Trace-Driven Memory Simulation: A Survey

RICHARD A. UHLIG

Intel Microcomputer Research Lab, Hillsboro, OR

TREVOR N. MUDGE

University of Michigan, Ann Arbor, MI

As the gap between processor and memory speeds continues to widen, methods for evaluating memory system designs before they are implemented in hardware are becoming increasingly important. One such method, trace-driven memory simulation, has been the subject of intense interest among researchers and has, as a result, enjoyed rapid development and substantial improvements during the past decade. This article surveys and analyzes these developments by establishing criteria for evaluating trace-driven methods, and then applies these criteria to describe, categorize, and compare over 50 trace-driven simulation tools. We discuss the strengths and weaknesses of different approaches and show that no single method is best when all criteria, including accuracy, speed, memory, flexibility, portability, expense, and ease of use are considered. In a concluding section, we examine fundamental limitations to trace-driven simulation, and survey some recent developments in memory simulation that may overcome these bottlenecks.

Categories and Subject Descriptors: B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids—*simulation*; C.4 [**Computer Systems Organization**]: Performance of Systems—*measurement techniques*; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms: Experimentation, measurement

Additional Key Words and Phrases: Caches, memory management, memory simulation, TLBs, trace-driven simulation

1. INTRODUCTION

It is well known that the increasing gap between processor and main memory speeds is one of the primary bottlenecks to good overall computer system performance. The traditional solution to this problem is to build small fast memories (caches) to hold recently used data and instructions close to the processor for quicker access [Smith 1982]. During the past decade, microprocessor clock rates have increased at a rate of 40% per year, while main memory (DRAM)

This work was supported by ARPA contract DAAH04-94-G-0327, by NSF contract CISE9121887, by an NSF Graduate Fellowship, and by a European Research Consortium for Informatics and Mathematics (ERCIM) Postgraduate Fellowship.

Authors Addresses: R. A. Uhlig, Microcomputer Research Lab, Intel Corp., 2111 N.E. 25th Ave., Hillsboro, OR 97124-5961; T. N. Mudge, Advanced Computer Architecture Lab (ACAL), Electrical Engineering and Computer Science Department, University of Michigan, 1301 Beal Avenue, Ann Arbor, Michigan 48109-2122.

speeds have increased at a rate of only about 11% per year [Upton 1994]. This trend has made modern computer systems increasingly dependent on caches. A case in point: disabling the cache of the VAX 11/780, a machine introduced in the late 1970s, would have increased its workload runtimes by a factor of only 1.6 [Jouppi 1990], whereas disabling the cache of the HP 9000/735, a more recent machine introduced in the early 1990s, would cause workloads to slow by a factor of 15 [Upton 1994].

It is clear that these trends are making overall system performance highly sensitive to even minor adjustments in cache designs. As a result, memory system designers are becoming increasingly dependent on methods for evaluating design options before having to commit them to actual implementation. One such method is to write a program that simulates the behavior of a proposed memory system design, and then to apply a sequence of memory references to the simulation model to mimic the way that a real processor might exercise the design. The sequence of memory references is called an *address trace*, and the method is called *trace-driven memory simulation*. Although conceptually simple, a number of factors make trace-driven simulation difficult in practice. Collecting a complete and detailed address trace may be hard, especially if it is to represent a complex workload consisting of multiple processes, the operating system, and dynamically linked or dynamically compiled code. Another practical problem is that address traces are typically very large, potentially consuming gigabytes of storage space. Finally, processing a trace to simulate the performance of a hypothetical memory design is a time-consuming task.

During the past 10 years, researchers working on these problems have made a number of important advances in *trace collection, trace reduction,* and *trace processing*. This survey documents these developments by defining various criteria for judging and comparing these different components of trace-driven simulation. We consider accuracy, speed, memory usage, flexibility, portability, expense, and ease of use in an analysis and comparison of over 50 actual implementations of recent trace-driven simulation tools. We discuss which methods are best under which circumstances, and comment on fundamental limitations to trace-driven simulation in general. Finally, we conclude this survey with a description of recent developments in memory system simulation that may overcome fundamental bottlenecks to strict trace-driven simulation.

2. SCOPE, RELATED SURVEYS, AND ORGANIZATION

Trace-driven simulation has been used to evaluate memory systems for decades. In his survey of cache memories, A. J. Smith [1982] gives examples of trace-driven memory system studies that date as far back as 1966, and several surveys of trace-driven techniques have been written since then [Holliday 1991; Kaeli 1991; Stunkel et al. 1991; Cmelik and Keppel 1994]. Holliday [1991] examined the topic for uniprocessor and multiprocessor memory system design and Stunkel et al. [1991] studied trace-driven simulation in the specific context of multiprocessor design. Pierce et al. [1995] surveyed one aspect of trace collection based on static code annotation techniques, and Cmelik and Keppel [1994] surveyed trace collectors based on code emulation.

This survey distinguishes itself from the others in that it is more up to date, and in its scope. Numerous developments in trace-driven simulation during the past five years warrant a new survey of tools and methods that have not been previously reviewed. This survey is broader in scope than the surveys by Pierce et al. and Cmelik and Keppel, in that it considers all aspects of trace-driven simulation, from trace collection and trace reduction to trace processing. On the other hand, its scope is more limited, yet more detailed, than the surveys by Holliday and Stunkel et al. in that it focuses mainly on uniprocessor memory simulation, but pays greater attention to tools capable of tracing multiprocess workloads and the operating system.

We do not examine analytical methods for predicting memory system performance. A good starting point for study of these techniques is Agarwal et al. [1989]. Although trace-driven methods have been successfully applied to other domains of computer architecture, such as the simulation of superscalar processor architecture, or the design of I/O systems, this survey focuses on trace-driven *memory system* simulation only. Memory performance can also be measured with hardware-based counters that keep track of events such as cache misses in a running system. Although useful for determining the memory performance of an existing machine, such counters are unable to predict the performance of hypothetical memory designs. We do not study them here, but several examples can be found in Emer and Clark [1984], Clark et al. [1985], IBM [1990], Nagle et al. [1992], Digital [1992], and Cvetanovic and Bhandarkar [1994].

We begin this survey by establishing several general criteria for evaluating trace-driven simulation tools in Section 3. Sections 4 through 7 examine the different stages of trace-driven simulation, and Section 8 studies some new methods for memory simulation that extend beyond the traditional trace-driven paradigm. Section 9 concludes the survey with a summary.

This survey makes frequent use of tables to summarize the key features, performance characteristics, and original references for each of the trace-driven simulation tools discussed in the main body of text. This organization enables the reader to approach the material at several levels of detail. We suggest a reading of Section 3, the opening paragraphs of Sections 4 through 7, and an examination of each of the accompanying tables to obtain a good cursory introduction to the field. A reader desiring further information can then read the remainder of the body text in greater detail. The original papers themselves, of course, offer the greatest level of detail, and their references can be found quickly in the summary tables and the bibliography at the end of the survey.

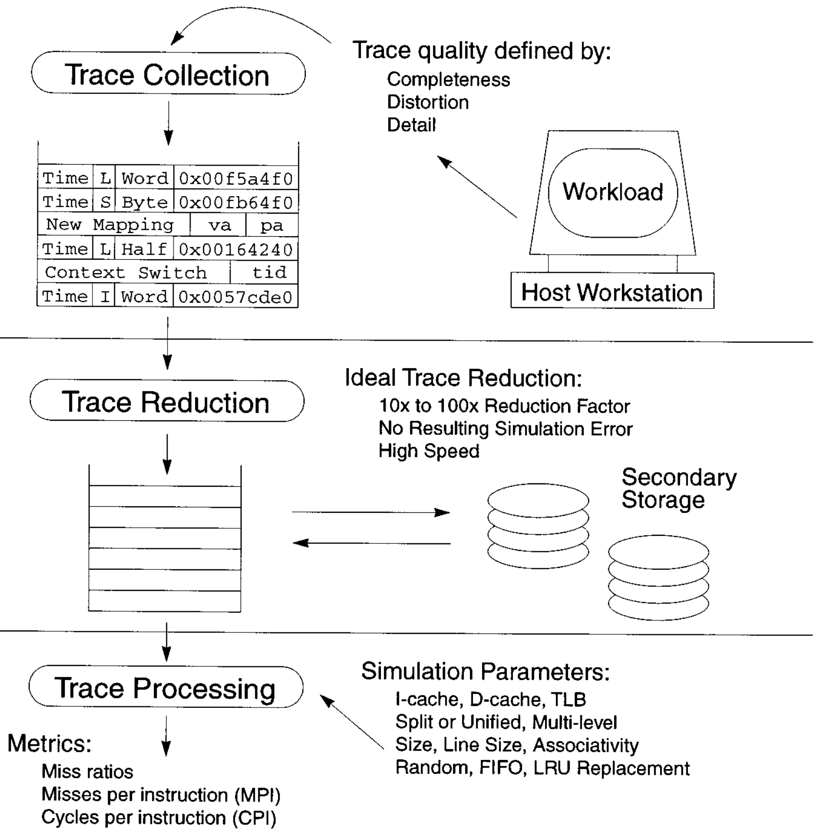
3. GENERAL EVALUATION CRITERIA AND METRICS

A trace-driven memory simulation is sometimes viewed as consisting of three main stages: *trace collection, trace reduction*, and *trace processing* [Holliday 1991] (see Figure 1). *Trace collection* is the process of determining the exact sequence of memory references made by some workload of interest. Because the resulting address traces can be very large, *trace-reduction* techniques are often used to remove unneeded or redundant data from a full address trace. In the final stage, *trace processing*, the trace is fed to a program that simulates the behavior of a hypothetical memory system. To form a complete trace-driven simulation system, 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 Sections 3 through 7, we examine each of the preceding components in greater detail, but it is helpful to define, at the outset, some general criteria for judging and comparing different trace-driven simulation tools.[1] Perhaps the most important criterion is *accuracy*, which we loosely define in terms of percent error in some performance metric

---

[1] Some evaluation criteria apply to only a specific stage of trace-driven simulation, so we cover them in future sections where the details are more relevant.



Time	L	Word	0x00f5a4f0
Time	S	Byte	0x00fb64f0
New Mapping		va	pa
Time	L	Half	0x00164240
Context Switch			tid
Time	I	Word	0x0057cde0

Figure 1. Three stages of trace-driven simulation.

such as miss ratio or misses per instruction:

$$\text{Error} = \left[\frac{\begin{array}{c}\text{(True Performance}\\ -\text{ Simulated Performance)}\end{array}}{\text{(True Performance)}}\right] \cdot 100\%. \quad (1)$$

Error is often difficult to determine in practice because true performance may not be known, or because it may vary from run to run of a given workload. Furthermore, accuracy is affected by many factors, such as the "representativeness" of the chosen workload, the quality of the collected address trace, the way that the trace is reduced, and the level of detail modeled by the trace-driven memory simulator. Although it may be difficult to determine from which of these factors some component of error originates, it is important to understand the nature of these errors, and how they can be minimized.

Ideally, a workload suite should be selected in a way that represents the environment in which the memory system is expected to perform. The memory system might be intended for commercial applications (database, spreadsheet, etc.), for engineering applications (computer-aided design, circuit simulation, etc.), for embedded applications (e.g., a postscript interpreter in a laser printer), or for some other purpose. Studies have shown that the differences between these types of workloads is substantial [Gee et al. 1993; Maynard et al. 1994; Uhlig et al. 1995; Romer et al. 1996], so good workload selection is crucial—even the most perfect trace acquisition and simulation tools cannot overcome the bias in predicted performance that results if this stage of the process is not executed with care.

We explore, in the next section, some reasons why a collected trace might differ from the actual stream of memory references generated by a workload, but it is easy to see at this point in the discussion why differences are important. Many trace-collection tools exclude, for example, memory references made by the operating system. Excluding the OS, which may constitute a large fraction of a workload's activity, is bound to affect simulation results [Chen and Bershad 1993; Nagle et al. 1993; Nagle et al. 1994].

When we look at trace reduction in Section 5 we see that some methods achieve higher degrees of reduction at the expense of lost trace information. When this happens, we can use a modified form of Equation (1) to measure the effects:

$$\text{Error} = \left[\frac{\begin{array}{c}\text{(Measurements with Full Trace}\\ \text{—Measurements with Reduced Trace)}\end{array}}{\text{(Measurements with Full Trace)}} \right] \cdot 100\%. \qquad (2)$$

Errors can also come from the final, trace-processing stage, where a memory system's behavior is simulated. Such errors arise whenever the simulator fails to model the precise behavior of the design under study, a task that is becoming increasingly difficult as processors move to memory systems that support features such as prefetching and nonblocking caches.

A second criterion by which each of the stages of trace-driven simulation can be evaluated is *speed*. The rate per second 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 different hardware platforms. 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 older or 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 or on which they are processed. Depending on the context, we compute slowdowns in a variety of ways:

$$\text{Slowdown} = \frac{\text{Address Collection Rate}}{\begin{array}{c}\text{Host System Address}\\ \text{Generation Rate}\end{array}}, \qquad (3)$$

$$\text{Slowdown} = \frac{\text{Address Processing Rate}}{\begin{array}{c}\text{Host System Address}\\ \text{Generation Rate}\end{array}}, \qquad (4)$$

$$\text{Slowdown} = \frac{\text{Total Simulation Time}}{\begin{array}{c}\text{Normal Host System}\\ \text{Execution Time}\end{array}}. \qquad (5)$$

Because each of these definitions divides by the speed of the host hardware, they enable an approximate comparison of two methods implemented on different hosts.

