This is because the page size on this machine is 4 K-bytes; any page allocation will appear the same because all pages overlap in caches that are 4 K-bytes or smaller.

With the physically-indexed cache, the greatest degree of variation (as a percentage of the mean) appears at a cache size of 32 K-bytes, which is roughly the size of the program text used by `mpeg_play`. This observation is consistent with Kessler's probabilistic model of cache conflicts

Size (KB)	Misses (x 10^6)		
	$\bar{x}$	s	%
Physically Indexed			
4	37.81	0.09	0%
8	22.38	5.89	26%
16	12.07	4.84	40%
32	9.01	5.62	62%
64	5.83	5.96	10%
128	2.92	4.60	15%
Virtually Indexed			
4	37.75	0.00	0%
8	14.03	0.00	0%
16	10.20	0.00	0%
32	1.90	0.00	0%
64	1.38	0.00	0%
128	0.28	0.00	1%

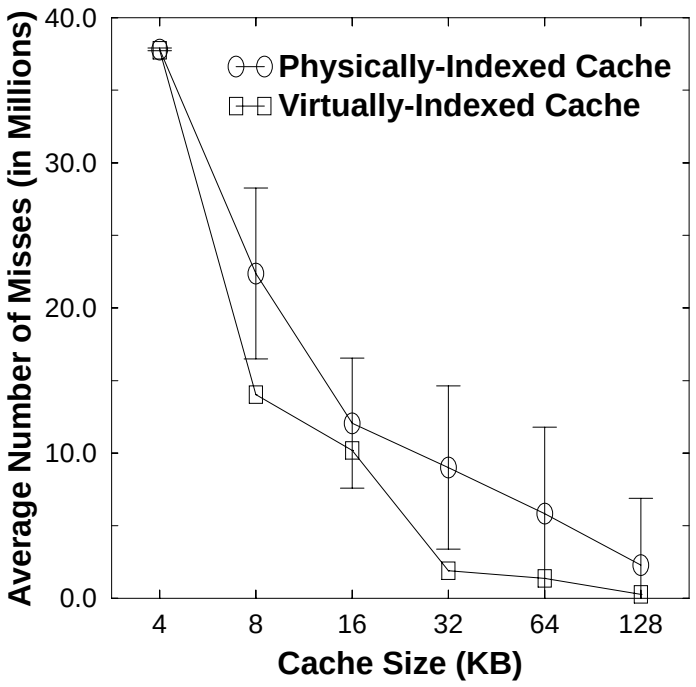


Figure 7.2 Variation Due to Page Allocation

This table shows how page allocation alone can vary cache performance. Tapeworm removed all other sources of variation by considering only activity from the `mpeg_play` process (no kernel or servers), and by not sampling. The two sets of data points are for a physically- and virtually-indexed cache (4-word line, direct-mapped). Each data point is the average of 4 trials. The error bars on the plot represent one standard deviation.

[Kessler91]. Kessler's model predicts that with random page allocation, the probability of cache conflicts peaks when the size of the cache roughly equals the address space size of the workload, and decreases for larger and smaller caches. Figure 7.3 illustrates this effect more clearly for other workloads and over a wider range of cache sizes and associativities. The plot shows that increased cache associativity reduces performance variation. This happens for two reasons. First, increased associativity increases the size of cache required for page allocation to have any affect at all.[1]

1. Increased associativity increases cache size, but does not increase the number of cache sets. Therefore, an 8-KB, 2-way set-associative cache is indexed in the same way as a 4-KB, direct-mapped cache.

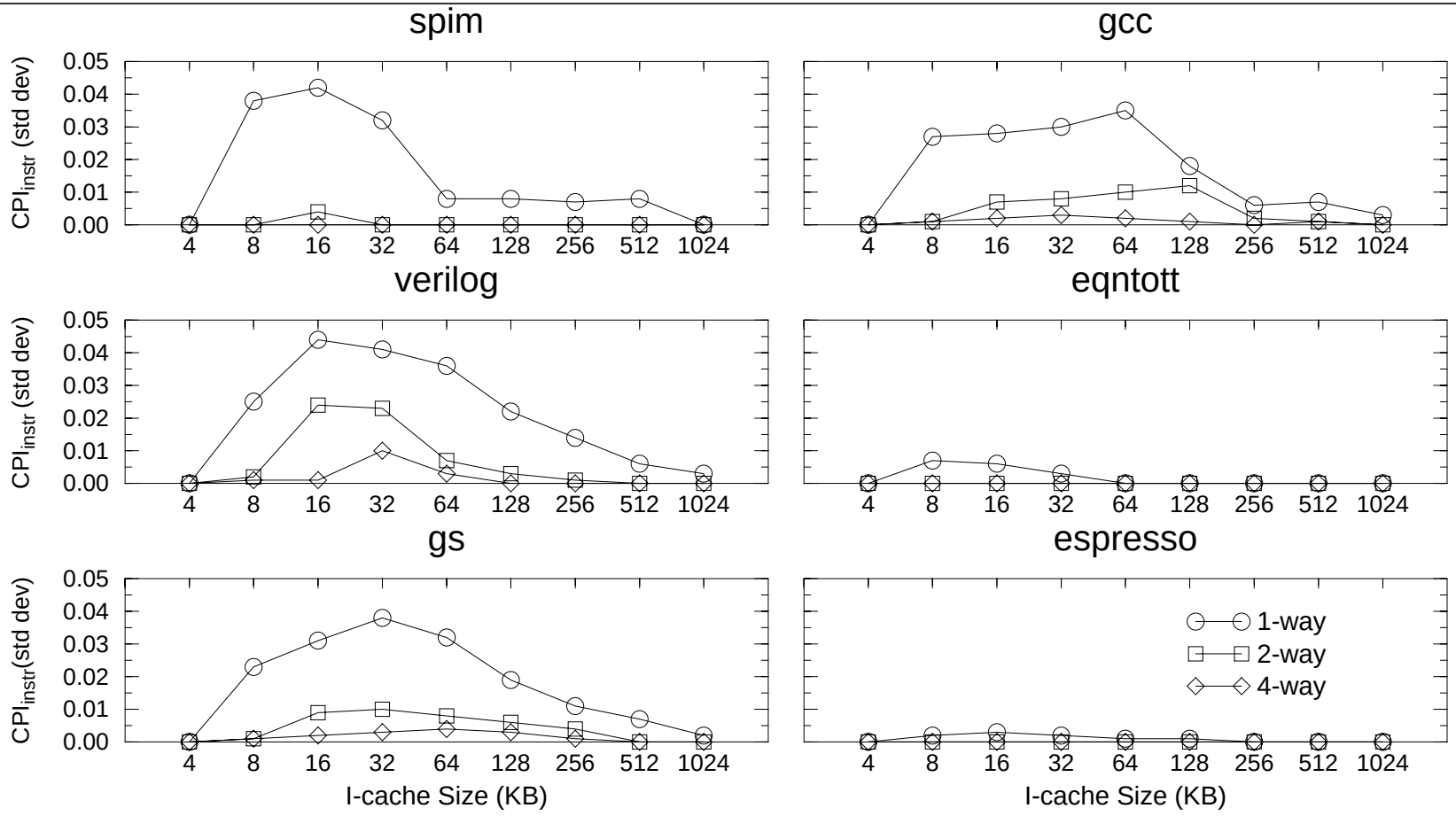


Figure 7.3 Variability in I-cache Performance versus Size and Associativity

These plots show variability in performance over multiple runs of the same workload in a physically-indexed I-cache. Performance varies because the allocation of virtual pages to physical cache page frames is different from run to run. Variability is reported on the y-axis in terms of one standard deviation of CPI_{instr}, the I-cache contribution to CPI.

Second, associativity reduces cache conflict misses, which are the type of cache misses that are affected by page-allocation decisions [Kessler91].

Variation due to page allocation is comparable to, if not larger than, that of set sampling. This suggests that the error introduced by sampling is a reasonable trade for increased speed when simulating physically-indexed caches. Of course, the combined effect of both sources of variance is greater than either in isolation, forcing a larger number of trials to be performed to increase the level of confidence in the mean value.

Variation due to Memory Fragmentation

In addition to page allocation, we have observed other sources of memory-system performance variation due to OS effects, such as substantial increases in TLB misses due to kernel and server memory fragmentation in a long-running system. Because fragmented data structures are more sparsely distributed, they require more page-table entries to map, which can lead to increased TLB misses. Figure 7.4 shows this effect in a plot of TLB performance for a 64-entry TLB against time in a long-running system that repeatedly runs the same workload. Because neither the workload nor the TLB configuration changes from run to run, one would expect repeated simulation trials to result in roughly constant TLB performance. The plot shows, however, sharp increases in misses at the beginning of the second and fourth days.

The first increase, between days 1 and 2, is due to the behavior of the Mach 3.0 kernel memory allocator. This allocator initially works with a 400-KB pool of unmapped kernel memory. Accesses to these locations do not contribute to the TLB miss rate. After long periods of allocation and reclamation, this unmapped memory pool becomes fragmented and the memory allocator must begin to use mapped kernel memory, which does contribute to TLB misses. The step in TLB misses shown in the plot corresponds to the moment in time when the allocator must reach into mapped regions of memory.