Some of the trace-driven simulation techniques that we examine can reduce overall slowdowns. We report their effectiveness in terms of *speedups*, which divide slowdowns to obtain overall slowdowns:

$$\text{Overall Slowdown} = \frac{\text{Slowdown}}{\text{Speedup}}. \qquad (6)$$

A third general evaluation criterion is the amount of extra memory used by a tool. Depending on the circumstances, memory can refer to secondary storage (disk or tape), as well as primary storage (main memory). As with speed, it is often not meaningful to report memory usage in terms of bytes because different workloads running on different hosts may begin with substantially different memory requirements. Therefore, whenever possible, we report memory usage as an expansion factor or *overhead* based on the usual memory required by the workload running on the host machine:

$$\text{Memory Overhead} = \frac{\text{Additional Memory Required}}{\text{Normal Host Memory Required}}. \tag{7}$$

Additional memory can be required at each stage. Some trace-collection methods annotate or emulate workloads, causing them to expand in size, some trace-processors use complex data structures that are memory intensive, and trace interfaces use additional memory to buffer trace data as they pass from stage to stage. The purpose of the second stage, trace reduction, is to reduce these memory requirements. We measure the effectiveness of trace reduction in terms of a memory *reduction factor*:

$$\text{Reduction Factor} = \frac{\text{Full Address Trace Size}}{\text{Reduced Address Trace Size}}. \tag{8}$$

In addition to *accuracy, speed*, and *memory*, there are other general evaluation criteria that recur throughout this survey. A tool has high *portability* if it is easy to re-implement it on different host hardware. It has *flexibility* if it is able to be used for the simulation of a wide range of memory parameters (cache size, line size, associativity, replacement policy, etc.) and for collecting a broad range of performance metrics (miss ratio, misses per instruction, cycles per instruction, etc.). By *expense* we mean the cost of any hardware or special monitoring equipment required solely for the purposes of conducting simulations. Finally, *ease of use* refers to the amount of effort required of the end-user to learn and to operate the trace-driven simulator once it has been developed.

4. TRACE COLLECTION

To ensure accurate simulations, collected address traces should be as close as possible to the actual stream of memory references made by a workload when running on a real system. Trace quality can be evaluated based on the *completeness* and *detail* in a trace, or on the degree of *distortion* that it contains. A *complete* trace includes all memory references made by each component of the system, including all user-level processes and the operating system kernel. User-level processes include not only applications, but also OS server and daemon processes that provide services such as a file system or network access. Complete traces should also include dynamically compiled or dynamically linked code, which is becoming increasingly important in applications such as processor or operating system emulation [Nagle et al. 1994; Cmelik and Keppel 1994]. 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 processes, and tags that mark each address with a reference type (read, write, execute), size (word, halfword, byte), and a timestamp. Traces should be *undistorted* so that they do not include any additional memory references, or references that appear out of order relative to the actual reference stream of the workload had it not been monitored. Common forms of distortion include *trace discontinuities*, which occur when tracing must stop because a trace buffer is not large enough to continue recording workload memory references, and *time dilation* and *memory dilation*, which occur when the tracing method causes a monitored workload to run slower, or to consume more memory than it normally would.

In addition to the three aspects of trace quality described, a good trace collector exhibits 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 have been extracted at virtually every system level, from the circuit and microcode levels to the com-

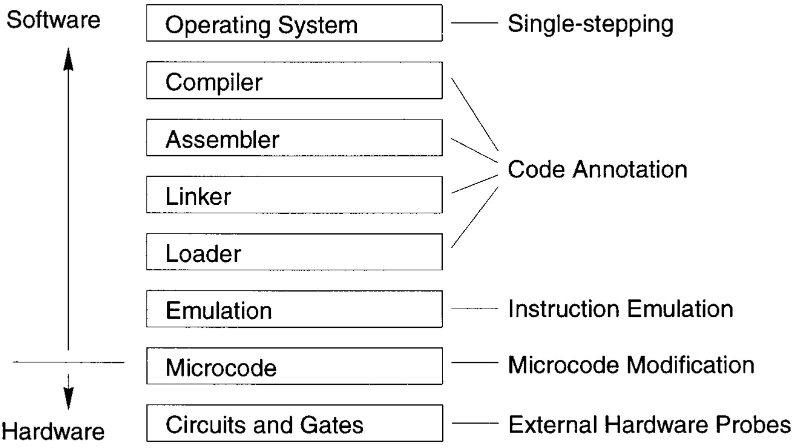


Figure 2. Levels of system abstraction and trace collection methods.

piler and operating system levels. (see Figure 2). We organize the remainder of this section accordingly, starting at the lower hardware levels.

4.1 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 host computer while it runs a workload. The address and control signals are fed into an external memory buffer at the full speed of the monitored host system, and when the buffer fills, its contents are transferred 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 host whenever the buffer becomes full. It is usually only possible to stall the processor—external I/O devices, such as disks or network controllers usually must be permitted to continue operating. If there is no way to stall the system, then several discontinuous address-trace samples can be acquired and concatenated. In either case, the resulting trace exhibits a form of distortion that we call *trace discontinuity*. Table I summarizes several probe-based trace collectors recently described in the literature. We discuss each in greater detail in the following.

Most commercial logic analyzers provide the necessary hardware to construct a probe-based trace collector [Tektronix 1994; Hewlett-Packard 1991]. Alexander et al. [1985, 1986] connected a logic analyzer to a National Semiconductor 32016-based workstation running Genix to collect address traces for TLB and cache simulation. The small size of the trace buffer (4096 entries of 32 bits each) necessitated the design of circuitry to place the processor in a stalled state while the buffer was unloaded to a secondary storage device. A similar approach was used in the Monster monitoring system by a group including the authors of this survey [Nagle et al. 1992]. Monster consists of a DAS 9200 logic analyzer connected to an R2000-based DECstation 3100. The operating system kernel was modified to stall the machine in a software loop, avoiding the need for any additional stalling hardware. Some logic analyzers provide interchangeable probes to support multiple architectures. The DAS 9200, for example, has probe modules for most popular microprocessors, a

Table I. External Probe-Based Trace Collectors

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

flexibility that Fuentes [1993] exploited to collect addresses from both Alpha- and Pentium-based workstations.

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 either in frequent processor stalls or smaller trace samples, and thus greater trace distortion due to discontinuities. Special-purpose hardware with very large, high-speed memories has been built to treat this problem. Biomation Corporation [1991] builds a trace-collection system with 80 million trace buffer entries. The trace collector described in Happel and Jayasumana [1992] has a 40 Mbyte trace buffer, large enough to hold 8 million memory references at a time. The Magellan Trace Machine (MTM) has a buffer that can hold 33 million bus transactions [Fuentes 1993], and recent versions of the Bach system use similarly large buffers [Flanagan 1994]. Bach offers the additional advantage that it supports monitoring of at least three different microprocessor architectures (i486, 68030, and SPARC).

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. Examples of probe-based trace collectors that are limited in this way are described in Torrellas et al. [1992] and Fuentes [1993]. 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. This solution can, however, 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 (i.e., in a way that forces *all* references off-chip) [Digital 1992; Fuentes 1993]. Although full address traces are desirable, a trace of just cache misses is by no means worthless. As we see in Section 5 on trace reduction, such a trace can still be used to simulate other cache configurations, albeit subject to certain restrictions.

The main advantage of all the probe-based trace collectors previously described is their ability to capture trace sequences complete with both user and kernel memory references, and free of most forms of trace distortion, provided that 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, halfword, byte) of the memory references made by a monitored processor. In processors that implement hardware prefetching or speculative execution, it may be difficult or impossible to separate "true" memory references from those that occur due to a prefetch that might not actually be used. Some of these problems can be overcome by implementing the inverse function of the processor sequencer, either in the trace-collecting hardware, or in a trace post-processing tool [Flanagan 1994; Nagle et al. 1992]. Because the addresses captured by a probe-based monitor are usually physical addresses, special methods that may require cooperation from the host OS must be used to reverse-translate addresses to their matching virtual addresses [Grimsrud 1993]. For similar reasons, it is often difficult to relate a given memory reference to the process that made it without assistance from a modified OS kernel that emits *trace markers* or other annotations as clues [Torrellas et al. 1992; Nagle et al. 1992; Fuentes 1993]. 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.

A common misconception regarding trace collection using hardware probes is that the technique is very fast. Although it is true that acquisition of the trace proceeds at the full speed of the monitored system, it is important to account for the overhead of managing trace-buffer overflow as well as the time required to empty the buffer. This overhead is typically not reported in published papers, but because most systems can unload these buffers only through some form of relatively low bandwidth channel (see Table I), this overhead is necessarily high. For a system where overhead data are available (Monster), approximately 12 hours are required to obtain 11 seconds of real-time system activity. Fuentes [1993] 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. The overhead from both these systems comes from moving trace-buffer data over an Ethernet to a machine with SCSI-connected disks, and represents effective slowdowns of more than a thousand times relative to the speed of the unmonitored host. Most of the other systems listed in Table I use similar or even lower bandwidth interconnect to the trace buffer, so their overheads are comparable or higher. Although trace collection with hardware probes is time consuming, once the traces have been captured and stored to a permanent file they require no special hardware to use,[2] and can be used repeatedly to achieve reproducible simulation results.

Hardware probe-based methods share other common disadvantages. The first is expense. Logic analyzers with deep trace memories cost from $50,000 to $200,000 [Tektronix 1994; Hewlett-Packard 1991]. These amounts are probably low compared to the engineering costs associated with designing custom hardware as in Flanagan et al. [1992] or Torrellas [1992]. A second problem is portability. Although logic analyzers like the DAS 9200 support probes for most popular microprocessors, it is often necessary to physically modify the motherboard or chassis of the monitored system to enable probe access to the signals of interest [Nagle et al. 1992; Fuentes 1994]. These systems also require an understanding of the electrical issues concerning the connection of probes to running hardware, and are therefore typically fragile, sensitive to their operating environment, and difficult to learn and operate.

As previously noted, 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 therefore depends mainly on the level of support designed into systems for this task. A small, on-chip trace buffer that traps to the operating system kernel whenever it becomes full is an example of the sort

---

[2] The Monster traces, complete with trace-interpreting tools, are available to the general research community and can be obtained by contacting the authors of this survey.

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 [Nagle et al. 1994] and thus would be relatively costly in terms of chip area. An alternative approach would be to send certain key internal signals through the microprocessor package pins so that they could be monitored externally. We are not aware of any existing microprocessor that includes documented monitoring support of this type.

4.2 Microcode Modification

The high cost of circuit-level probing has motivated many researchers to develop methods for collecting traces at higher levels of system abstraction. One such alternative is to collect traces at the borderline between the hardware and software levels of a system in microcode (see Figure 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 [Wilkes 1969]. When implemented off-chip, a microcode memory was often writable or could be modified through replacement, making it possible to change the behavior of instructions, or to support multiple instruction sets. Agarwal realized that this mechanism made it possible to collect address traces [Agarwal et al. 1986, 1988]. 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 called *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 no buffer unloading overhead as with external probe-based trace collection. Finally, 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 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 could be increased only up to a certain point because it took away from the usable memory of the host system. Agarwal has developed a method, called *trace stitching*, to counter this problem [Agarwal 1989]. 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 devices such as disks and network controllers appear to the workload to be faster than they actually are, and interrupts from the system clock occur more frequently, thus changing the workload's behavior.

The primary disadvantage of the microcode-modification technique is that the technique is now effectively obsolete because most new microprocessors use hardwired control or have an on-chip microcode memory that is not easily modified. The fundamental idea behind microcode modification—augmenting the interpretation of instructions to generate trace addresses as a side effect of their execution—can, however, be implemented at other levels in a system. This has been made easier by some of the very trends that have made microcode modification obsolete. Hardwired

Table II. Instruction-Set Emulators that Support Trace Collection

Method	Reference	Name	Target(s)	Host(s)	Other Characteristics			Slowdown
					Register State Held in	Predecode / Translation Policy	Chain, Thread or Block	
Iterative Interpretation	[Cmelik93]	Spa (Spy)	SPARC	SPARC	Host Registers	N/A	No	40 - 600
	[Davies94]	Mable	MIPS-I, MIPS-III	MIPS-I	Memory	N/A	No	20 - 200
Predecode Interpretation	[Larus91}	SPIM	MIPS-I	SPARC, 680x0, MIPS, x86, HP-PA	Memory	All-at-once	No	25
	[Magnusson93]	gsim	88100	HP-PA, SPARC	Memory	Lazy	Threading	45 - 75
	[Bedicheck95]	Talisman	88100	SPARC	Memory	Lazy	Threading	100 - 150
	{Veenstra94]	MINT	R3000	R3000	Hybrid	All-at-once	Block	20 - 70
Dynamic Translation	[Cmelik94]	Shade	SPARC-V8, SPARC-V9, MIPS	SPARC-V8	Memory	Lazy	Chaining	9 - 14

control, for example, has been made possible (or at least easier) with the advent of RISC instruction sets [Hennessy and Patterson 1996]. The relatively simple and uniform coding of RISC instruction sets has also made it easier to develop fast instruction-set emulators and binary-rewriting tools for annotating executables to produce traces as a side effect of their normal execution. We examine these tools in the following sections on *instruction-set emulation* and *code annotation*.

4.3 Instruction-Set Emulation

An instruction-set architecture (ISA) is the collection of instructions that defines the interface between hardware and software for a particular computer system. A microcode engine, as described in the previous section, 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 [Sites et al. 1992]. 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.[3] The degree of slowdown is clearly related to the level of emulation detail. For some applications, such as the verification of a processor's logic design, the simulation detail required is very high and the corresponding slowdowns may agree with those cited. In the context of this review, however, we consider an instruction-set emulator to be sufficiently detailed for the purposes of address-trace collection if it can produce an accessible trace of memory references made by the instructions that it emulates. Given this minimal requirement, there are several recent examples of instruction-set emulators that have achieved slowdowns much lower than 1,000 (see Table II).

Spa [Cmelik and Keppel 1993] and Mable [Davies et al. 1994] are examples of emulators that use straightforward iterative interpretation (see top of Figure 3); they work by fetching, decoding, and then dispatching instructions one at a time in an iterative emulation loop, re-interpreting instructions each time they are encountered. Instructions are fetched by reading the contents of the emulated program's text segment, and are decoded through a series of mask and shift operations to extract the various fields of the instruction (opcode, register, specifiers, etc.). Once an in-

---

[3] Please see Agarwal [1989], Wall [1989], Borg et al. [1989], Stunkel et al. [1991], and Flanagan et al. [1992].

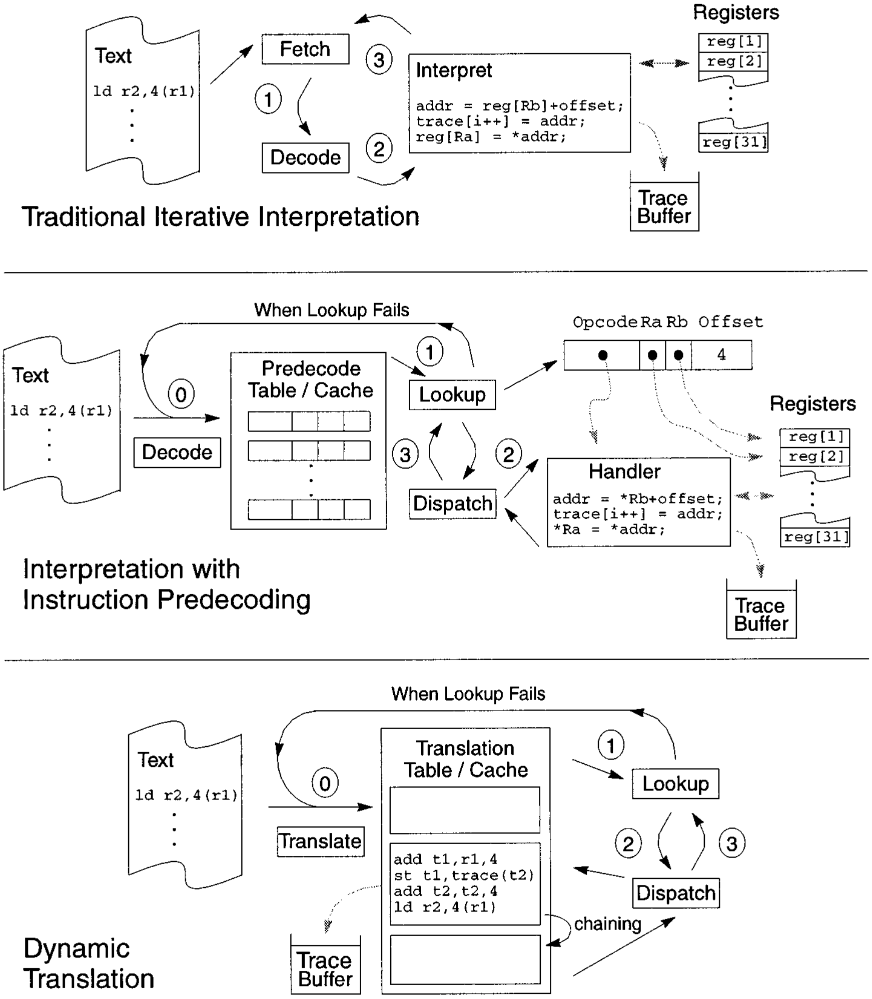


Figure 3. Some emulation methods.

struction has been decoded, it is emulated (*dispatched*) by updating the machine state, such as the emulated register set, which can be stored in memory as a *virtual register* data structure (as in Mable), or which may be held in the actual hardware registers of the host machine (as is done for part of the register set in Spa). An iterative interpreter may use some special features of the host machine to speed instruction dispatch,[4] but this final step is more commonly preformed by simply jumping to a small subroutine or *handler* that updates machine state as dictated by the instruction's semantics (e.g., updating a register with the results of an add or load operation). The reported slow-

---

[4] Spa, for example, exploits an artifact of the SPARC architecture called delayed branching. Spa issues two branch instructions immediately next to each other, with the second falling in the delay slot of the first. The first branch is to the instruction to be emulated, and the second branch is back to the interpreter. This technique enables Spa to "emulate" the instructions from a program's text segment via direct execution, while at the same time allowing the interpreter loop to maintain control of execution.

downs for iterative emulators such as Spa and Mable range from 20 to about 600, but these figures should be interpreted carefully because larger slowdowns may represent the time required to emulate processor activity that is not strictly required to generate address traces. The range of Mable slowdowns, for example, includes the additional time to simulate the pipeline of a dual-issue superscalar processor.

Some interpreters avoid the cost of repeatedly decoding instructions by saving *predecoded* instructions in a special table or cache (see middle of Figure 3). A predecoded instruction typically includes a pointer to the handler for the instruction, as well as pointers to the memory locations that represent the registers on which the instruction operates. The register pointers save both decoding time as well as time in the instruction handler, because fewer instructions are required to compute the memory address of a virtual register. An example of such an emulator is SPIM, which reads and translates a MIPS-I executable, in its entirety, to an intermediate representation understood by the emulation engine [Larus 1991]. After translation, SPIM can look up and emulate predecoded instructions with a slowdown factor of approximately 25. Talisman [Bedichek 1995] and gsim [Magnusson 1993] also use a form of instruction predecoding, but instead of decoding all instructions of a workload before it begins running, these emulators predecode instructions lazily, as they are executed for the first time. By caching the results, these emulators can benefit from predecoding without the initial start-up delay exhibited by SPIM. Both Talisman and gsim implement a further optimization, called *code threading*, in which the handler for one instruction directly invokes the handler for the subsequent instruction, without having to pass through the dispatch loop. The slowdowns of Talisman and gsim are higher than those of SPIM, but it should be noted that they are complete system simulators that model caches and memory management units, as well as I/O devices. MINT, a trace generator for shared-memory multiprocessor simulation, also uses a form of predecoded interpretation in which a handler for sequential blocks of code that do not contain memory references or branches are formed in native host code, which can then be quickly dispatched via a function pointer [Veenstra and Fowler 1994]. Veenstra reports slowdowns for MINT in the range of 20 to 70 for emulation of a single processor, which is comparable to the slowdowns of SPIM.

Shade takes instruction decoding a step further by dynamically compiling target instructions into equivalent sequences of host instructions [Cmelik and Keppel 1994]. As each instruction is referenced for the first time, Shade compiles it into an efficient sequence of native instructions that run directly on the host machine (see bottom of Figure 3). Shade records compiled sequences of native code in a lookup table, which is checked by its core emulation loop each time it dispatches a new instruction. If a compiled translation already exists, it is found through the lookup mechanism and the code sequence need not be recompiled. Like gsim and Talisman, Shade's compile-and-cache method enables it to translate source instructions lazily, only as needed. Shade implements an optimization similar to code threading, in which two consecutive translations are *chained* together so that the end of one translation can directly invoke the beginning of the next translation, without having to return to the core emulation loop. Shade supports 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 as addresses generated by the previous instruction, so that memory simulations are possible. The slowdowns reported in Table II are for Shade emulations that generate a trace of both instruction and data addresses, which are then passed to a *null* analyzer that does not add overhead to the emulation process. The resulting slowdowns (9 to 14) are therefore a good estimate of the minimal slowdown for emulator-generated address traces and demonstrate that fast emulators can, indeed, be used effectively for this task.

All of these emulators collect references from only a single process and exclude kernel references, so they are limited with respect to trace completeness. Some of these tools claim to support multithreaded applications and emulation of operating system code, but this statement should be interpreted carefully. All of these emulators run in their own user-level process and require the full support of a host operating system. Within this process, they may emulate certain operating system functions by intercepting system calls and passing them on to the host OS, but this does not mean that they are able to monitor the address references made by the actual host OS, nor are they able to see any references made by any other user-level processes in the host system. An important advantage of dynamic emulation is that it can be made to handle dynamically compiled and dynamically linked code (Shade is an example). With respect to trace detail, instruction-set emulation naturally produces virtual addresses, and is generally unable to determine the actual physical addresses to which these virtual addresses correspond.

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 [Larus 1991]. Tools that only predecode target instructions are likely to be more portable than those that actually compile code that executes directly on the host. Shade has been used to simulate several target architectures, one of which (SPARC-V9) had yet to be implemented at the time the article was written [Cmelik and Keppel 1993, 1994]. 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. Shade analyzers, for example, can specify that only load data addresses in a specific address range should be traced [Cmelik and Keppel 1994]. 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 annotation 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, can use an order of magnitude more memory than equivalent native code [Cmelik and Keppel 1994]. 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 II, but for Shade they are reported to be in the range of 4 to 40 times the usual memory required by normal native execution [Cmelik and Keppel 1993, 1994]. Increased memory usage means that these systems must be equipped with additional physical memory to handle large workloads.

4.4 Static Code Annotation

The fastest instruction-set emulators *dynamically* translate instructions in the target ISA to instructions in the host ISA, and optionally annotate 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. This additional flexibility comes at some cost, both in overall execution slowdown and in memory usage. For the purposes of

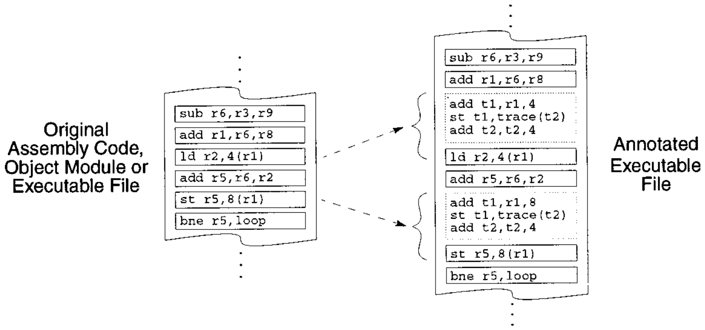


Figure 4. Static code annotation.

trace collection, it is often acceptable to trade some flexibility for increased speed. If the target and host ISAs are the same and if dynamically changing code is not of interest, then a workload can be annotated *statically*, before run-time. With this technique, instructions are inserted around memory operations in a workload to create a new executable file that deposits a stream of memory references into a trace buffer as the workload executes (see Figure 4). Static code annotation can be performed at the source (assembly) level, the object-module level, or the executable (binary) level (see Figure 2 and Table III), with different consequences for both the implementation and the end-user [Stunkel et al. 1991; Wall 1992; Pierce and Mudge 1994].

The main advantage of annotating code at the source level is ease of implementation. At this level, the task of relocating the code and data of the annotated 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 annotation points. Unfortunately, annotation at this level may render the tool unusable in many situations because the complete source code for a workload of interest is often not available. An early example of code annotation performed at the source level is the TRAPEDS system [Stunkel and Fuchs 1989]. 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. Some other tools take greater advantage of the additional information about program structure available at the source level. Both MPtrace [Eggers et al. 1990] and AE [Larus 1990] use control-flow analysis to annotate programs in a minimal way so that they produce a trace of only significant dynamic events. AE, for example, analyzes a program to find those instructions that contribute to address calculations. It then determines which addresses are easy to reconstruct, and which addresses depend on values that are difficult or impossible to determine through static analysis. Larus gives an example annotation of a simple subroutine that initializes 100 elements in an array structure starting from a location specified as a parameter to the procedure. The starting address is a value that cannot be known statically, so it is considered to be a *significant event*, and the program is annotated to emit this value to a trace file. The remaining addresses, however, can be easily reconstructed later, given the starting address and a description of the striding pattern through the array, which AE specifies in a program *schema*. Given a trace of significant events, along with the program schema, Larus describes how to construct a postprocessing program that reconstructs the full trace.