The second jump in TLB miss rates is due to a temporary change in workloads. Instead of continuing to repeatedly run the same workload (`ousterhout`), the system was made to run a random series of workloads from our benchmark suite for about an hour. After this period, the repeated execution of ousterhout was resumed. As the plot shows, the OS memory fragmentation that resulted from the brief period of changed workload continued to affect the TLB performance after returning to the repeating `ousterhout` workload.

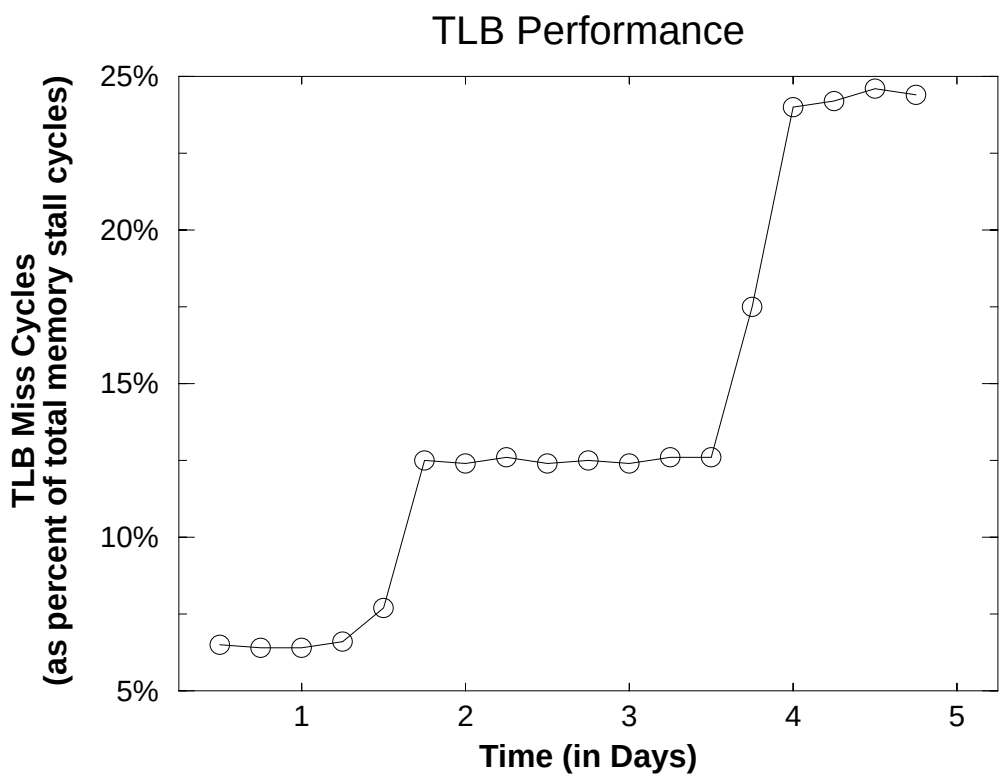


Figure 7.4 Measurement Variation due to Memory Fragmentation

Memory becomes fragmented as time passes, requiring more TLB entries to map. This plot shows TLB performance as a fraction of total memory stall cycles over a period of 5 days, starting with a freshly-booted system. The TLB holds 64 virtual-to-physical mappings and the workload is `ousterhout` which was run repeatedly over the five-day period. Between days 3 and 4, a different set of workloads was run for an hour, and then the repeated `ousterhout` workload was resumed.

The poor TLB performance described above is due to a design problem in the Mach kernel-memory allocator, something that is best solved in software. The problem itself, although easily detected by Tapeworm, would probably go unnoticed in a trace-driven simulation environment that cannot easily conduct multiple experiment trials on an aging system.

Summary of Measurement Variation

Trap-driven simulation results, as produced by the Tapeworm prototype, are subject to both artificial sources of variation (set sampling), as well as natural sources of variation (page-

allocation effects and memory fragmentation). Because Tapeworm is part of an actual, running system, is also sensitive to other system effects, such as link order in a system that implements dynamic linking, or randomness in the order of task scheduling.

Tapeworm's sensitivity to natural sources of performance variation, which may necessitate multiple experimental trials, is not a liability. Performance variations due to page allocation and memory fragmentation are real system effects that should be understood and taken into account when making design decisions. If necessary, however, Tapeworm simulations can be configured to remove these effects and produce measurements with less variation, like those from traditional trace-driven simulators. An example of this is shown in Table 7.3.

7.1.2 Sources of Measurement Bias

With sufficient experimental trials, the variance errors of a workload can be quantified and analyzed. In the absence of other sources of error, the resulting mean value will provide a good estimate of true system performance. In this section we examine more serious forms of measurement error that systematically over- or under-estimates true system performance. Sources of measurement bias are hard to correct for because they are more difficult for the simulator to account for and remove. Nevertheless, we will use certain Tapeworm features to isolate and identify the magnitude of sources of measurement bias, whenever possible.

Bias due to Omitted Workload Components

If a simulation method completely omits memory references made by certain portions of a workload, the accuracy of the resulting simulations will clearly be affected. The most common form of omission is to restrict memory references to a single task. This occurs, for example, when the Cache2000 simulator is driven by Pixie-collected traces. We illustrate the importance of including all workload components (user, server and kernel)[2] by using Tapeworm to measure their individual contributions to the total number of I-cache misses.

2. By *user task*, we mean any of several tasks that are children of the shell from which the workload was initiated. We collect tasks together in our simulations with the Tapeworm inheritance attribute. A *server task* is the X display server or the BSD server, which exist prior to the initiation of a workload. We refer to the *server tasks* and the *kernel* as the *system components* of the workload.

Workload	Misses ($\bar{x}$) (x 10^6)	s (x 10^6)		Minimum (x 10^6)		Maximum (x 10^6)		Range (x 10^6)	
eqntott	4.19	0.10	(2%)	4.11	(2%)	4.26	(2%)	0.15	(4%)
espresso	4.26	0.06	(1%)	4.21	(1%)	4.30	(1%)	0.09	(2%)
jpeg_play	20.60	0.06	(0%)	20.56	(0%)	20.64	(0%)	0.08	(0%)
kenbus	22.03	0.05	(0%)	21.99	(0%)	22.06	(0%)	0.07	(0%)
mpeg_play	53.16	0.06	(0%)	53.12	(0%)	53.20	(0%)	0.08	(0%)
ousterhout	34.69	1.22	(4%)	33.83	(2%)	35.55	(2%)	1.72	(5%)
sdet	41.23	0.00	(0%)	41.22	(0%)	41.23	(0%)	0.00	(0%)
xlisp	21.67	0.19	(1%)	21.53	(1%)	21.80	(1%)	0.27	(1%)

Table 7.3 Measurement Variation Removed

These measurement were made as in Table 7.2, but with variation due to sampling and page allocation removed. This was accomplished by configuring Tapeworm for simulation of virtually-indexed caches without set sampling.

Table 7.4 shows I-cache miss counts and miss ratios for each of our workloads in a 4 K-byte cache. The table shows the number of misses from the kernel, the BSD and X servers, and the user tasks when each is allowed to run in a dedicated cache.[3] The *All Activity* column gives results when each of these workload components share a single cache. Due to cache interference among the individual workload components, the sum of the individual miss columns is less than the *All Activity* column.

Note, first, that the SPEC92 benchmarks `eqntott` and `espresso` exhibit very low miss counts overall. This is consistent with previous observations that many of the SPEC92 benchmarks require only small I-caches to run well [Gee93]. The servers and kernel contribute the majority of total misses, but even with their contribution, the total number of misses is negligible. Other workloads, such as `mpeg_play`, `jpeg_play`, `sdet` and `ousterhout` exhibit the same predominance of server and kernel misses, but with much higher overall miss ratios. In `ousterhout`, for example, the total miss ratio is over 10%, mostly due to the system components and interference effects. A simulator that considers only the user-task component of `ousterhout` would incorrectly estimate the I-cache miss ratio to be less than 1%. The only workload in our suite with a greater fraction of misses coming from a user task is `xlisp` which performs much better in a cache that is only slightly larger.

Bias due to Memory Dilation

The amount of memory used by Tapeworm is small in comparison with many of the trace-driven tools described in Chapter 2. Tapeworm does not cause a program to increase in size due to code annotation, nor does it require large regions of host memory to be reserved for trace buffers. Small amounts of host memory are, however, required for the Tapeworm code and data structures. About 256 K-bytes of physical memory are allocated to Tapeworm at boot time. This removes 64 pages from the free memory pool, resulting in a possible increase in paging activity. This effect can be offset by adding a small amount of additional memory to the host machine.