Table III. Static Code Annotators

Method	Reference	Name	Slowdown	Time Dilation	Memory Dilation	Completeness		Processor	Analyzer Interface
						Multi-process	OS Kernel		
Source	[Stunkel89]	TRAPEDS	20 - 30	20 - 30	8 - 10	No	No	iPSC/2	Linked into Process
	[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
	[Goldschmidt93]	TangoLite	45	45	4	No	No	MIPS	Memory Buffers
Object	[Borg89]	Epoxie	8 - 12	8 - 12	5	Yes	No[1]	Titan	Global Buffer
	[Chen93]	Epoxie2	15	15	2	Yes	Yes	R3000	Global Buffer
	[Srivastava94] [Eustace94]	ATOM	6 -13	6 - 13	—	No	Yes	Alpha	Linked into Process
Binary	[Smith91]	Pixie	10	10	4 - 6	No	No	MIPS	File / Pipe
	[Stephens91]	Goblin	20	20	10	No	No	RS/6000	Linked into Process
	[Pierce94]	IDtrace	12	12	12	No	No	i486	File / Pipe
	[Larus93]	Qpt	10 - 60	2-5	3	No	No	MIPS, SPARC	File + Post Process
	[Larus95]	EEL	—	—	—	No	No	MIPS, SPARC	—

Tracing only significant events reduces both the size and execution time of the annotated program. Programs annotated by MPtrace, for example, are only about 4 to 6 times larger than usual, and exhibit slowdowns 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 trace distortion due to time dilation, a source of error that can be substantial in the case of multiprocessor memory simulation. TangoLite [Goldschmidt and Hennessy 1993], a successor to Tango [Davis et al. 1991], minimizes the effects of time dilation in a different way by determining event order through event-driven simulation. 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 increases overall slowdowns 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 annotation at the object-module level can help to simplify the preparation of a workload. In particular, source code for library object modules is no longer needed. Wall [1992] argues that annotating code at this level is only slightly more difficult because data-relocation tables and symbol tables are still available. An early example of this form of code annotation is Epoxie, implemented for the DEC Titan [Borg et al. 1989, 1990; Mogul and Borg 1991], and later ported to MIPS-based DECstations [Chen 1993]. In both of these systems, slowdowns for the annotated programs ranged from about 8 to 15 and code expansion ranged from 2 to 5.

Code annotation at the executable level is the most convenient to the end-user because it is not necessary to annotate a collection of source and/or object files to produce the final program. Instead, a single command applied to one executable file image generates the desired annotated program. Unfortunately annotation at this level is also the most difficult to implement because executable files 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 [Pierce and Mudge 1994].

Despite these difficulties, there exist several program-annotation tools that operate at the executable level. An early example is Pixie, which operates on MIPS executables [MIPS 1988; Smith 1991]. The popularity of Pixie has prompted the development of several similar programs that work on other instruction-set architectures. These include Goblin [Stephens et al. 1991] and IDtrace [Pierce and Mudge 1994], which operate on RS/6000 and i486 binaries, respectively. A second generation of the AE tool, called Qpt, can operate on both MIPS and SPARC binaries [Larus 1993]. The slowdowns and memory overheads for each of these static annotators compare favorably with the best dynamic emulators discussed in the previous section.

A common problem with many code annotators is that they produce traces with an inflexible level of detail, requiring a user to select the monitoring of either data or instruction references (or both) with an all-or-nothing switch. Many tools are similarly rigid in the mechanism that they use to communicate addresses, typically forcing the trace through a file or pipe interface to another process containing the trace processor. Some more recent tools, such as ATOM [Srivastava and Eustace 1994; Eustace and Srivastava 1994] and EEL [Larus 1995] overcome these limitations. ATOM offers a flexible interface that enables a user to specify how to annotate each individual instruction, basic block, and procedure of an executable file; at each possible annotation point the user can specify the machine state to extract, such as register values or addresses, as well as an analysis routine to process the extracted data. If no annotation is desired at a given location, ATOM does not add it, thus enabling a minimal degree of annotation to be specified for a given application. For 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. Eustace and Srivastava [1994] report that addresses for cache simulation can be collected from ATOM-annotated SPEC92 benchmarks with slowdowns of between 6 and 13. EEL is a similarly flexible executable editor that is the basis of a new version of qpt as well as a high-speed cache simulator named Fast-Cache [Lebeck and Wood 1995], which we discuss in Section 8.

In general, code annotators are not capable of monitoring multiprocess[5] workloads or the operating system kernel, but there are some exceptions. Borg et al. [1989, 1990] and Mogul and Borg [1991] describe modifications to the Titan operating system, Tunix, that support tracing of multiple workload processes by Epoxie. Tunix interleaves the traces generated by multiple processes into a global trace buffer that is periodically emptied by a trace-processing program. These researchers also experimented with annotating the Tunix kernel itself, although they do not report any results obtained from these traces [Mogul and Borg 1991]. Chen continued this work by porting a version of Epoxie to a MIPS-based DECstation running both Ultrix and Mach 3.0 to produce traces from single-process workloads including the user-level X and BSD servers, and the kernel itself [Chen and Bershad 1993; Chen 1994]. A recent version of ATOM can annotate OSF/1 kernels, but because ATOM analyzer routines are linked into each annotated executable, there is no straightforward way to capture systemwide, multiprocess activity. For example, ATOM cannot easily simulate a cache that is shared among several processes and the kernel because the analyzer routines for each executable have no

---

[5] Many of the tracing tools discussed in this section were designed to monitor multithreaded workloads running on a multiprocessor memory system (e.g., MPtrace, TRAPEDS, TangoLite). However, the multiple threads in these workloads run in the same protection domain (process), so we consider them to be single-process workloads.

knowledge of the memory references made in other executables.

By definition, static code annotation does not handle code that is dynamically compiled at run-time. Dynamically linked code also poses a problem although some systems, such as Chen's, treat this problem in special cases (he modified the BSD server to cause it to dynamically map a special annotated version of the BSD emulation library into user-level processes that require a BSD API).

With respect to trace detail, these methods naturally produce virtual addresses tagged by access type and size, and some of the systems that can annotate multiprocess workloads are also able to tag references with a process identifier [Borg et al. 1989]. Associating a true physical address with each virtual address is, however, very difficult because an annotated program is expanded in size and therefore utilizes virtual memory very differently than an unannotated workload would.

The tools that include multiprocess 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 annotated 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 countered by reprogramming the clock-generation chip [Borg et al. 1989; Chen and Bershad 1993]. 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 annotated 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 annotate, the TLB miss handlers (to account for increased TLB misses).

These tools share a number of common characteristics. First, they are on average about twice as fast as instruction-set emulation techniques, although some of these tools are outperformed by very efficient emulators, such as 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 annotation, it can be difficult to locate all of the executables that make up a complex multiprocess workload. 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 annotated by hand in assembly code. Note that static annotation 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, such as X applications, which are often larger than a megabyte, but only touch a fraction of their text segment [Chen 1994].

4.5 Single-Step Execution

Figure 2 shows that the highest level of system abstraction for collecting address traces is 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,[6] or by breakpoint instructions that cause kernel traps whenever they are executed [Kane and Heinrich 1992; Intel 1990]. 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 describe the use of traces obtained through single-stepping include Wiecek [1982], Clark et al. [1985], and Winsor [1989].

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. Unfortunately, slowdowns for this technique are high, with estimates varying widely from 100 [Agarwal et al. 1988] to 1,000 [Flanagan et al. 1992] to 10,000 [Holliday 1991]. 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 [Thekkath and Levy 1994].

Although there is nothing inherent in this approach that limits traces to a single process, 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 [Holliday 1991], 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. Recently, however, some new tools that trap only after certain events (such as a simulated cache miss) have led to a resurgence of trap-based monitoring. We examine some of these tools near the end of this survey in Section 8.

4.6 Summary of Trace Collection

Table IV 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., ease of use), it is difficult to accurately and fairly summarize all considerations in a single table. It is nevertheless worthwhile to attempt to do so, so that some general conclusions can be drawn. We begin by describing how to interpret the table.

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 com-

---

[6] Please see Digital [1986], AMD [1991, 1993], Motorola [1993, 1990], and Hewlett-Packard [1990].

Table IV. Summary of Trace-Collection Methods

Characteristics		External Probe-based	Microcode Modification	Instruction-set Emulation	Static Code Annotation	Single-step Execution
Completeness	Multi-process Workloads	Yes	Yes	Maybe	Maybe	No
	OS Kernel Code	Yes	Yes	Maybe	Maybe	No
	Dynamically-compiled Code	Yes	Yes	Yes	No	No
	Dynamically-linked Code	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
	Process 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
Speed (Slowdown)		1,000 +	10 - 20	15 - 70	10 - 30	100 - 10,000
Memory (Workload Expansion + Buffers)		External Buffer	Buffer	4 - 40	10 - 30 + Buffer	Buffer
Portability		Low	Very Low	High-Medium	Medium	High
Expense		High	Medium	Medium-Low	Medium-Low	Low
Ease-of-Use		Low	High	High	High-Low	High

plete address trace from a significant-events file (in the case of certain code annotation 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 include external trace buffers and memory from the simulator host machine that is consumed either by trace data or by a workload expanded in size due to annotation. 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 these qualifications, it is possible to draw some general conclusions about how the different trace-collection methods compare. A first observation is that high-quality traces are still quite difficult to obtain. Methods that by their nature produce 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 annotation) generally have to fight inherent limitations in the quality of traces that they can collect, particularly with respect to completeness (multiprocess and kernel references). Second, none of the methods are able to collect complete traces with a slowdown of 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 from favor. The probe-based and microcode-based methods probably produce the highest quality traces as measured by completeness, detail, and distortion, but their applicabil-

Table V. Methods for Address Trace Reduction

Method	Reference	Reduction Factor	Decompression Slowdown	Simulation Speedup	Exact?	Error	Restrictions
Trace Compression	[Samples89]	10 - 100	100 - 200	1	Yes	N/A	None
Significant-event Traces	[Larus90; 93]	10 - 40	20 - 60	1	Yes	N/A	None
	[Eggers90]	—	1,000 +	1	Yes	N/A	None
Stack Deletion Filter	[Smith77]	5 - 100	0	4 - 50	No	< 4 - 5%	Fully-associative Memories
Snapshot Filter	[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

ity could be limited if designers fail to provide certain types of hardware support or greater accessibility in future machines. Code annotation 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 multiprocess 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 static code annotation in the future.

5. TRACE REDUCTION

Once an address trace has been collected, it is input to a trace-processing simulator or stored on disk or tape for processing at a later time. Considering that a modern uniprocessor operating at 100 MHz can easily produce half a gigabyte of address-trace data every second, there has been considerable interest in finding ways to reduce the enormous size of traces to minimize both processing and storage requirements. Fortunately address traces exhibit high spatial and temporal locality, so there are many opportunities for achieving high factors of trace reduction. Several studies have, in fact, shown that the information content of address traces tends to be very low, suggesting that trace compaction or compression techniques could be quite effective [Hammerstrom and Davidson 1977; Becker and Park 1993; Pleszkun 1994].

There are several criteria for evaluating and comparing different methods of trace reduction (see Table V). The first, of course, is the trace *reduction factor*. The time required to reconstruct or decompress a trace is also important because it directly affects simulation times. Ideally, trace reduction achieves high factors of compression without reducing the accuracy of simulations performed by the reduced traces. It may, however, be acceptable to relax the constraint of exact trace reduction if higher factors of compression can be attained and if the resulting simulation error is low. If results are not exact, Table V shows the amount of error and its relationship to the parameters of the memory structure being simulated. Many trace reduction methods make assumptions about the type of memory simulation that will be performed using the reduced trace. Table V shows when and how these assumptions imply restrictions on the use of the reduced trace.

5.1 Trace Compression

One approach to trace reduction is to apply standard data-compression algorithms. As an example, the UNIX compress utility, which implements the Lempel-Ziv algorithm [Ziv and Lempel 1976], achieves a compression factor of about 3 to 5 on typical address traces

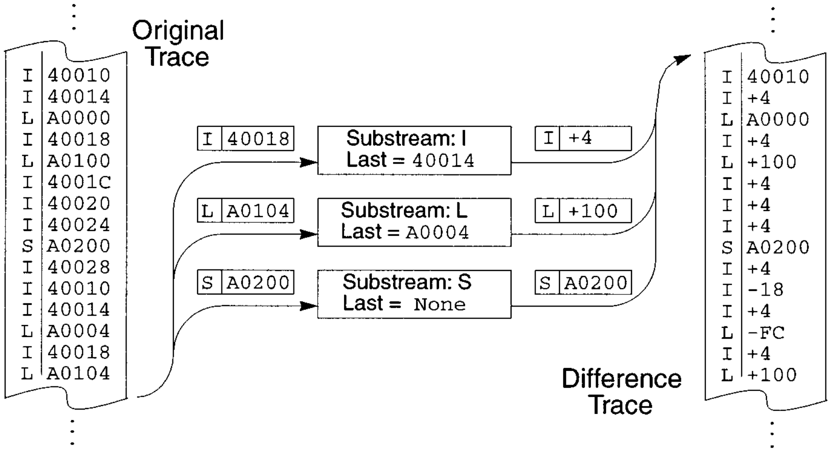


Figure 5. Computing a difference trace.