---

3. The cache is shared by multiple user tasks in the case of `kenbus`, `sdet` and `ousterhout`.

Workload	Miss Counts					
	From Traces	User Tasks	Servers	Kernel	All Activity	Interference
eqntott	0.06 (0.000)	0.07 (0.000)	2.52 (0.002)	2.44 (0.002)	8.44 (0.007)	3.41 (0.003)
espresso	1.60 (0.003)	1.80 (0.003)	2.28 (0.004)	1.96 (0.004)	9.53 (0.018)	3.49 (0.007)
jpeg_play	2.98 (0.002)	3.14 (0.002)	14.58 (0.008)	9.21 (0.005)	36.28 (0.020)	9.35 (0.005)
kenbus	—	7.50 (0.043)	11.89 (0.068)	12.78 (0.073)	45.70 (0.260)	13.53 (0.077)
mpeg_play	37.63 (0.027)	37.91 (0.027)	33.92 (0.024)	19.27 (0.014)	112.5 (0.079)	21.39 (0.015)
ousterhout	—	1.93 (0.003)	18.62 (0.033)	21.72 (0.038)	61.39 (0.108)	19.12 (0.034)
sdet	—	20.14 (0.024)	25.18 (0.031)	18.09 (0.022)	104.6 (0.127)	41.25 (0.050)
xlisp	85.77 (0.061)	90.02 (0.064)	6.31 (0.004)	2.98 (0.002)	135.8 (0.096)	36.55 (0.026)

Table 7.4 Miss Contributions of Different Workload Components

This table gives the number of misses (in millions) and the miss ratios (in parentheses) for different workload components. The data were collected by running separate trials in which each workload was run in a dedicated direct-mapped cache of 4 K-bytes, with a 4-word line. Whenever possible (e.g., for the single-task workloads), *From Traces* gives the miss ratios predicted by a trace-driven simulation using Pixie+Cache2000. *All Activity* gives total miss counts when all workload components share the same cache. Note that because of cache interference effects, the values in this column are greater than the sum of the individual components. This difference is shown in the last column, *Interference*.

All miss ratios are relative to the total number of instructions in the workload. Hence, the miss ratios from each individual component, plus interference, all sum to the total miss ratio given under *All Activity*.

Bias due to Time Dilation

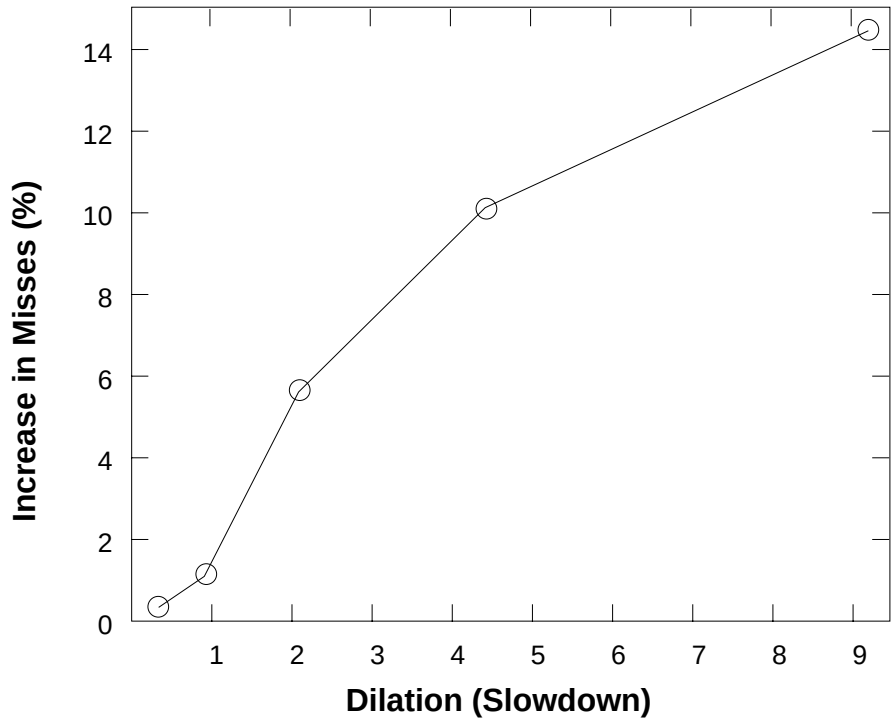
Because Tapeworm slows execution of a system, it is subject to the same form of time dilation errors present in memory traces. One effect of time dilation is that it causes more clock interrupts to occur during the run of a workload, leading to increased cache conflict misses. Figure 7.5 shows the magnitude of error induced by time dilation. Notice that error grows most steeply from slowdowns of 0 to 2, and then levels off for larger slowdowns. Most Tapeworm slowdowns are under 4 where bias tends to be under 10%. Because the amount of slowdown varies from workload to workload, time dilation cannot be removed by a simple adjustment to the clock interrupt frequency as is done in [Borg90, Chen93b]. The most effective way to remove measurement bias due to time dilation is to use set sampling to reduce simulation slowdowns.

Bias due to Non-trapping Memory Locations

The previous sections described forms of measurement bias shared by both trace-driven and trap-driven simulators. One source of bias that is specific to trap-driven simulation is due to the masking of certain Tapeworm memory traps. In the DECstation 5000/200, single-bit ECC errors raise a hardware interrupt line to cause a kernel trap. If interrupts are disabled, a kernel trap cannot occur, resulting in a reduction of simulated cache misses seen by Tapeworm. Because only the kernel runs with interrupts masked, this limitation affects only kernel references. Moreover, only a very small fraction of kernel code is affected. Special code around these regions helps Tapeworm to account for their cache effects, and better host-hardware support for controlling memory access (see Section 5.3.1) would avoid this problem altogether.

Summary of Measurement Bias

Most of the sources of trap-driven simulation bias described in this section also affect trace-driven simulation accuracy. The survey in Chapter 2, for example, revealed that many trace-driven systems also suffer from bias due to memory and time dilation. Similarly, the omission of workload components has been a long-standing problem of trace-driven simulation [Flanagan93]. Tapeworm was designed from the outset to avoid this form of measurement bias, and its ability to flexibly cache different workload components makes it an ideal tool for quantifying this type of error. The current version of Tapeworm does not implement time sampling, but if it did, it would be subject to the same forms of cold-start bias that occurs in trace-driven implementations of time



Dilation (slowdown)	Misses (x 10^6)	Increase %
0.43	90.56	0.0%
0.96	91.54	1.2%
2.08	95.70	5.7%
4.42	99.66	10.1%
9.29	103.57	14.4%

Figure 7.5 Error Due to Time Dilation

Increases in cache misses due to time dilation were measured for the `mpeg_play` workload including all system activity (kernel and servers), running in a physically-addressed 4 K-byte, direct-mapped I-cache with 4-word lines. Time dilation was varied by changing the degree of sampling.

sampling [Laha88; Kessler91]. The one form of bias that is unique to trap-driven simulation is caused by non-trapping memory locations, a problem that could be overcome through better host hardware support. In summary, trap-driven simulation is no more subject to forms of measurement bias than is standard trace-driven simulation.

7.2 Accuracy Summary

We have used the Tapeworm II simulator to identify, isolate and measure a variety of sources of memory-simulation error. Unlike trace-driven simulators, which always obtain the same simulation result with a given trace, trap-driven simulators are sensitive to dynamic-system effects, such as page allocation and memory fragmentation, which cause variations in performance from run to run. This is a positive feature of trap-driven simulation because it provides better insight into the true behavior of real machines.

With respect to artificial sources of measurement variation and bias, trap-driven simulation is subject to many of the same sources of error as trace-driven simulation. In particular, variation due to set sampling, and bias due to time sampling, time dilation, and memory dilation are forms of error that both methods must contend with. The magnitude of these errors, however, is sometimes less with trap-driven simulation (e.g., with memory dilation), and trap-driven simulators are often able to employ certain techniques to minimize the effect of other sources of error (e.g., using set sampling to reduce slowdowns and hence error due to time dilation). One of the most significant features of the Tapeworm II prototype is its ability to monitor all system activity (with the exception of small regions of un-interruptible kernel code). As a result, it is not subject to bias due to omission of workload components. Our measurements showed that this form of error was among the most significant.

Chapter 8

Conclusions and Future Work

Our study of trace-driven and trap-driven simulation methods has enabled us to answer the questions from the introduction:

(1) *Flexibility:* A model based on access constraints enabled us to determine when trap-driven simulation is feasible. Using this model, we developed trap-driven simulation algorithms for a broad range of memory configurations and performance metrics. With a few exceptions (e.g., write-buffer simulation), trap-driven simulation is nearly as flexible as trace-driven simulation.

(2) *Portability:* We examined several methods for implementing trap-driven simulator primitives on existing machines, and showed that most machines provide enough support for trap-driven TLB and I-cache simulation. Unfortunately, some existing machines lack support for D-cache simulation and make the cost of kernel traps unnecessarily expensive. We suggested minor and inexpensive hardware modifications that could easily overcome these problems in future machines.

(3) *Speed:* Our prototype implementation of the Tapeworm II design on existing hardware is faster than the fastest trace-driven simulators over a range of simulated-memory configurations. Its slowdowns start at 10 in the worst case (with a 1-KB cache), and approach 0 for larger or more associative caches. When set sampling is used, slowdowns drop to less than twice the normal workload execution time in the worst case, and add less than 20% to 30% to run times in more common cases. An analysis of time spent in the Tapeworm trap handler showed that inexpensive host-hardware modifications could realistically reduce these slowdowns by a further factor of about 5.

(4) *Accuracy:* Trap-driven and trace-driven simulation share many of the same artificial sources of measurement error, but trap-driven simulation is often more able to reduce the magnitude of their effects. Trap-driven simulators are more sensitive to naturally-occurring system effects that cause variation in the performance of real systems. The Tapeworm II prototype shows that a properly-designed trap-driven simulator can account for the full activity of complex, multi-task workloads that frequently invoke operating-system services.

Trace-driven simulation has evolved and improved over a 30-year period to the point where it is now a well-understood, mature methodology that is unlikely to exhibit substantial further improvements in performance. The method of trap-driven simulation, on the other hand, is only 2 or 3 years old and still in a relatively early stage of development. Nevertheless, in this short time it has equalled or surpassed trace-driven simulation with respect to speed and accuracy.

The main weaknesses of trap-driven simulation are its portability and flexibility. It remains an open question whether trap-driven simulation will be able to make continued advances in these regards. The outcome will depend, in large part, on the willingness of computer architects to make minor modifications in future designs to better support trap-driven simulation. Even with such support, trap-driven simulation is not suited to certain forms of architectural simulation, such as instruction-pipeline simulation, or simulations that require detailed, cycle-by-cycle accounting of time.

In short, trap-driven simulation offers important advantages over trace-driven simulation, the most important of which is speed. It is unlikely, however, that it will ever completely replace trace-driven simulation due to limits in the detail of simulation that it can perform. Given this, the role of trap-driven simulation is perhaps best understood as a new tool among the many used by computer designers (see Figure 8.1). The selection of the most appropriate tool depends on the level of simulation detail that is required. In the same way that trace-driven simulation cannot replace circuit-level simulation, trap-driven simulation is unlikely to ever completely replace trace-driven simulation.

8.1 Future Work

Work in trap-driven simulation and analysis could be developed and extended in many ways:

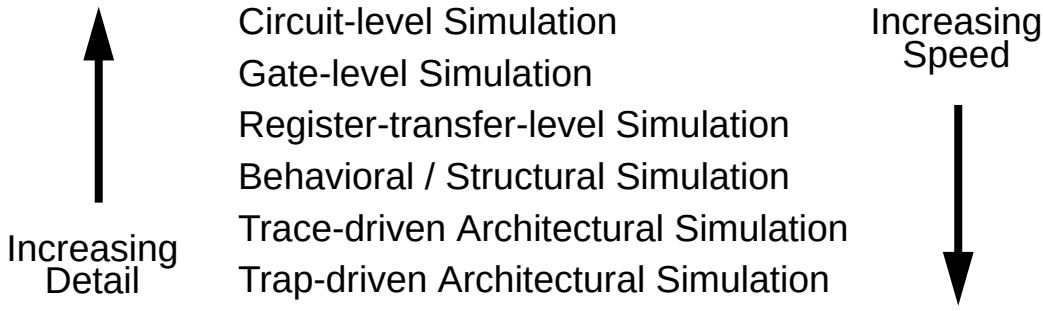


Figure 8.1 Computer Design Tools

The level of simulation detail required determines the best selection of simulator. In general, as the level of simulation detail increases, simulation speed decreases. Trap-driven simulation gives computer designers a new option for memory-system simulation. It is faster, but less detailed than other forms of simulation.

- *Porting Efforts*: Implementing trap-driven simulation on new platforms will increase acceptance of the method by making it available to a larger group of users.

- *Trap-driven Simulator Design*: Redesigning Tapeworm as a filtered-trace generator, as described in Section 4.2.4, would enable a larger fraction of Tapeworm code to run outside the kernel, thus making it easier to install new simulation algorithms. Such a redesign would simplify the Tapeworm trap handlers, potentially leading to further improvements in overall simulation speed.

- *Software Performance Tuning*: Our perspective of memory-system analysis has been that of a computer architect who needs tools to evaluate hardware-design alternatives for *future* memory systems. However, the speed of trap-driven simulation, combined with its ability to dynamically monitor complex workloads in running systems also makes it an ideal tool for software developers who wish to tune their applications to an *existing* computer's memory system.

- *Real-time Performance Analysis*: When Tapeworm employs set sampling in simulations of moderate-size caches (> 16-KB), slowdowns are low enough to be virtually imperceptible to a user of the monitored system. This opens up the

possibility of real-time memory-system monitoring and the detection of performance problems that cannot be identified by traditional batch simulations.

- *Software Engineering*: Kernel traps could be the basis for new approaches to program testing and debugging. It may be possible to use traps to efficiently measure the percentage of program branches, code paths, or static variables exercised by a given set of test inputs to a program. Such a tool would be analogous to the fault-coverage analysis used by integrated-circuit designers to determine the effectiveness of a given set of test vectors in causing all nodes in a logic design to switch on and off.

The main hinderance to trap-driven simulation and analysis tools is insufficient host-hardware support. We have shown that the necessary modifications to host hardware are minor, and have justified them through our successful implementation of the Tapeworm II prototype. We hope that computer designers will choose to support trap-driven simulation in their future designs. In return for this small investment, they will obtain a powerful new tool for analyzing memory systems, a tool that will help to more quickly evaluate the memory requirements of new applications, and thus reduce the chances of costly design mistakes.

Appendices

Appendix A ISCA Paper Abstracts

In addition to serving as prototypes for the exploration of trap-driven simulation issues in this dissertation, Tapeworm and Tapeworm II have also enabled studies of the interaction between modern operating systems and computer architecture. The abstracts of three case studies that used the Tapeworm simulators are included in this appendix as an indication of the types of memory-system studies that can be performed with trap-driven simulation.

Design Tradeoffs for Software-Managed TLBs

David Nagle, Richard Uhlig, Tim Stanley,
Stuart Sechrest, Trevor Mudge & Richard Brown

Abstract: An increasing number of architectures provide virtual memory support through software-managed TLBs. However, software management can impose considerable penalties, which are highly dependent on the operating system's structure and its use of virtual memory. This work explores software-managed TLB design tradeoffs and their interaction with a range of operating systems including monolithic and microkernel designs. Through hardware monitoring and simulation, we explore TLB performance for benchmarks running on a MIPS R2000-based workstation running Ultrix, OSF/1, and three versions of Mach 3.0.

Results: New operating systems are changing the relative frequency of different types of TLB misses, some of which may not be efficiently handled by current architectures. For the same application binaries, total TLB service time varies by as much as an order of magnitude under different operating systems. Reducing the handling cost for kernel TLB misses reduces total TLB service time up to 40%. For TLBs between 32 and 128 slots, each doubling of the TLB size reduces total TLB service time by as much as 50%.

Keywords: Translation Lookaside Buffer (TLB), Simulation, Hardware Monitoring, Operating Systems.

Appeared in the 20th International Symposium on Computer Architecture, San Diego, California, May 1993.

Optimal Allocation of On-chip Memory for Multiple-API Operating Systems

David Nagle, Richard Uhlig, Trevor Mudge & Stuart Sechrest

Abstract: The allocation of die area to different processor components is a central issue in the design of single-chip microprocessors. Chip area is occupied by both core execution logic, such as ALU and FPU datapaths, and memory structures, such as caches, TLBs, and write buffers. This work focuses on the allocation of die area to memory structures through a cost/benefit analysis. The cost of memory structures with different sizes and associativities is estimated by using an established area model for on-chip memory. The performance benefits of selecting a given structure are measured through a collection of methods including on-the-fly hardware monitoring, trace-driven simulation and kernel-based analysis. Special consideration is given to operating systems that support multiple application programming interfaces (APIs), a software trend that substantially affects on-chip memory allocation decisions.

Results: Small adjustments in cache and TLB design parameters can significantly impact overall performance. Operating systems that support multiple APIs, such as Mach 3.0, increase the relative importance of on-chip instruction caches and TLBs when compared against single-API systems such as Ultrix.

Keywords: On-chip Memory, Cache, TLB, Multiple-API Operating System, Mach

Appeared in the 21st International Symposium on Computer Architecture, Chicago, Illinois, April, 1994.

Instruction Fetching: Coping with Code Bloat

Richard Uhlig, David Nagle, Trevor Mudge, Stuart Sechrest & Joel Emer

Abstract: Previous research has shown that the SPEC benchmarks achieve low miss ratios in relatively small instruction caches. This paper presents evidence that current software-development practices produce applications that exhibit substantially higher instruction-cache miss ratios than do the SPEC benchmarks. To represent these trends, we have assembled a collection of applications, called the *Instruction Benchmark Suite* (IBS), that provides a better test of instruction-cache performance. We discuss the rationale behind the design of IBS and characterize its behavior relative to the SPEC benchmark suite. Our analysis is based on trace-driven and trap-driven simulations and takes into full account both the application and operating-system components of the workloads.

This paper then reexamines a collection of previously-proposed hardware mechanisms for improving instruction-fetch performance in the context of the IBS workloads. We study the impact of cache organization, transfer bandwidth, prefetching, and pipelined memory systems on machines that rely on the use of relatively small primary instruction caches to facilitate increased clock rates. We find that, although of little use for SPEC, the right combination of these techniques substantially benefits IBS. Even so, under IBS, a stubborn lower bound on the instruction-fetch CPI remains as an obstacle to improving overall processor performance.

Key words: code bloat, address traces, caches, instruction fetching.

Appeared in the 22nd International Symposium on Computer Architecture, Santa Magherita Ligure, Italy, June, 1995.

Bibliography

Bibliography

[Accetta86] Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, USENIX, 1986.

[Agarwal86] Agarwal, A., Sites, R. L. and Horowitz, M. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, Tokyo, Japan, IEEE, 119-127, 1986.

[Agarwal88] Agarwal, A., Hennessy, J. and Horowitz, M. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems* **6** (4): 393-431, 1988.

[Agarwal89] Agarwal, A. Analysis of cache performance for operating systems and multiprogramming. Ph.D. dissertation, Stanford. 1989.

[Agarwal90] Agarwal, A. and Huffman, M. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, ACM, 48-57, 1990.

[Alexander85] Alexander, C. A., Keshlear, W. M. and Briggs, F. Translation buffer performance in a UNIX environment. *Computer Architecture News* **13** (5): 2-14, 1985.