[Agarwal and Huffman 1990]. Samples showed that much higher degrees of compression can be attained if a full address trace is first preprocessed to produce a *difference trace*, as is done in Mache [Samples 1989]. Mache computes a difference trace by dividing a full address trace into substreams according to some separation rule (see Figure 5). A simple separation rule is to create one substream from all instruction references, one from all data reads, and one from all data writes. Since the full trace will often have *labels* attached to each address to identify their type (instruction fetch, load, store, etc.), it is a simple matter to determine to which substream a given address corresponds. As they are encountered, the first (base) addresses from each substream are emitted, along with their identifying substream labels, to the output difference trace. The arithmetic difference between subsequent addresses and their immediate predecessors within each substream is then computed, and the absolute value of this difference is compared against some predetermined threshold. When the difference is less than the threshold, only the difference and the substream label are emitted to the output. If the difference is greater than the threshold, then the entire address value and label are emitted. The original, full address trace can be reconstructed from the difference trace by starting with the base address in each substream, and then adding the sequence of difference values, step by step, to obtain a sequence of full address values.

A difference trace improves trace reduction factors for the following reasons. First, the number of bytes required to encode difference values is less than that required for full addresses. Only 16 bits are required to encode a difference value with a threshold of 8192 and three label types (13 bits for the absolute value of the difference, 1 sign bit, and 2 bits for the label), which is one half or one quarter the amount of data required to specify a full 32-bit or 64-bit address. Second, a difference trace exposes regularity and striding patterns in a trace that can be better exploited by the Lempel-Ziv algorithm. When Samples applied Lempel-Ziv compression to his difference traces, overall compression factors increased to 10 to 20 for traces with mixed instruction and data references, and to as high as 100 for traces with instruction references only. Mache retains the full information content of traces, so simulations using Mache 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, times reported by Samples imply that decompression can add a slowdown factor of as much as 200 to trace-driven simulations.

5.2 Significant-Event Traces

Tools such as MPtrace [Eggers et al. 1990], AE [Larus 1990], and qpt [Larus 1993], which we first described in Section 4.4, produce significant-event traces that are typically much smaller than full address traces. AE traces, for example, are 10 to 40 times smaller than full traces, whereas those from MPtrace are reported to be as much as 1,000 times smaller. Because these systems provide a method for reconstructing the full address trace, they can be viewed as trace-reduction systems that annotate a workload to produce a reduced trace directly. As with Mache, the complete trace is regenerated in these systems, so simulations using these traces are unrestricted and exact, but there is no simulation-time savings. As noted previously, AE and qpt can slow overall simulations by 20 to 60, whereas MPtrace can make overall simulation times as much as three orders of magnitude slower.

5.3 Trace Filtering

A designer often has a specific purpose in mind for a given set of address traces. The traces might only be used for cache simulations where the cache size is larger than some specific minimum size and where the line size is fixed. In such a situation, a full address trace can be reduced in size substantially, provided that the resulting reduced trace is used only for simulations in an appropriately constrained design space. Smith [1977] has suggested two examples of this form of trace reduction. 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*. With the first method, stack deletion, a full memory trace is used to simulate an LRU stack memory. Addresses that hit in the top D entries of the stack are discarded, and addresses that miss are concatenated to form a reduced trace. The rationale behind this procedure 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. Smith's second technique, the snapshot method, constructs a reduced trace by concatenating snapshots of memory contents taken at periodic intervals separated 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 [Prieve 1974]. The rationale for this method is similar to that of the stack-deletion method; the memory snapshots capture the most important references, while filtering repeated references to the same location. Depending on the values of the deletion parameter, D, or 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 (4 to 50) are not directly proportional to the compression factors (5 to 100). This is because simulations with reduced traces 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

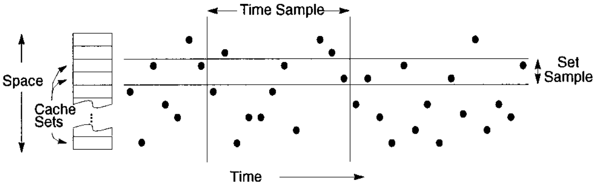


Figure 6. Time and set sampling.

more time, per trace event, than they do on the full trace.

Trace stripping, first suggested by Puzak [1985] in his dissertation, also produces reduced traces that can be used only in a restricted design space. A full address trace is used to simulate a small direct-mapped cache with a given line size, and 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 those of the filter cache, Puzak showed that some simulation error results, but it is generally less than 10% and decreases with increasing cache associativity. Wang and Baer [1990] extended the cache filter concept to enable the simulation of write-back caches. 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, reduction factors are in the range of 10 to 20, but as with Smith's methods, the simulation speedups may not be directly proportional to the trace-reduction factor.

Agarwal and Huffman [1990] noted that cache filters exploit only temporal, but not spatial, locality in address trace. They devised another form of trace filter, called a *block filter*, which provides an additional order of magnitude reduction in the size of a trace that has already been cache-filtered. 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 emits only one 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 [Hodges and Lehmann 1964], where the strata correspond to spatial localities. Agarwal and Huffman show that application of the block filter can increase overall trace reduction factors to as high as 100, while keeping the error in simulation results under 10% to 12%.

5.4 Trace Sampling

When faced with a very large (or infinite) set of data to analyze, it is often helpful to resort to statistical methods to select a subset, or *sample*, of the complete data population. When properly constructed, a sample can be used to derive estimates for some statistic of interest without having to process the entire data set. A full address trace can be viewed as a large data set, and traditional methods for statistical sampling can therefore be used as another method for reduction of trace data. Two basic approaches to trace sampling have been proposed in the literature: *time sampling* [Laha et al. 1988] and *set sampling*, which is also known as *congruence-class sampling* [Puzak 1985] (see Figure 6). We discuss the pros and cons of each method in greater detail in the following.

Laha et al. [1988] constructed trace samples by extracting from a full trace contiguous segments of memory references over certain windows of time. Each trace segment (or trace sample) 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 then averaged to form an estimate of the true performance for the entire trace. This method, called *time sampling*, must be conducted with care to avoid errors. First, a sufficient number of trace segments must be collected (Laha et al. suggest 35) to ensure that different phases of execution along the full trace are adequately represented. A second source of error is due to not knowing the state of a simulated cache at the beginning of a trace sample. This form of error, commonly known as *cold-start bias*, occurs because it is not possible to know whether the initial references to each cache set hit or miss. For the simulation of relatively small caches (<128 Kbytes), where errors due to cold-start bias are small, Laha's 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.

As simulated cache sizes increase, cold-start bias becomes an increasingly significant source of error. Several ad hoc methods have been proposed to remove or reduce this effect. One technique is to begin measuring miss ratios only in cache sets that have been primed (i.e., sets that have become filled with references from the beginning of the trace sample) [Laha et al. 1988; Stone 1993]. Another method is to concatenate, or "stitch" together, the individual trace samples under the assumption that the state of the cache at the end of one sample approximates the true cache state at the beginning of the subsequent sample [Agarwal et al. 1988]. Still another method is to use the first half of the references in a trace sample to partially prime the cache, and then to simulate the remaining references to estimate the miss ratio [Kessler 1991]. Wood et al. [1991] proposed a more theoretically sound method for estimating the miss ratio of unknown references by using renewal theory. A key observation of Wood's model is that the miss ratio of *unknown references* (i.e., references to cache sets that have not yet been filled) is typically substantially higher than the miss ratio of the remaining references in the time sample. Kessler compared and evaluated the effectiveness of several bias-reducing techniques when simulating large (multimegabyte) caches, and concluded that Wood's method generally performed best, although even it was unable to compensate for trace samples that were too short relative to the simulated cache size [Kessler 1991]. Kessler suggested two rules of thumb for deciding when the trace-sample length is sufficiently large to avoid errors when using Wood's method: the trace sample must fill at least half of the cache, and there must be at least as many misses to full sets as cold-start misses [Kessler 1991].

An alternative sampling method is to select memory references from a full trace on the basis of the cache set or sets to which they map. This method is commonly called *set sampling* or *congruence-class sampling* [Puzak 1985; Kessler 1991]. With set sampling, the reduced trace is constructed by defining the parameters (size, associativity, line size) of some cache, and then keeping exactly those addresses that reference a certain collection of cache sets, while discarding references to the other sets. 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 the references made to it by the full trace, this method does not suffer from cold-start bias as does time sampling.

Set sampling does, however, introduce some complications of its own.

The first issue to resolve is the method for selecting which cache sets to sample. One approach is to select the sampled sets randomly (given a cache of a specific size, associativity, and line size). Though simple, this approach suffers from the disadvantage that an entirely different set sample might need to be obtained to simulate caches with a different set of parameters. This is so because randomly selected set samples for two caches may be incompatible whenever the set-indexing bits for the two caches differ. To overcome this problem, Kessler [1991] proposed *constant-bits selection*, which includes in the trace sample all addresses with the same constant value in certain address bits. A simple example, drawn from Kessler's explanation, helps to illustrate the technique. Assume, first, that cache sets are selected (indexed) by the lowest-order address bits immediately to the left of the address bits that specify the offset within a line (where bit 0 is the least significant address bit). Kessler shows that if all references to memory addresses with some specific constant value (e.g., 0000, 0010, etc.) in address bits 11 to 8 are retained, then approximately 1/16th of the total trace will be sampled, assuming that the probability of accessing different addresses is uniformly distributed. Kessler proved that trace samples obtained in this way can be used to simulate any cache whose address index bits include the constant bits. In other words, any cache whose line size is 256 bytes or less, and whose size divided by its associativity is greater than 2 kilobytes can use the sampled trace.

Puzak [1985] 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. Kessler [1991] compared the effectiveness of set sampling with time sampling in his simulations of multimegabyte secondary caches. He showed that set sampling is generally able to satisfy a goal of 10% sampling with less than 10% error for large caches (greater than one megabyte), but time sampling breaks down in this range, mainly due to error from cold-start bias.

An important disadvantage of set sampling is that it cannot be used for simulations of memory systems that must model time-dependent behavior or that must take into account interactions between sets. For example, write buffers, which handle write access to all cache sets, cannot be simulated accurately if only a subset of cache sets is represented by the trace. Similarly, many cache prefetch algorithms depend on accesses to other cache sets. Sequential prefetch, for example, fetches the cache line following the current line. Because a set sample may not include one of two adjacent cache lines, it is impossible to simulate the initiation of a sequential prefetch, or to determine if the prefetch results in any benefit.

5.5 Summary of Trace Reduction

The most appropriate trace reduction method often depends on the questions to be answered by the simulation study, and because many of the methods restrict the way that a 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 methods based on standard data compression or significant-events tracing provide good solutions with size-reduction factors as high as 100, but with trace-reconstruction times that can slow simulations by as much as 50 to 200. If simulation errors of 10% or less are considered acceptable, then filtering and sampling methods provide a good solution, with space *and* time reduction factors of as high as 10 to 50.

As a final note, some of these trace reduction methods can be combined to produce multiplicative improvements in compression factors. A cache-filtered trace, for example, could also be time or set sampled. Similarly, standard data-compression algorithms can be applied to most traces 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.

6. TRACE PROCESSING

The ultimate objective of trace-driven simulation is, of course, 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 final stage of trace-driven simulation is often the most time-consuming component because a designer is typically interested in hundreds or thousands of different memory configurations in a given design space. As an example, the space of simple caches defined by sizes ranging from 4 to 64 Kbytes (in powers of two), line sizes ranging from 1 to 16 words (in powers of two), and associativities ranging from 1-way to 4-way, contains 100 possible design points. Adding the choice of different replacement policies (LRU, FIFO, Random), different set-indexing methods (virtually or physically indexed), and different write policies (write-back, write-through, write-allocate) creates thousands of additional possibilities. These design options are for a single cache, but actual memory systems are typically composed of multiple caches that cooperate and interact in a multilevel 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. Researchers have explored two basic approaches to dealing with this problem: parallel distributed simulations, and multiconfiguration simulation algorithms.

The first approach exploits the trivially parallelizable nature of trace-driven simulations and the abundance of unused computing cycles on networks of workstations; each memory configuration of interest can be simulated completely independently from other configurations, so it is a relatively simple matter to distribute multiple simulation jobs across the underutilized workstations on a network. In practice, there are some complications with this approach. If, for example, the "owner" of a workstation wants to reclaim the resources of the computer sitting on his or her desk, it is useful to have a method for suspending or moving a compute-intensive simulation task that has been started on the machine. Another problem is that networks of workstations are notoriously unreliable, so keeping track of which simulation configurations have successfully run to completion can be an unwieldy task. Several software packages for workstation-cluster management, which offer features such as process migration, load balancing, and checkpointing of distributed batch simulation jobs, help to solve these problems. These systems are well-documented elsewhere (see Baker [1995] for a survey), so we discuss them no further here.