[Alexander86] Alexander, C., Keshlear, W., Cooper, F. and Briggs, F. Cache memory performance in a UNIX environment. *Computer Architecture News* 14 : 14-70, 1986.

[Alpert88] Alpert, D. and Flynn, M. Performance trade-offs for microprocessor cache memories. *IEEE Micro* (Aug): 44-54, 1988.

[Alverson90] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B. The tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, 1-6, 1990.

[AMD91] AMD. *Am29050 Microprocessor User's Manual*. Sunnyvale, CA, 1991.

[AMD93] AMD. *Am486 DX/DX2 Microprocessor Hardware Reference Manual*. Sunnyvale, CA, Advanced Micro Devices, Inc., 1993.

[Anderson91] Anderson, T. E., Levy, H. M., Bershad, B. N. and Lazowska, E. D. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, ACM, 108-119, 1991.

[Appel91] Appel, A. and Li, K. Virtual memory primitives for user programs. In *Pro-*

ceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM, 96-107, 1991.

[Baer87] Baer, J.-L. and Wang, W.-H. Architectural choices for multi-level cache hierarchies. In *Proceedings of the 16th International Conference on Parallel Processing*, 258-261, 1987.

[Baer88] Baer, J.-L. and Wang, W.-H. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 73-80, 1988.

[Bala94] Bala, K., Kaashoek, M. F. and Weihl, W. E. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, 243-253, 1994.

[Ball92] Ball, T. and Larus, J. Optimally profiling and tracing programs. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, 1992.

[Becker93] Becker, J. and Park, A. An analysis of the information content of address and data reference streams. In *Proceedings of the 1993 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Santa Clara, CA, 262-263, 1993.

[Bedichek90] Bedichek, R. Some efficient architecture simulation techniques. In *Proceedings of the Winter 1990 USENIX Conference*, 53-63, 1990.

[Bedichek94] Bedichek, R. *The meerkat multicomputer: tradeoffs in multicomputer architecture*. Ph.D. dissertation, University of Washington. 1994.

[Bershad90] Bershad, B. N., Anderson, T. E., Lazowska, E. D. and Levy, H. M. Lightweight remote procedure call. *ACM Transactions on Computer Systems* **8** (1): 37-55, 1990.

[Bershad92] Bershad, B. N. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Proceedings of the Micro-kernels and Other Kernel Architectures*, Seattle, Washington, USENIX, 205-211, 1992.

[Bershad94a] Bershad, B. N., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S. and Sirer, E. G. *SPIN* - An extensible microkernel for application-specific operating system services. *University of Washington Technical Report 94-03-03*, 1994.

[Bershad94b] Bershad, B., Lee, D., Romer, T. and Chen, B. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, ACM, 158-170, 1994.

[Biomation91] Biomation. *Biomation CLAS 4000 Application Note 4032*. Cupertino, CA, Biomation. 1991.

[Black89] Black, D. L., Rashid, R. F., Golub, D. B., Hill, C. R. and Baron, R. V. Translation lookaside buffer consistency: a software approach. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, ACM, 113-122, 1989.

[Black92] Black, D. L., Golub, D. B., Julin, D. P., Rashid, R. F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G. and Bohman, D. Microkernel operating system architecture and mach. In *Proceedings of the Micro-kernels and Other Kernel Architectures Workshop*, Seattle, Washington, USENIX, 11-30, 1992.

[Bomberger92] Bomberger, A., Hardy, N., Frantz, A. P., Landau, C. R., Frantz, W. S., Shapiro, J. S. and Hardy, A. C. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Micro-Kernels and Other Kernel Architectures Workshop*, Seattle, Washington, USENIX, 95-112, 1992.

[Borg89] Borg, A., Kessler, R., Lazana, G. and Wall, D. Long address traces from RISC machines: generation and analysis. *DEC Western Research Lab Technical Report 89/14*, 1989.

[Borg90] Borg, A., Kessler, R. and Wall, D. Generation and analysis of very long address traces. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE, 1990.

[Braunstein89] Braunstein, A., Riley, M. and Wilkes, J. Improving the efficiency of UNIX file buffer caches. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, ACM, 71-82, 1989.

[Bray90] Bray, B., Lynch, W. and Flynn, M. J. Page allocation to reduce access time of physical caches. *Technical Report, Stanford University, Computer Systems Laboratory. CSL-TR-90-454*, 1990.

[Brunner91] Brunner, R. A. *VAX Architecture Reference Manual*. Digital Press, 1991.

[Budd91] Budd, T. *An Introduction to Object-Oriented Programming*. Addison-Wesley Publishing IBSN 0-201-54709-0, 1991.

[Calder94] Calder, B., Grunwald, D. and Zorn, B. Quantifying behavioral differences between C and C++ programs. *Technical Report, The Department of Computer Science, University of Colorado. CU-CS-698-94*, 1994.

[Campbell91] Campbell, R. H., Johnston, G. M., Madany, P. W. and Russo, V. F. *Principles of object-oriented operating system design*, 1991.

[Chao90] Chao, C., Mackey, M. and Sears, B. Mach on a virtually addressed cache architecture. In *Proceedings of the USENIX MACH Workshop*, Burlington, Vermont, USENIX, 31-51, 1990.

[Chase92] Chase, J. S., Levy, H. M., Baker-Harvey, M. and Lazowska, E. D. How to use a 64-Bit virtual address space. *Technical Report, University of Washington.*

92-03-02, 1992.

[Chen92] Chen, J. B., Borg, A. and Jouppi, N. P. A simulation based study of TLB performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, IEEE, 114-123, 1992.

[Chen93a] Chen, B. Software methods for system address tracing. *Technical Report CMU-CS-93-188*, 1993.

[Chen93b] Chen, B. Software methods for system address tracing. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa, California, 1993.

[Chen93c] Chen, B. and Bershad, B. The impact of operating system structure on memory system performance. In *Proceedings of the 14th Symposium on Operating System Principles*, 1993.

[Chen94a] Chen, B. Memory behavior of an X11 window system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, 1994.

[Chen94b] Chen, B., Wall, D. and Borg, A. Software methods for system address tracing: implementation and validation. *Technical Report, Carnegie-Mellon University, DEC Western Research Lab, DEC Network Systems Laboratory.* 1994.

[Cheriton84] Cheriton, D. R. The V kernel: A software base for distributed systems. *IEEE Software* **1** (2): 19-42, 1984.

[Cheriton88] Cheriton, D. The vmp multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 14th Annual Sumposium on Computer Architecture*, 1988.

[Chu85] Chu, C. *MILS: Mips instruction level simulator.* 1985.

[Clark83] Clark, D. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems* **1** : 24-37, 1983.

[Clark85a] Clark, D. W. and Emer, J. S. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems* **3** (1): 31-62, 1985.

[Clark85b] Clark, D. W., Bannon, P. J. and Keller, J. B. Measuring VAX 8800 performance with a histogram hardware monitor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, IEEE, 176-185, 1985.

[Cmelik93] Cmelik, R. and Keppel, D. Shade: A fast instruction-set simulator for execution profiling. *University of Washington Technical Report UWCSE 93-06-06*. 1993.

[Cmelik94] Cmelik, B. and Keppel, D. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, ACM, 128-137, 1994.

[Covington88] Covington, R. C., Madala, S., Mehta, V., Jump, J. R. and Sinclair, J. B. The Rice parallel processing testbed. In *Proceedings of the 1988 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM, 4-11, 1988.

[Custer93] Custer, H. *Inside Windows NT*. Redmond, Washington, Microsoft Press, 1993.

[Cvetanovic94] Cvetanovic, Z. and Bhandarkar, D. Characterization of Alpha AXP performance using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Ill., IEEE, 1994.

[Cypress90] Cypress. *SPARC RISC Users Guide*. ROSS Technology Inc, A Cypress Semiconductor Company, 1990.

[Davis91] Davis, H., Goldschmidt, S. and Hennessy, J. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, 99-107, 1991.

[Dean91] Dean, R. W. and Armand, F. Data movement in kernelized systems. In *Proceedings of the Micro-kernels and Other Kernel Architectures Workshop*, Seattle, Washington, USENIX, 243-261, 1991.

[DeMoney86] DeMoney, M., Moore, J. and Mashey, J. Operating system support on a RISC. In *Proceedings of COMPCON*, 138-143, 1986.

[Digital81] Digital. *VAX Architecture Handbook*. Digital, 1981.

[Digital86] Digital. *VAX architecture handbook*. Bedford, MA, Digital Equipment Corporation, 1986.

[Digital92] Digital. *Alpha Architecture Handbook*. USA, Digital Equipment Corporation, 1992.

[Diwan93] Diwan, A., Tarditi, D. and Moss, E. Memory Subsystem Performance of Programs with Intensive Heap Allocation. *Carnegie Mellon University Technical Report CS TR 93-227*. 1993.

[Draves91] Draves, R. P., Bershad, B. N., Rashid, R. F. and Dean, R. W. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 122-136, 1991.

[Druschel92] Druschel, P., Peterson, L. L. and Hutchinson, N. C. Beyond micro-kernel design: decoupling modularity and protection in Lipto. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, IEEE, 512-520, 1992.

[Eggers88] Eggers, S. J. and Katz, R. H. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Conference on Computer Architecture*, Honolulu, HI, 373-383, 1988.

[Eggers90] Eggers, S., Keppel, D., Koldinger, E. and Levy, H. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, 37-47, 1990.

[Eickemeyer88] Eickemeyer, R. and Patel, J. Performance evaluation of on-chip register and cache organizations. In *Proceedings of the 15th Annual Symposium on Computer Architecture*, Honolulu, Hawaii, 64-72, 1988.

[Emer84] Emer, J. and Clark, D. A characterization of processor performance in the VAX-11/780. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, Ann Arbor, MI, IEEE, 301-309, 1984.

[Farrens89] Farrens, M. and Pleszkun, A. Improving performance of small on-chip instruction caches. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ACM, 234-241, 1989.

[Flanagan92] Flanagan, K., Grimsrud, K., Archibald, J. and Nelson, B. BACH: BYU address collection hardware. *Brigham Young University Technical Report TR-A150-92.1*. 1992.

[Flanagan92a] Flanagan, J. K., Nelson, B. E., Archibald, J. K. and Grimsrud, K. BACH: BYU address collection hardware, the collection of complete traces. In *Proceedings of the 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 128-137, 1992.

[Flanagan93] Flanagan, J. K., Nelson, B. E. and Archibald, J. K. The inaccuracy of trace-driven simulation using incomplete trace data, Brigham Young University. 1993.

[Flanagan93a] Flanagan, J. K. A new methodology for accurate trace collection and its application to memory heirarchy performance modeling. Brigham Young University. 1993.

[Flanagan93b] Flanagan, J. K., Nelson, B. E., Archibald, J. K. and Grimsrud, K. Incomplete trace data and trace-driven simulation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS*, 203-209, 1993.

[Flanagan94] Flanagan, J. K. *Personal Communication*. 1994.

[Forin91] Forin, A., Golub, D. and Bershad, B. An I/O system for Mach 3.0. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, USENIX, 163-176, 1991.

[Fuentes93] Fuentes, C. Hardware support for operating systems. *University of Michigan Technical Report*. 1993.

[Gee93] Gee, J., Hill, M., Pnevmatikatos, D. and Smith, A. J. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro* (August): 17-27, 1993.

[Ginsberg93] Ginsberg, M., Baron, R. V. and Bershad, B. N. Using the mach communica-

tion primitives in X11. Carnegie Mellon University Technical Report. 1993.

[Golub90] Golub, D., Dean, R., Forin, A. and Rashid, R. Unix as an application program. In *Proceedings of the USENIX Summer Conference*, USENIX, 1990.

[Golub91] Golub, D. and Draves, R. Moving the default memory manager out of the Mach kernel. In *Proceedings of the USENIX Mach Symposium*, Monterey, CA, USENIX, 177-188, 1991.

[Goodman83] Goodman, J. Using cache memory to reduce processor memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, Stockholm, Sweden, 124-131, 1983.

[Goodman86] Goodman, J. and Hsu, W.-C. On the use of registers vs. cache to minimize memory traffic. In *Proceedings of the 13th International Symposium on Computer Architecture*, 375-383, 1986.

[Goscinski91] Goscinski, A. *Distributed Operating Systems*. Singapore, Addison-Wesley Publishers Ltd., 1991.

[Grimsrud92] Grimsrud, K., Archibald, J., Frost, R., Nelson, B. and Flanagan, K. Estimation of simulation error due to trace inaccuracies. In *Proceedings of the 26th Asilomar Conference on Signals, Systems, and Computers*, 1992.

[Grimsrud93] Grimsrud, K. Address translation of BACH i486 traces. Brigham Young University. 1993.

[Grimsrud93a] Grimsrud, K., Archibald, J., Ripley, M., Flanagan, K. and Nelson, B. BACH: A hardware monitor for tracing microprocessor-based systems. In *Proceedings of the Microprocessors and Microsystems Conference*, 1993.

[Grimsrud93b] Grimsrud, K. S. *Quantifying Locality.* Ph.D. dissertation, Brigham Young University. 1993.

[Gwennap94] Gwennap, L. HP-7200 Enables Inexpensive MP Systems, *Microprocessor Report, Vol 8, Issue 3, 1994*. (http://www.dmo.hp.com/wsg/strategies/parisc5.html).

[Hammerstrom77] Hammerstrom, D. and Davidson, E. Information content of CPU memory referencing behavior. In *Proceedings of the 4th International Symposium on Computer Architecture*, 184-192, 1977.

[Happel92] Happel, L. P. and Jayasumana, A. P. Performance of a RISC machine with two-level caches. *IEE Proceedings-E* **139** (3): 221-229, 1992.

[Hartman94] Hartman, J. H. The Zebra striped network file system. *Faculty Candidate Talk at Uof M*. 1994.

[Hennessy90] Hennessy, J. L. and Patterson, D. A. *Computer Architecture A Quantitative Approach*. San Mateo, Morgan Kaufmann, 1990.

[Henry83] Henry, R. R. VAX address and instruction traces. University of California at

Berkeley Technical Report. 1983.

[HP90] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard, Inc., 1990.

[HP91] Hewlett-Packard. *Test and Measurement Catalog*. Santa Clara, CA, Marketing Communications, 1991.

[Hill84] Hill, M. and Smith, A. J. Experimental evaluation of on-chip microprocessor cache memories. In 11th *Annual International Symposium on Computer Architecture*, Ann Arbor, Michigan, 158-166, 1984.

[Hill87] Hill, M. *Aspects of cache memory and instruction buffer performance*. Ph.D. dissertation, The University of California at Berkeley. 1987.

[Hill89] Hill, M. and Smith, A. Evaluating associativity in CPU caches. *IEEE Transactions on Computers* **38** (12): 1612-1630, 1989.

[Hodges64] Hodges, J. L. and Lehmann, E. L. *Basic concepts of probability and statistics*. San Fransciso, CA, Holden-day, Inc., 1964.

[Holliday90a] Holliday, M. and Ellis, C. Accuracy of memory reference traces of parallel computations in trace-driven simulation. *Department of Computer Science, Duke University, Durham, NC. Technical Report CS-1990-8*. 1990.

[Holliday90b] Holliday, M. and Ellis, C. An example of correct global trace generation. *Scalable Shared Memory Multiprocessors*. Kluwer Academic. 1990.

[Holliday91] Holliday, M. Techniques for cache and memory simulation using address reference traces. *International journal in computer simulation* **1** : 129-151, 1991.

[Hsu89] Hsu, P. *Introduction to Shade*. Sun Microsystems. 1989.

[Huck93] Huck, J. and Hays, J. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, IEEE, 39-50, 1993.

[Hwu89] Hwu, W.-m. and Chang, P. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th International Symposium on Computer Architecture*, Jerusalem, Isreal, IEEE, 242-251, 1989.

[IBM90] IBM. *IBM RISC System/6000 Technology.* Austin , TX, IBM, 1990.

[Intel90] Intel. *i860 64-bit Microprocessors Programmer's Manual*. Santa Clara, CA, Intel Corporation, 1990.

[ISCA92] ISCA92. *The Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, ACM SIGARCH and IEEE Computer Society, 1-424, 1992.

[ISCA93] ISCA93. *The Proceedings of the 20th Annual International Symposium on*

Computer Architecture, San Diego, California, ACM SIGARCH and IEEE Computer Society, 1-360, 1993.

[ISCA94] ISCA94. *The Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, ACM SIGARCH and IEEE Computer Society, 1-394, 1994.

[Jacob95] Jacob, B., Mudge, T., Emer, J. Hybrid TLBs for High Performance Microprocessors, in preparation. (http://www.umich.edu/~blj/)

[Jouppi90] Jouppi, N. 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*, Seattle, WA, IEEE, 364-373, 1990.

[Jouppi94] Jouppi, N. and Wilton, S. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, IEEE Computer Society Press, 34-45, 1994.

[Kane92] Kane, G. and Heinrich, J. *MIPS RISC Architecture*. Prentice-Hall, Inc., 1992.

[Kessler91] Kessler, R. *Analysis of multi-megabyte secondary CPU cache memories*. Ph.D. dissertation, University of Wisconsin-Madison. 1991.

[Kessler92] Kessler, R. and Hill, M. Page placement algorithms for large real-indexed caches. *ACM Transaction on Computer Systems* **10** (4): 338-359, 1992.

[Khalidi93] Khalidi, Y. A. and Nelson, M. N. An implementation of UNIX on an object-oriented operating system. In *Proceedings of the Winter '93 USENIX Conference*, San Diego, California, USENIX, 469-480, 1993.

[Knuth73] Knuth, D. *Sorting and Searching from the Art of Computer Programming*. Reading, Massachusetts, Addison-Wesley, 1973.

[Koch94] Koch, P. Emulating the 68040 in the PowerPC Macintosh. *Presented at Microprocessor Forum*, San Francisco, CA, 1994.

[Koldinger92] Koldinger, E. J., Chase, J. S. and Eggers, S. J. Architectural support for single address space operating systems. *In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, ACM, 175-186, 1992.

[Kuck88] Kuck, D. The perfect club benchmarks: effective performance evaluation of supercomputers. The University of Illinois. CSRD Technical Report. 1988.

[Lacy88] Lacy, F. An address trace generator for trace-driven simulation of shared memory multiprocessors. *University of California at Berkeley. Technical Report UCB/CSD 88/407*. 1988.

[Laha88] Laha, S., Patel, J. and Iyer, R. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers* **37** (11): 1325-1336, 1988.

[Larus90] Larus, J. R. Abstract Execution: A technique for efficiently tracing programs. *University of Wisconsin-Madison Technical Report*. 1990.