Algorithms that enable the simulation of multiple memory configurations in a single pass of an address trace offer another solution to the compute-intensive task of exploring a large design space. We use several criteria to judge multiconfiguration simulation algorithms in this survey (see Table VI). First, it is desirable that the algorithm be able to vary several simulation *parameters* (cache size, line size, associativity, etc.) at a time and, second, that it be able to produce any of several different *metrics* for performance, such as miss counts, miss ratios, misses per instruction (MPI), write backs, and cycles

Table VI. Multiconfiguration Memory Simulators

Reference	Name	Range of Parameters					Metrics	Overhead
		Sets	Line	Assoc	Write Policy	Sector		
[Mattson70]	Stack Processing	Fixed	Fixed	Vary	None	No	Misses, Miss Ratio, MPI	—
[Hill87]	Forest Simulation	Vary	Fixed	1-way	None	No	Misses, Miss Ratio, MPI	< 5%
[Hill87]	All-Associativity	Vary	Fixed	Vary	None	No	Misses, Miss Ratio, MPI	< 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%

per instruction (CPI). The *overhead* of performing a multiconfiguration simulation relative to a single-configuration simulation is also of interest because this value can be used to compute the effective simulation speedup relative to the time that would normally be required by several single-configuration simulations.

6.1 Stack Processing

Mattson et al. [1970] 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. In their original paper they introduced a method, called *stack processing*, which determines the number of memory references that hit in any size of fully associative memory that uses a *stack algorithm* for replacement. Their technique relies on the property of *inclusion*, which is exhibited by certain classes of caches with certain replacement policies. Mattson et al. show, for example, that an n-entry, fully associative cache that implements a least recently used (LRU) replacement policy includes all 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 7. 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 (i.e., *hits*) in the stack at some *stack depth* (Case I), 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. The figure shows that 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.

To record the performance of different cache sizes, the algorithm also maintains an array that counts the number of hits at each stack depth. As a consequence of the inclusion property, 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:

$$\text{hits}_n = \sum_{i=0}^{n-1} \text{hits}[i]. \tag{9}$$

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:

$$\text{misses}_n = \text{totalReferences} - \text{hits}_n \tag{10}$$

$$\text{missRatio}_n = \text{misses}_n/\text{totalReferences} \tag{11}$$

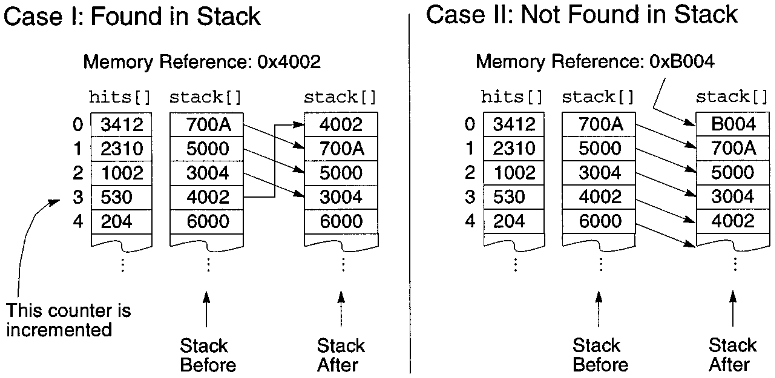


Figure 7. Data structures for stack simulation.

$$MPI_n = misses_n/totalInstructions. \tag{12}$$

Mattson et al. [1970] give other examples of stack replacement algorithms (such as OPT), and also note that some replacement policies, such as FIFO, are not stack algorithms. In their original paper, and in a collection of other follow-up reports (see Sugumar [1993] or Thompson and Smith [1989] 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 multiconfiguration simulation algorithms. Many researchers have advanced multiconfiguration simulation by proposing various enhancements and by reporting simulation times for actual implementations of these improvements. We focus on a selection of recent papers that extend the range of multiconfiguration parameters, and that characterize the current state-of-the-art in this form of simulation (see Table VI).

6.2 Forrest and All-Associativity Simulation

Hill [1987] noted that the original stack algorithm of Mattson et al. requires the number of cache sets and the line size to be fixed. 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; Hill's *forest-simulation* algorithm supports this form of multiconfiguration simulation. 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 [1989] developed extensions that count the number of writes to main memory for different-sized caches that implement a write-back write policy. They also studied multiconfiguration algorithms for sector or subblock caches. Wang and Baer combined the work of Mattson et al. [1970], Hill and Smith [1989] and Thompson and Smith [1989] to compute both miss ratios and write-backs in a range of caches where both the number of sets and the associativity are varied. In his dissertation, Sugumar [1993] 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.

6.3 Summary of Trace Processing

There are several points to be made about multiconfiguration 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 multiconfiguration simulation of two or more configurations would perform as well as or better than collections of two or more single-configuration simulations. These results should, however, 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 multiconfiguration is very low. If, however, the trace input times are relatively low, then the multiconfiguration overheads will be much higher. This is the case with Sugumar's Cheetah simulator which appears to have very high overheads relative to Hill's Tycho simulator [Hill 1987; Sugumar 1993] (see Table VI). Cheetah's overall simulation times are, however, approximately 8 times faster than Tycho because its input processing is more optimized [Sugumar 1993].

A second point is that even though multiple configurations can be simulated with one trace pass, it is often still necessary to re-apply multiconfiguration algorithms several times to cover an entire design space. Hill [1987] 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. For the same example, forest simulation still requires 3 separate passes but can cover only half of the space. Hill argues that all-associativity simulation is the best method in this case because although it also requires 3 separate passes, it can cover the entire design space.

Finally, despite many advances in multiconfiguration simulation, there are many types of memory systems and performance metrics that cannot be evaluated in a single trace pass. Most of these algorithms restrict replacement policies to LRU, which is rarely implemented in actual hardware. Similarly, performance metrics that require very careful accounting of clock cycles, such as CPI, generally cannot be computed for a range of configurations in a single simulation pass (e.g., simulating contention for a second-level cache between split primary I- and D-caches requires a careful accounting of exactly when cache misses occur in each cache).

7. COMPLETE TRACE-DRIVEN SIMULATION SYSTEMS

Until now, we have examined the three components of trace-driven simulation in isolation. In this section we examine some of the ways that these components can be combined to form a complete simulation system. Figure 1 suggests a natural composition of the three components in which they communicate through a simple linear interface of streaming addresses that may include some form of buffering between the components. Because of the high data rates required, the selection of mechanisms used to transfer 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 increase overall slowdowns. In this section we examine the pros and cons of different interfacing methods and summarize some overall simulation slowdowns as reported in the literature, as well as those measured by our own experiments.

7.1 Trace Interfaces

Because address traces conform to a simple linear data stream model, there are several options available for communicating and buffering them (see Figure 8). Some simulators rely on mechanisms provided by the host operating system (*files* or *pipes*), and others implement communication on their own using standard procedure calls or regions of memory shared between the trace col-

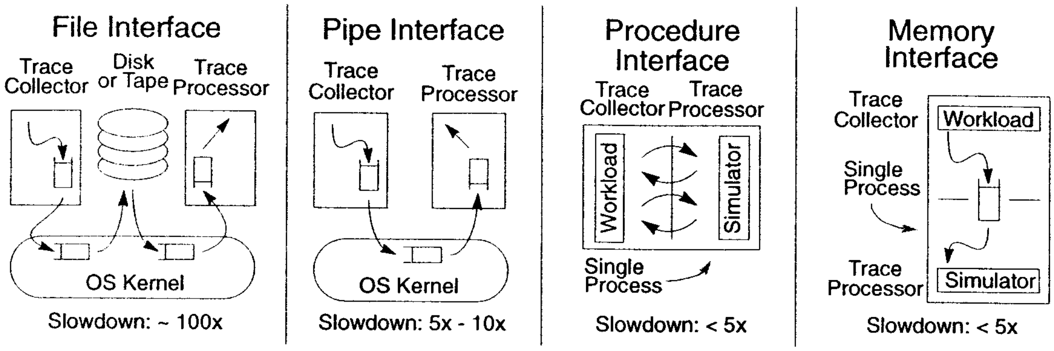


Figure 8. Trace interfaces.

lector and the trace processor. We examine each of the possibilities in turn.

Because they are backed by secondary storage devices, files provide the advantages of deep and nonvolatile buffering. These capabilities enable the postponement of trace processing as well as the ability to repeatedly use the same traces to obtain reproducible simulation results. Unfortunately, files suffer some important disadvantages, the first of which is speed. Assuming disk bandwidth of 1 MB/sec and an address-generation rate of 100 MB/sec by the host, a file stored on disk can slow both trace collection and trace processing by a factor of 100 or more. A second disadvantage of files is that they are simply never large enough. Assuming again a host address-generation rate of 100 MB/sec, a one gigabyte hard disk would be filled to capacity in about 10 seconds of real-time execution. This underscores the importance of the trace-reduction methods, described in Section 5, which can improve effective file capacity and bandwidth by one to two orders of magnitude.

Pipes, which establish a one-way channel for the flow of sequential data from one process to another, are another communication abstraction that can sometimes overcome the limitations of files. Pipes use only a moderate amount of memory (on the order of kilobytes) to buffer the data flowing between the two processes, which implies that both a trace collector and trace processor must be running at the same time to prevent buffer overflow. With this approach, which is often called *on-the-fly simulation*, traces are discarded just after they are processed. Because traces must 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 instruction-set emulators and code annotators, where slowdowns range from 10 to 70, this requirement is usually met. Communication via pipes is substantially faster than via files, with overheads typically adding 5 to 10 to overall simulation slowdown. Note that when pipes are used, trace-reduction methods are less attractive because they must be reapplied during each simulation run and thus provide little or no advantage over simply processing the full address trace.

Both files and pipes are interprocess communication mechanisms provided by an OS file system. 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 processes. These overheads can be avoided if a trace collector and trace processor run in the same process 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

Table VII. Slowdowns for Some Complete Trace-Driven Memory Simulation Systems

Name	Reference	Trace Collection	Trace Reduction	Trace Processing	Interface Method	Slowdown	Effective Slowdown
Pixie + Cache2000	[MIPS88]*	Annotation	None	Single Config	Pipe	60 - 80	60 - 80
Monster + Cheetah	—	Probe-based	Time Sample	Multi (8)	File	419	52
Pixie + Cheetah	[Sugumar93]*	Annotation	None	Multi (44)	Pipe	183	4
Pixie + Tycho	[Gee93]	Annotation	None	Multi (44)	Pipe	6250	142
gsim	[Magnusson93]	Emulation	None	Single Config	Procedure	45 - 75	45 - 75
Talisman	[Bedichek94; 95]	Emulation	None	Single Config	Procedure	100 - 150	100 - 150
TangoLite	[Goldschmidt92; 93]	Annotation	None	Single Config	Memory	765	765
Epoxie + Panama	[Borg89]	Annotation	None	Single Config	Memory	100	100

same process address space (see Table III). 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 process'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 procedure calls over many addresses. By bringing communication slowdowns under a factor of 5, both of these methods improve over files and pipes, but it should be noted that placing a simulator in the same process as the monitored workload can complicate the monitoring multiprocess workloads.

7.2 Complete Trace-Driven Simulation Slowdowns

Because of the variety of trace-driven simulation techniques and the ways to interconnect them, overall trace-driven simulation slowdowns range widely. 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 VII lists several complete trace-driven simulators composed of many different types of trace-collection and trace-processing tools. As such, these systems are fairly representative of the sort of simulators that can be constructed with state-of-the-art methods. We must be careful when comparing the different slowdowns reported in Table VII because each corresponds to the simulation of different memory configurations[7] at different levels of detail, running different workloads, and using different instruction-set architectures. The table does, however, enable us to draw some general conclusions about the achievable speed of standard trace-driven simulation systems.

As Table VII shows, complete simulators rarely exhibit slowdowns of less than about 100, with a few rare exceptions that are able to achieve slowdowns of around 50. The fastest integrated simulator was gsim, with reported slowdowns in the range of 45 to 75 for a relatively simple workload (an optimized version of the Drystone bench-

---

[7] For tools that enable multiprocessor memory simulations we report the slowdowns for one processor only to enable more meaningful comparisons with the uniprocessor-only simulators.

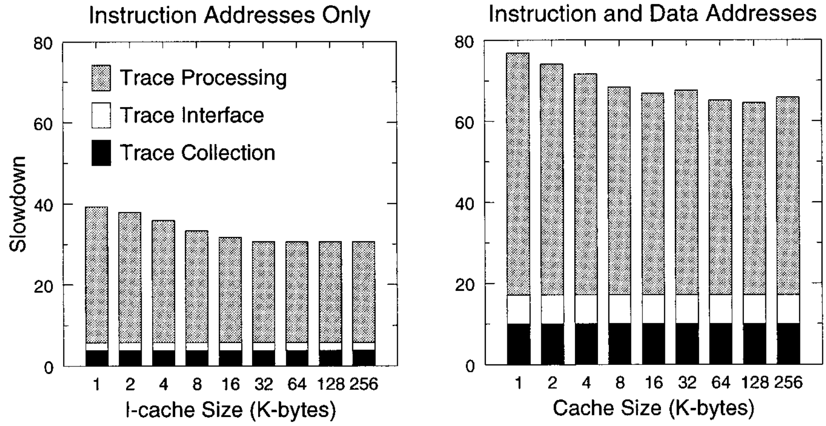


Figure 9. Components of trace-driven simulation slowdowns.

mark). The fastest composed simulator, constructed by driving Pixie traces through a pipe to the Cache2000 [MIPS 1988] trace processor, exhibits slowdowns in the range of about 60 to 80. The workload in this case is more substantial: an MPEG video decoder. By comparing the slowdowns for Cheetah driven by traces coming from a file (Monster traces) versus coming from a pipe (Pixie traces), we can see the benefits of on-the-fly trace generation and processing; the Pixie + Cheetah combination is more than two times faster than the Monster + Cheetah system, despite the fact that a greater number of configurations (44 versus 8, respectively) are being simulated. Note that the overheads of the two multiconfiguration simulators (Tycho and Cheetah) cause their overall slowdowns, relative to single-configuration simulation with Cache2000, to be much higher than the values reported in Section 6. For Cheetah, the overheads are at least 300%, and for Tycho they are an order of magnitude higher. Given the degree of their simulation detail, the integrated simulators Talisman and gsim, which are based on emulation techniques similar to those described in Section 4.3, perform quite well, providing further evidence that instruction-set emulation is a very viable technique for memory-system evaluation.

To better understand the sources of trace-driven slowdown, we measured the speed of the Cache2000 + Pixie combination over a range of instruction- and data-cache sizes. The results, shown in Figure 9, illustrate that most of the slowdowns are due to trace processing. This observation is supported by reported experiences with other tools as well. Goldschmidt and Hennessy [1992] report that trace processing in TangoLite slows a system by an additional factor 17 relative to a workload that is annotated to produce address traces only (compare the TangoLite entries in Table III with those of Table VII). Borg et al. [1989] report a similar observation, noting that their Epoxie-driven Panama simulations spend far more time processing address references than collecting them.

7.3 Summary of Complete Trace-Driven Simulation Systems

As Table VII and Figure 9 show, the generation, transfer, and processing of trace data for memory system simulation are extremely challenging—few traditional trace-driven simulators achieve slowdowns much lower than about 50, with the main bottleneck being the time required to process address traces. These results suggest that the biggest gains in overall trace-driven

Table VIII. Beyond Traces: Some Recent Fast Memory Simulators

Method	References	Name	Cycles per Hit	Cycles per Miss	Overall Slowdown	Miss-detection Mechanism	Type of Simulation	Completeness	
								Multi-process	OS Kernel
Software-based Miss Detection	[Martonosi92;93]	MemSpy	25	320 - 510	10 - 20	Annotation	D-cache	No	No
	[Lebeck95]	Fast-Cache	4	55	2 - 7	Annotation	D-cache	No	No
	[Rosenblum95] [Witchel96]	SimOS + Embra	10	—	7 - 21	Emulation	D-cache, I-cache, TLB	Yes	Yes
Hardware-based Miss Detection	[Nagle93]	Tapeworm	1 - 2	100 - 650	0.5 - 4.5	TLB Miss	TLB	Yes	Yes
	[Reinhardt93]	WWT	1 - 2	2,500[1]	1.4 - 46[1]	ECC	D-cache	No	No
	[Uhlig94]	Tapeworm II	1 - 2	300	0 - 10	ECC	I-cache, TLB	Yes	Yes
	[Lee97]	Tapeworm486	1 - 2	3,600 - 4,000	0 - 14	Page Fault	TLB	Yes	Yes
	[Talluri94]	Foxtrot	1 - 2	1,500 - 4,000	—	TLB Miss	TLB	No	No

simulation speed are likely to come either from methods that speed up trace processing, or from techniques that can avoid invoking the trace processor altogether. The latter strategy is the subject of our next section.

8. BEYOND TRACE-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 methods for eliminating or reducing the cost of processing memory references (see Table VIII). Although the mechanisms that they use differ, each of these tools works by finding special cases where a memory reference has no effect on simulated memory state. A common example is a cache hit which, unlike a cache miss, typically does not require any updates to a cache's contents.

8.1 Software-Based Miss Detection

MemSpy [Martonosi et al. 1992] is a memory simulation and analysis tool built on top of the TangoLite trace collector discussed in Section 4.4. Original implementations of MemSpy, which annotated assembly code to call a simulation routine after each heap or static-data reference, 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. Martonosi et al. observed that in the case of a cache hit, memory state need not be updated, and the call to the cache simulator can be avoided altogether. To exploit this fact, Martonosi et al. modified the annotations around each memory reference to test for a cache hit before invoking the full cache simulator. When a hit 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 320 to 510 cycles for a full call into the MemSpy simulator on a cache miss. Because cache hits are far more common than misses, the long path is infrequently invoked, and the MemSpy slowdowns were effectively reduced to the range of 10 to 20.

Fast-Cache [Lebeck and Wood 1995] is another example of a simulator that optimizes for the common case of cache hits. Fast-Cache is based on an abstraction called *active memory*, which is a block of memory with a pointer to an associated *handler* routine that is called whenever memory locations in the block are referenced. During a cache simulation, these handlers are changed dynamically to detect when cache misses occur. At the beginning of a simulation, all Fast-Cache memory blocks point to a handler for cache misses. As the blocks of memory are accessed for the first time, the miss handler is invoked, it counts the miss, and then sets the handler for the missing memory block to point to a NULL routine. Future accesses to these memory blocks (which are now resident in the simulated cache) are processed much more quickly because the NULL routine simply returns to the workload without invoking the complete cache simulator. As the simulated cache begins to fill, the miss handler will eventually begin loading newly referenced memory blocks into the cache at locations that are already occupied by other memory blocks. These cache conflict misses are modeled by resetting the handler for the displaced memory blocks to point back to the miss handler again so that future references to the displaced block will register a miss. Fast-Cache implements active memory blocks by using the EEL executable editor, described in Section 4.4, to annotate each workload instruction that makes a memory reference with 9 additional instructions that look up the state of an active memory block and invoke the appropriate handler. In the case of a NULL handler, only 5 additional instructions are required per memory reference. Depending on the workload, Fast-Cache achieves overall slowdowns in the range of about 2 to 7 for the simulation of direct-mapped data caches ranging in size from 16 KB to 1 MB. Like MemSpy, Fast-Cache simulates only data caches for single process workloads (i.e., it does not monitor instruction or operating system references).

Embra [Witchel and Rosenblum 1996] uses dynamic compilation techniques similar to those of Shade (see Section 4.3) to generate code sequences that test for simulated TLB and cache hits before invoking slower handlers for misses in these structures. Embra's overall slowdowns (7 to 21) compare very favorably with those of MemSpy and Fast-Cache, given that it simulates a more complete memory system consisting of TLB, I-cache, and D-cache. Embra runs as part of the SimOS [Rosenblum et al. 1995] simulation environment, which enables it to fully emulate multiprocess workloads as well as operating-system kernel code.

8.2 Hardware-Based Miss Detection

Simulators such as MemSpy, Fast-Cache, and Embra reduce the cost of processing cache hits, but because they are based on code annotation or emulation, they always add a minimal base overhead to the execution of every memory operation. One way around this problem is to use the host hardware to assist in the detection of simulated misses. This can sometimes be accomplished by using certain features of the host hardware, such a memory management unit or error-correcting memory, to constrain access to the host's memory and cause kernel traps to occur whenever a workload makes a memory access that would cause a simulated cache or TLB miss. If implemented properly, this method requires no instructions to be added to a workload, enabling simulated hits to proceed at the full speed of the underlying host hardware. Trap-driven simulations can thus, in principle, achieve near-zero slowdowns when the simulated miss ratio is low.

Tapeworm is an early example of a trap-driven TLB simulator that relies on the fact that all TLB misses in its host machine (a MIPS-based DECstation) are handled by software in the operating system kernel [Nagle et al. 1993]. Tapeworm works by becoming part of the operating system of the host machine that it runs on—the usual software handlers for TLB misses are modified to pass the relevant information about all user and kernel TLB misses directly to the Tapeworm simulator after each miss. Tapeworm then uses this information to maintain its own data structures for simulating other possible TLB configurations, using algorithms similar to the software-based tools described in the previous section. There are two principal advantages to compiling the Tapeworm simulator into the host operating system to intercept TLB miss traps. First, by being in the kernel, Tapeworm can capture TLB misses from all user processes, as well as the OS kernel itself. Second, because Tapeworm does not add any instructions to the workload that it monitors, nontrapping memory references proceed at the full speed of the underlying host hardware, which results in zero-slowdown processing of simulated TLB hits. On the other hand, a simulated TLB miss incurs the full overhead of a kernel trap and the simulator code, which varies 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 [Nagle et al. 1993]. The result is that Tapeworm TLB simulation slowdowns range from about 0.5 to 4.5.

Trap-driven TLB simulation has recently been implemented on other architectures with similar success. Lee [1997] has implemented a trap-driven TLB simulator on a 486-based PC running Mach 3.0. Because the i486 processor has hardware-managed TLBs, Lee's 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 causes TLB misses to result in kernel traps in the same way that they do in a machine with software-managed TLBs. Talluri and Hill [1994] use 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. Talluri and Lee both report that the overall slowdowns for their simulators are comparable to those of Tapeworm.

A limitation of the trap-driven simulators previously described 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 [Reinhardt et al. 1993]. Because each memory location has ECC bits, this method enables traps to be set and cleared with a much finer granularity, enabling cache simulation. As with the trap-driven TLB simulators previously noted, a simulated cache hit in WWT runs at the full speed of the host machine, and for caches with low miss ratios, overall slowdowns are measured to be as low as 1.4. However, in a comparison with Fast-Cache, Lebeck and Wood [1994] report that WWT exhibits slowdowns of greater than 30 or 40 for caches smaller than 32KB. 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.

Tapeworm II, a second-generation Tapeworm simulator which also uses ECC-bit modification to simulated caches, improves on the speed of WWT by showing that trap-handling times can be reduced by nearly an order of magnitude to about 300 cycles, bringing overall simulation slowdowns for instruction caches into the range of 0 to 10 [Uhlig et al. 1994]. Tapeworm II, like the original Tapeworm, also demonstrates that trap-driven cache simulation is capable of complete monitoring multiprocess and operating system workloads. Experiments performed with Tapeworm II show that trap-driven simulation slowdowns are highly dependent on the memory structure being simulated, with the relationship between slowdown and configuration parameters often being quite different from trace-driven simulation. Trace-driven simulations of associative caches, for example, are typically slower than direct-mapped cache simulations because of the extra work required to simulate an associative search. With trap-driven simulations, however, the opposite is true: Tapeworm's associative-cache simulations are faster because there is a lower ratio of misses (and thus traps) to total memory references relative to simulations of direct-mapped caches of the same size. Other experiments with Tapeworm II have examined sources of measurement and simulation error of trap-driven simulation compared with those of trace-driven simulation. Many sources of error are the same (e.g., time dilation), but some were found to be unique to trap-driven simulation. In particular, because Tapeworm II becomes part of its running host system, it is more sensitive to dynamic system effects, such as virtual-to-physical page allocation and memory fragmentation in a long-running system. Although Tapeworm's sensitivity to these effects may necessitate multiple experimental trials, this should not be viewed as a liability; a trap-driven simulator that becomes part of a running system can give insight into real, naturally occurring system effects that are beyond the scope of static traces.

8.3 Summary of New Memory Simulation Methods

With slowdowns commonly around 10, and in some cases approaching 0, the new simulators discussed in this section show that memory simulation speeds can be improved dramatically by rejecting the traditional trace-driven simulation paradigm of collecting and processing each and every memory reference made by a workload. There are substantial performance gains to be had by optimizing for the common case of cache or TLB hits.

The three software-based systems (MemSpy, Fast-Cache, and Embra/SimOS) share a number of important advantages. They are flexible, low in cost, and relatively portable because they do not rely on special hardware support. Because they are based on the same basic techniques as trace collectors that use code annotation or emulation, these three tools suffer from some of the same disadvantages, such as memory overheads as high as 5 to 10 due to added instructions and/or emulation state. Code expansion may not be a concern for applications with small text segments, but annotating larger multiprocess workloads along with the kernel can cause substantial expansion.

The hardware-based trap-driven simulators, such as Tapeworm II and WWT, avoid the problems of code expansion, and they are also able to achieve near-zero slowdowns when miss ratios are small. The main weakness of trap-driven simulation is low flexibility and portability—all of the trap-driven simulators that we examined were limited in the simulations that they could perform, and all rely on ad hoc methods to cause OS kernel traps.

Although hit overheads are zero with the hardware-based methods, their miss costs are on average much higher than those for the software-based techniques. This suggests that the fastest method depends highly on the ratio of hits to misses for a given workload and memory configuration. Lebeck and Wood [1995] studied this issue and concluded that a hardware-based approach is better for miss ratios up to about 5%, at which point the high cost of servicing miss traps begins to make a software-based approach more attractive. Given this, the software-based methods are probably the better choice for simulating small on-chip caches with their higher miss ratios, but the trap-driven methods are more effective for simulating large off-chip caches, which have traditionally been difficult to manage with standard trace-driven simulation because of the time it takes to overcome cold-start bias [Kessler 1991].