[Larus91] Larus, J. SPIM S20: A MIPS R2000 Simulator. *University of Wisconsin-Madison Technical Report, Revision 9*. 1991.

[Larus93] Larus, J. R. Efficient program tracing. *IEEE Computer* May, 1993 : 52-60, 1993.

[Lebeck94] Lebeck, A. and Wood, D. Fast-Cache: A new abstraction for memory-system simulation. *University of Wisconsin - Madison Technical Report 1211*, 1994.

[Lebeck95] Lebeck, A. and Wood, D. Active Memory: A new abstraction for memory-system simulation. In *Proceedings of the 1995 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, May, 1995.

[Lee94] Lee, C.-C. A case study of a hardware-managed TLB in a multi-tasking environment. *University of Michigan Technical Report*. 1994.

[Levy80] Levy, H. M. and Jr., R. H. E. *Computer Programming and Architecture: The VAX-11*. Bedford, Mass., Digital Press, 1980.

[Lin88a] Lin, Y., Baer, J.-L. and Lazowska, E. *Parallel trace-driven simulation of multiprocessor cache performance: algorithms and analysis*. Department of Computer Science, University of Washington, Seattle, WA. 1988.

[Lin88b] Lin, Y., Baer, J.-L. and Lazowska, E. *Tailoring a parallel trace-driven simulation technique to specific multiprocessor cache coherence protocols*. Department of Computer Science, University of Washington, Seattle, WA. 1988.

[Lynch93] Lynch, W. L. The Interaction of Virtual Memory and Cache Memory. *Stanford University Technical Report CSL-TR-93-587*. 1993.

[MacWeek94] MacWeek Staff, Apple holds up 603 for cache, *MacWeek **8** (21): 1, 100***,*** May 24, 1994

[Magnusson93] Magnusson, P. *A design for efficient simulation of a multiprocessor.* In *Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS-93)*, La Jolla, California, 1993.

[Malan91] Malan, G., Rashid, R., Golub, D. and Baron, R. DOS as a Mach 3.0 application. In *Proceedings of the USENIX Mach Symposium*, USENIX, 27-40, 1991.

[Martonosi92] Martonosi, M., Gupta, A. and Anderson, T. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, ACM, 1992.

[Martonosi93] Martonosi, M., Gupta, A. and Anderson, T. Effectiveness of trace sampling for performance debugging tools. In *Proceedings of the 1993 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Santa

Clara, California, ACM, 248-259, 1993.

[Martonosi94] Martonosi, M. *Analyzing and tuning memory performance in sequential and parallel programs*. Ph.D. dissertation, Stanford University. 1994.

[Mattson70] Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal* **9** (2): 78-117, 1970.

[May87] May, C. Mimic: A fast S/370 simulator. In *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, ACM, 1-13, 1987.

[Maynard94] Maynard, A. M., Donnelly, C. and Olszewski, B. 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*, San Jose, CA, ACM, 145-156, 1994.

[McFarling89] McFarling, S. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, ACM, 183-191, 1989.

[McKusick84] McKusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems* **2** (3): 181-197, 1984.

[Milenkovic90] Milenkovic, M. Microprocessor memory management units. *IEEE Micro* **10** (2): 70-85, 1990.

[MIPS88] MIPS. *RISCompiler Languages Programmer's Guide*. MIPS, 1988.

[Mogul91] Mogul, J. C. and Borg, A. The effect of context switches on cache performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, ACM, 75-84, 1991.

[Motorola90a] Motorola. *MC88200 Cache/Memory Management Unit User's Manual*. Prentice Hall, 1990.

[Motorola90b] Motorola. *MC88100 RISC Microprocessor User's Manual*. Englewood Cliffs, NJ, Prentice Hall, 1990.

[Motorola93] Motorola. *PowerPC 601 RISC Microprocessor Users' Manual*. Motorola, Inc., 1993.

[MReport92] MReport. Sebastopol, CA, MicroDesign Resources, 1992.

[MReport93] MReport. Sebastopol, CA, MicroDesign Resources, 1993.

[MReport94] MReport. Sebastopol, CA, MicroDesign Resources, 1994.

[Mulder91] Mulder, J., Quach, N. and Flynn, M. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits* **26** (2): 98-106, 1991.

[Nagle92] Nagle, D., Uhlig, R. and Mudge, T. Monster: A tool for analyzing the interaction between operating systems and computer architectures. *University of Michigan Technical Report CSE-TR-147-92*. 1992.

[Nagle93] Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, IEEE, 27-38, 1993.

[Nagle94] Nagle, D., Uhlig, R., Mudge, T. and Sechrest, S. Optimal Allocation of On-chip Memory for Multiple-API Operating Systems. In *Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, 1994.

[Nielson91] Nielson, R. DOS on the dock. *NeXT World. 50-51*, 1991.

[Olukotun91] Olukotun, O. A., Mudge, T. N. and Brown, R. B. Implementing a cache for a high-performance GaAs microprocessor. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, 138-147, 1991.

[Olukotun92] Olukotun, K., Mudge, T. and Brown, R. Performance optimization of pipelined primary caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, IEEE, 181-190, 1992.

[Ousterhout89] Ousterhout, J. Why aren't operating systems getting faster as fast as hardware. *WRL Technical Note* (TN-11): 1989.

[Ousterhout94] Ousterhout, J. K. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[Palcharla94] Palcharla, S. and Kessler, R. E. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, Illinois, IEEE, 24-33, 1994.

[Patel92] Patel, K., Smith, B. C. and Rowe, L. A. Performance of a Software MPEG Video Decoder. *Technical Report, University of California, Berkeley.* 1992.

[Patterson80] Patterson, D. A. and Sequin, C. H. Design considerations for single-chip computers of the future. *IEEE Transactions on Computers* **C-29** (2): 108-116, 1980.

[Peterson90] Peterson, L., Hutchinson, N., O'Malley, S. and Rao, H. The x-kernel: A platform for accessing internet resources. *IEEE Computer* **23** (5): 23-33, 1990.

[Pierce94a] Pierce, J. and Mudge, T. IDtrace - A tracing tool for i486 simulation. *University of Michigan Technical Report CSE-TR-203-94*. 1994.

[Pierce94b] Pierce, J. and Mudge, T. IDtrace - A tracing tool for i486 simulation (extended abstract). In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 419-420, 1994.

[Pierce94c] Pierce, J. and Mudge, T. The effect of speculative execution on cache performance. In *Proceedings of the International Parallel Processing Symposium*, Cancun, Mexico, April, 1994.

[Pierce95] Pierce, J., Cache behavior in the presence of speculative execution - the benefits of misprediction, *Ph.D. dissertation, The University of Michigan*, 1995.

[Pleszkun94] Pleszkun, A. Techniques for compressing program address traces. *Technical Report, Department of Electrical and Computer Engineering, University of Colorado-Boulder.* 1994.

[Prieve74] Prieve, B. G. *A page partition replacement algorithm*. University of California at Berkeley. 1974.

[Przybylski89] Przybylski, S., Horowitz, M. and Hennessy, J. Characteristics of performance-optimal multi-level cache hierarchies. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, 114-121, 1989.

[Przybylski90] Przybylski, S. The performance impact of block sizes and fetching strategies. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Seattle, WA, IEEE, 160-169, 1990.

[Puzak85] Puzak, T. *Analysis of cache replacement algorithms*. Ph.D. dissertation, University of Massachusetts. 1985.

[Rashid88] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers* **37** (8): 896-908, 1988.

[Reinhardt93] Reinhardt, S., Hill, M., Larus, J., Lebeck, A., Lewis, J. and Wood, D. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, ACM, 48-60, 1993.

[Reinhardt94] Reinhardt, S., Larus, J., and Wood, D. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April, 1994.

[Romer94] Romer, T. H., Lee, D., Bershad, B. N. and Chen, J. B. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, Monterey, CA, 255-266, 1994.

[Rozier92] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaise, C., Langlois, S., Leonard, P. and Neuhauser, W. Overview of the Chorus distributed operating system. In *Proceedings of the Micro-kernels and Other Kernel Architectures Workshop*, Seattle, Washington, USENIX, 39-69, 1992.

[Samples89] Samples, A. Mache: no-loss trace compaction. In *Proceedings of 1989 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ACM, 89-97, 1989.

[Scheifler86] Scheifler, R. and Gettys, J. The X window system. *ACM Transactions on Graphics* **5** (2): 79-109, 1986.

[Schoinas94] Schoinas, I., Falsafi, B., Lebeck, A., Reinhardt, S., Larus, J., Wood, D. *Fine-grain access control for distributed shared memory.* In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, ACM Press, 1994.

[Short88] Short, R. and Levy, H. A simulation study of two-level caches. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, 81-88, 1988.

[Sites88] Sites, R. L. and Agarwal, A. Multiprocessor cache analysis with ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture,* Honolulu, Hawaii, IEEE, 186-195, 1988.

[Sites92] Sites, R., Chernoff, A., Kirk, M., Marks, M. and Robinson, S. Binary translation. *Digital Technical Journal* **4** (4): 137-152, 1992.

[Smith77] Smith, A. J. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering* **SE-3** (1): 94-101, 1977.

[Smith78] Smith, A. J. Sequential program prefetching in memory hierarchies. *IEEE Computer* **11** (12): 7-21, 1978.

[Smith82] Smith, A. J. Cache memories. *Computing Surveys* **14** (3): 473-530, 1982.

[Smith85] Smith, A. J. Cache evaluation and the impact on workload choice. In Proceedings of the 12th *International Symposium on Computer Architecture*, Boston, Mass., IEEE, 64-73, 1985.

[Smith86] Smith, A. Bibliography and readings on CPU cache memories and related topics. *Computer Architecture News* **14** : 22-42, 1986.

[Smith89] Smith, M. D., Johnson, M. and Horowitz, M. A. Limits on multiple instruction Issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, ACM, 290-302, 1989.

[Smith91] Smith, M. D. Tracing with pixie. *Technical Report, Stanford University,* Stanford, CA. 1991.

[Smith92] Smith, J. E. and Hsu, W.-C. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing '92*, 588-597, 1992.

[Smith94] Smith, J. E. and Weiss, S. PowerPC 601 and Alpha 21064: A tale of two RISCs. *IEEE Computer* **27** (6): 46-58, 1994.

[SPEC91] SPEC. The SPEC Benchmark Suite. *SPEC Newsletter.* **3:** 3-4, 1991.

[SPEC93] SPEC. SPEC: A five year retrospective. *The SPEC Newsletter* **5** (4): 1-4, 1993.

[Srivastava94] Srivastava, A. and Eustace, A. ATOM: A system for building customized program analysis tools. *DEC Western Research Lab. Technical Report TN-41*. 1994.

[Stephens91] Stephens, C., Cogswell, B., Heinlein, J., Palmer, G. and Shen, J. Instruction level profiling and evaluation of the IBM RS/6000. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, ACM, 180-189, 1991.

[Stolarchuk92] Stolarchuk, M. T. Faster AFS. *Center for Information Technology Integration Technical Report, 92-3*. Ann Arbor, MI, 1992.

[Stone93] Stone, H. *High-performance Computer Architecture*. Reading, Massachusetts, Addison-Wesley, 1993.

[Stunkel89] Stunkel, C. and Fuchs, W. TRAPEDS: producing traces for multicomputers via execution-driven simulation. In *Proceedings of the 1989 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Berkeley, CA, ACM, 70-78, 1989.

[Stunkel91] Stunkel, C., Janssens, B. and Fuchs, W. K. Collecting address traces from parallel computers. In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, Hawaii, 373-383, 1991.

[Srivastava94] Srivastava, A., Eustace, A., ATOM: A System for Building Customized Program Analysis Tools. *Digital Western Research Laboratory Research Report 94/2*, March, 1994.

[Sugumar93] Sugumar, R. *Multi-configuration simulation algorithms for the evaluation of computer designs*. Ph.D. dissertation, University of Michigan. 1993.

[Sun94] Sun Microsystems, Nested traps in UltraSPARC, *http://www.sun.com/stb/Processors/UltraSPARC/WhitePapers/NestedTraps/NestedTraps.html*. September, 1994.

[Talluri92] Talluri, M., Kong, S., Hill, M. D. and Patterson, D. A. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, 415-424, 1992.

[Talluri94] Talluri, M. and Hill, M. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, ACM, 1994.

[Taylor86] Taylor, G., Hilfinger, P., Larus, J., Patterson, D., Zorn, B., Evaluation of the SPUR lisp architecture. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 444-452, June 1986.

[Taylor90] Taylor, G., Davies, P. and Farmwald, M. The TLB slice - A low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 355-363, 1990.

[Tektronix94] Tektronix. *Test and Measurement Product Catalog*. Wilsonville, OR, 1994.

[Thekkath94] Thekkath, C. and Levy, H. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, ACM Press, 110-119, 1994.

[Thompson89] Thompson, J. and Smith, A. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems* **7** (1): 78-116, 1989.

[Torrellas92] Torrellas, J., Gupta, A. and Hennessy, J. Characterizing the caching and synchronization performance of multiprocessor operating system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, ADM, 162-174, 1992.

[Torrellas95] Torrellas, J., Xia, C. and Daigle, R. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the 1st International Syposium on High-Performance Computer Architecture*, Raleigh, North Carolina, to appear, 1995.

[Touma92] Touma, W. R. *The Dynamics of the Computer Industry.* Ph.D. dissertation, University of Texas at Austin. 1993.

[Uhlig92] Uhlig, R., Nagle, D., Mudge, T. and Sechrest, S. Software TLB management in OSF/1 and Mach 3.0. *University of Michigan Technical Report CSE-TR-156-93*. 1992.

[Uhlig94a] Uhlig, R. Kernel-based Memory Simulation (Extended Abstract). In *Proceedings of the 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, University of Michigan, 286-287, 1994.

[Uhlig94b] Uhlig, R., Nagle, D., Stanley, T., Sechrest, S., Mudge, T. and Brown, R. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems* (Fall): 1994.

[Uhlig94c] Uhlig, R., Nagle, D., Mudge, T. and Sechrest, S. Trap-driven simulation with Tapeworm II. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, ACM Press (SIGARCH), 132-144, 1994.

[Uhlig95] Uhlig, R., Nagle, D., Mudge, T. Sechrest, S., and Emer, J. Instruction Fetching: Coping with Code Bloat. To Appear In *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June, 1995.

[Upton94] Upton, M. D. *Architectural trade-offs in a latency tolerant gallium arsenide microprocessor.* Ph.D. Dissertation, The University of Michigan, 1994.

[Veenstra94] Veenstra, J. and Fowler, R. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication systems (MASCOTS)*, 201-207, 1994.

[Wada92] Wada, T., Rajan, S. and Przybylski, S. An analytical access time model for on-chip cache memories. *IEEE Journal of Solid-State Circuits* **27** (8): 1147-1156, 1992.

[Wall89] Wall, D. Link-time code modification. *DEC Western Research Lab Technical Report 89/17*. 1989.

[Wall92] Wall, D. Systems for late code modification. *DEC Western Research Lab. Technical Report92/3*. 1992.

[Wang89] Wang, W.-H., Baer, J.-L. and Levy, H. Organization and performance of a two-level virtual-real cache hierarchy. In *Proceedings of the 16th Annual International Symposium on Computer Archtecture*, Jerusalem, Isreal, IEEE, 140-148, 1989.

[Wang90] Wang, W.-H. and Baer, J.-L. Efficient trace-driven simulation methods for cache performance analysis. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, ACM, 27-36, 1990.

[Welch91] Welch, B. The file system belongs in the kernel. In *Proceedings of the USENIX Mach Symposium*, Monterey, California, USENIX, 233-249, 1991.

[Wiecek82] Wiecek, C. A. A case study of VAX-11 instruction set usage for compiler execution. In Proceedings of Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, 177-184, 1982.

[Wiecek92] Wiecek, C. A., Kaler, C. G., Fiorelli, S., Davenport, W. C. and Chen, R. C. *A* Model and Prototype of VMS Using the Mach 3.0 Kernel. In *Proceedings of the USENIX Micro-kernels and Other Kernel Architectures Workshop*, Seattle, Washington, USENIX, 187-203, 1992.

[Wilkes92] Wilkes, J. and Sears, B. A comparison of protection lookaside buffers and the PA-RISC protection architecture. *HP Laboratories Technical Report HPL-92-55*. 1992.

[Wilton94] Wilton, S. and Jouppi, N. An enhanced access and cycle time model for on-chip caches. *DEC Western Research Lab. 93/5*. 1994.

[Winsor89] Winsor, D. *Bus and cache memory organizations for multi-processors*. Ph.D. dissertation, The University of Michigan. 1989.

[Wood86] Wood, D., Eggers, S., Gibson, G., Hill, M., Pendleton, J., Ritchie, S., Taylor, G., Katz, R. and Patterson, D. An in-cache address translation mechanism. In

Proceedings of the 13th Annual Symposium on Computer Architecture, IEEE Computer Society Press, 358-365, 1986.

[Wood91] Wood, D., Hill, M. and Kessler, R. A model for estimating trace-sampled miss ratios. In *Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1991.

[Ziv76] Ziv, J. and Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* **23(1976)** : 75-81, 1976.

Abstract

Trap-driven Memory Simulation

by Richard Albert Uhlig

Chair: Professor Trevor Mudge

The use of *trace-driven simulation* in the evaluation of computer memory systems has been popular for at least 30 years. Despite considerable progress in the development of this method, the best trace-driven simulators still run at least one to two orders of magnitude slower than actual hardware and it is unlikely that they will exhibit further substantial improvements in speed.

Trap-driven simulation is a new method for memory-system simulation that overcomes the bottlenecks inherent in trace-driven simulation speed. By invoking a memory-system simulator only on references that cause a change in the simulated memory state, trap-driven simulators can potentially reduce memory-system slowdowns to zero. Although promising in principle, little is known about how trap-driven simulation compares, in practice, to trace-driven simulation.

Through a literature survey of trace-driven simulation and the implementation of a prototype trap-driven simulator, Tapeworm II, this dissertation compares the trace-driven and trap-driven methods on the basis of their flexibility, portability, speed, and accuracy. We show that trap-driven simulation offers clear advantages over trace-driven simulation with respect to speed and accuracy. With some exceptions, trap-driven simulation is nearly as flexible as trace-driven simulation, but suffers from certain problems of portability. We suggest methods for alleviating these problems, through simulation-host hardware support in existing and future systems.

Keywords: Trap-driven Simulation, Trace-driven Simulation, Memory-system Simulation, Cache, TLB