Both the hardware- and software-based techniques have been shown capable of monitoring complete OS and multitask workloads (e.g., SimOS, Tapeworm II). The Tapeworm II approach of compiling the trap handlers directly into the kernel of the host system enables it to benefit from much of the existing host infrastructure. SimOS, by contrast, must develop detailed simulation models of several system components (such as network controllers, disk controllers, etc.) to achieve the same effect. Although more work is required to establish these models, SimOS, in the end, is able to account for effects such as time dilation, a form of distortion for which Tapeworm II has difficulty compensating.

When hit-bypassing is implemented in software, it limits the effectiveness of techniques such as time sampling [Laha et al. 1988] and set sampling [Puzak 1985]. Martonosi et al. [1993] investigated time sampling by adding an additional check to MemSpy's annotations that enabled and disabled monitoring at regular intervals. When enabled, annotation overheads are similar to those cited previously (25 instructions per hit), but when disabled, an annotated reference executes only 6 extra instructions. When trapping is enabled for 10% of the entire execution time, MemSpy slowdowns dropped to about 4 to 10, a factor of 2 improvement over simulations without sampling. Ideally 10% sampling would result in a factor of 10 speedup, but in this case, code annotation adds an unavoidable base overhead; even when trapping is turned off, each annotated memory reference still results in the execution of 6 extra instructions. In contrast, experiments with Tapeworm II show that the trap-driven approach lends itself well to sampling [Uhlig et al. 1994]—when Tapeworm samples 1/*N*th of all references, slowdowns are reduced in direct proportion, by a factor of *N*. This is true because unsampled references, like simulated cache hits, can run at the full speed of the host hardware.

The tradeoffs between these new memory system simulators are complex, and neither the software-based nor hardware-based approaches are clear winners in every situation. The reliance on ad hoc trapping mechanisms is a considerable disadvantage for the trap-driven simulators, so the software-based tools are likely to be more popular in the immediate future. If, however, future machines begin to provide better support for controlling memory access in a fine-grained manner, trap-driven simulation could become more attractive. Such support is not necessarily expensive, and could be useful for other applications as well, such as distributed shared memory [Reinhardt et al. 1996].

9. SUMMARY

Trace-driven simulation has played an important role in the design of memory systems in the past, and because of the increasing processor memory speed gap its usefulness is likely to continue growing in the future. This survey has defined several criteria to use when judging the features of a trace-driven simulation system, and has come 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 multiconfiguration simulations include more overhead than typically reported. Most important, no single method is best when all points of comparison, including speed, accuracy, flexibility, expense, portability, and ease of use, are taken into consideration.

Perhaps the most important factor to keep in mind when selecting the components of a complete trace-driven memory simulator is balance. Research in trace-driven simulation frequently places too much emphasis on one aspect of the process (e.g., speed) at the expense of others (e.g., completeness or portability). In the quest for raw speed, a simulator writer might, for example, be tempted to select a static code annotator over an instruction-set emulator because the former is typically twice as fast as the latter for collecting address traces. When trace-processing times are taken into account, however, this difference may make a negligible contribution to overall slowdowns and may not be worth the flexibility and ease of use that annotators sacrifice to obtain their speed advantage over emulators. Similarly, the results obtained from the fastest known cache simulator may not be of much value if they can only be used to study single-process workloads. A slower, but more complete system, capable of capturing multiprocess and operating system activity, may often be the better choice.

Looking forward, we can expect to see continued changes in the way that memory system simulation is performed. The biggest change is likely to come in the contents of the traces themselves. As we saw in Section 8, there is much to be gained by moving beyond a simple sequential trace interface in which each and every memory reference is passed from trace collector to trace processor. Richer trace interfaces will result not only in faster simulation times, but may become a necessity to enable accurate simulations of tomorrow's complex microprocessors, which will be capable of making out of order, nonblocking accesses to the memory system.

ACKNOWLEDGMENTS

Many thanks to Stuart Sechrest, Peter Bird, Peter Honeyman, David Kaeli, and Mike Smith, as well as the anonymous reviewers from *Computing Surveys* for their helpful comments on earlier versions of this article. A special thanks to Andre Seznec and IRISA for supporting this work during its final stages.

REFERENCES

AGARWAL, A. 1989. Analysis of cache performance for operating systems and multiprogramming. Ph.D. Dissertation, Stanford.

AGARWAL, A. AND HUFFMAN, M. 1990. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Boulder, CO), ACM, New York, 48–57.

AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. 1988. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. 6,* 4, 393–431.

AGARWAL, A., HOROWITZ, M., AND HENNESSY, J. 1989. An analytical cache model. *ACM Trans. Comput. Syst. 7,* 2, 184–215.

AGARWAL, A., SITES, R. L., AND HOROWITZ, M. 1986. ATUM: A new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture* (Tokyo, Japan), IEEE, New York, 119–127.

ALEXANDER, C. A., KESHLEAR, W. M., AND BRIGGS, F. 1985. Translation buffer performance in a UNIX environment. *Comput. Arch. News 13,* 5, 2–14.

ALEXANDER, C., KESHLEAR, W., COOPER, F., AND BRIGGS, F. 1986. Cache memory performance in a UNIX environment. *Comput. Arch. News 14,* 14–70.

AMD 1991. *Am29050 Microprocessor User's Manual.* Advanced Micro Devices, Inc., Sunnyvale, CA.

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

BAKER, M. 1995. Cluster computing review. Northeast Parallel Architectures Center (NPAC) Techn. Rep. SCCS-748, Nov.

BECKER, J. AND PARK, A. 1993. 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.

BEDICHEK, R. 1994. The meerkat multicomputer: Tradeoffs in multicomputer architecture. Ph.D. Dissertation, University of Washington Department of Computer Science Techn. Rep. 94-06-06, Aug.

BEDICHEK, R. 1995. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 14–24.

BIOMATION 1991. Biomation CLAS 4000 application note 4032. Biomation, Cupertino, CA.

BORG, A., KESSLER, R., LAZANA, G., AND WALL, D. 1989. Long address traces from RISC machines: Generation and analysis. DEC Western Research Lab Tech. Rep. 89/14.

BORG, A., KESSLER, R., AND WALL, D. 1990. Generation and analysis of very long address traces. In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* IEEE, New York.

CHEN, B. 1993. Software methods for system address tracing. In *Proceedings of the Fourth Workshop on Workstation Operating Systems* (Napa, CA).

CHEN, B. 1994. Memory behavior of an X11 window system. In *Proceedings of the USENIX Winter 1994 Technical Conference.*

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

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

Cmelik, R. and Keppel, D. 1993. Shade: A fast instruction-set simulator for execution profiling. University of Washington Tech. Rep. UWCSE 93-06-06.

Cmelik, B. and Keppel, D. 1994. 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, New York, 128–137.

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

Cvetanovic, Z. and Bhandarkar, D. 1994. Characterization of Alpha AXP performance using TP and SPEC Workloads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (Chicago, IL), IEEE, New York.

Davies, P., Lacroute, P., Heinlein, J., and Horowitz, M. 1994. Mable: A technique for efficient machine simulation. Stanford University Tech. Rep. CSL-TR-94-636, Oct.

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

Digital 1986. *VAX Architecture Handbook.* Digital Equipment Corporation, Bedford, MA.

Digital 1992. *Alpha Architecture Handbook.* Digital Equipment Corporation, Bedford, MA.

Eggers, S., Keppel, D., Koldinger, E., and Levy, H. 1990. 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.

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

Eustace, A. and Srivastava, A. 1995. ATOM: A flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX Winter 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans, LA), Jan., 303–314.

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

Flanagan, J. K. 1994. Personal communication.

Fuentes, C. 1993. Hardware support for operating systems. University of Michigan Unpublished report.

Gee, J., Hill, M., Pnevmatikatos, D., and Smith, A. J. 1993. Cache performance of the SPEC92 benchmark suite. *IEEE Micro* (Aug.), 17–27.

Goldschmidt, S. and Hennessy, J. 1992. The accuracy of trace-driven simulation of multiprocessors. Stanford University Computer Systems Laboratory Tech. Rep. CSL-TR-92-546, Sept.

Goldschmidt, S. and Hennessy, J. 1993. The accuracy of trace-driven simulation of multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), 146–157.

Grimsrud, K. 1993. Address translation of BACH i486 traces. Brigham Young University Tech. Rep.

Hammerstrom, D. and Davidson, E. 1977. Information content of CPU memory referencing behavior. In *Proceedings of the Fourth International Symposium on Computer Architecture,* 184–192.

Happel, L. P. and Jayasumana, A. P. 1992. Performance of a RISC machine with two-level caches. *IEEE Proceedings-E 139,* 3, 221–229.

Hennessy, J. L. and Patterson, D. A. 1996. *Computer Architecture, A Quantitative Approach.* 2nd ed. Morgan Kaufmann, San Mateo, CA.

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

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

Hill, M. 1987. Aspects of cache memory and instruction buffer performance. Ph.D. Dissertation, The University of California at Berkeley.

Hill, M. and Smith, A. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38,* 12, 1612–1630.

Hodges, J. L. and Lehmann, E. L. 1964. *Basic*

Concepts of Probability and Statistics. Holden-Day, Inc., San Francisco, CA.

HOLLIDAY, M. AND ELLIS, C. 1990. Accuracy of memory reference traces of parallel computations in trace-driven simulation. Department of Computer Science, Duke University, Durham, NC. Tech. Rep. CS-1990-8.

HOLLIDAY, M. 1991. Techniques for cache and memory simulation using address reference traces. *Int. J. Comput. Simul. 1,* 129–151.

IBM 1990. *IBM RISC System/6000 Technology.* IBM, Austin, TX.

INTEL 1990. *i860 64-bit Microprocessors Programmer's Manual.* Intel Corporation, Santa Clara, CA.

JOUPPI, N. 1990. 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, New York, 364–373.

KANE, G. AND HEINRICH, J. 1992. *MIPS RISC Architecture.* Prentice-Hall, Englewood Cliffs, NJ.

KAELI, D. 1991. Issues in trace-driven simulation. In *Proceedings of the 22nd Annual Pittsburgh Modeling and Simulation Conference,* Vol. 22, Part 5, May, 2533–2540.

KESSLER, R. 1991. Analysis of multi-megabyte secondary CPU cache memories. Ph.D. Dissertation, University of Wisconsin-Madison.

LACY, F. 1988. An address trace generator for trace-driven simulation of shared memory multiprocessors. University of California at Berkeley. Tech. Rep. UCB/CSD 88/407.

LAHA, S., PATEL, J., AND IYER, R. 1988. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Trans. Comput. 37,* 11, 1325–1336.

LARUS, J. R. 1990. Abstract execution: A technique for efficiently tracing programs. *Softw. Pract. Exper. 20,* 12 (Dec.) 1241–1258.

LARUS, J. 1991. SPIM S20: A MIPS R2000 simulator. University of Wisconsin-Madison Tech. Rep., Revision 9.

LARUS, J. R. 1993. Efficient program tracing. *IEEE Computer* (May), 52–60.

LEBECK, A. AND WOOD, D. 1994. Fast-Cache: A new abstraction for memory-system simulation. University of Wisconsin-Madison Tech. Rep. 1211.

LEBECK, A. AND WOOD, D. 1995. 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), 220–230.

LEE, C.-C. 1997. A case study of a hardware-managed TLB in a multi-tasking environment. University of Michigan Tech. Rep.

MACWEEK STAFF 1994. Apple holds up 603 for cache. *MacWeek 8,* 21 (May 24), 1, 100.

MAGNUSSON, P. A. 1993. 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 Telecommunications Systems (MASCOTS-93)* (La Jolla, CA) 69–78.

MARTONOSI, M. 1994. *Analyzing and tuning memory performance in sequential and parallel programs.* Ph.D. Dissertation, Stanford University.

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. 1992. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems,* ACM, New York.

MARTONOSI, M., GUPTA, A., AND ANDERSON, T. 1993. 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, CA), ACM, New York, 248–259.

MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9,* 2, 78–117.

MAY, C. 1987. Mimic: A fast S/370 simulator. In *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques* (St. Paul, MN), ACM, New York, 1–13.

MAYNARD, A. M., DONNELLY, C., AND OLSZEWSKI, B. 1994. 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, New York, 145–156.

MIPS. 1988. *RISCompiler Languages Programmer's Guide.* MIPS.

MOGUL, J. C. AND BORG, A. 1991. The effect of context switches on cache performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA), ACM, New York, 75–84.

MOTOROLA 1990. *MC88100 RISC Microprocessor User's Manual.* Prentice-Hall, Englewood Cliffs, NJ.

MOTOROLA 1993. *Power PC 601 RISC Microprocessor User's Manual. Motorola, Inc.*

NAGLE, D., UHLIG, R., AND MUDGE, T. 1992. Monster: A tool for analyzing the interaction between operating systems and computer architectures. University of Michigan Tech. Rep. CSE-TR-147-92.

NAGLE, D., UHLIG, R., STANLEY, T., SECHREST, S., MUDGE, T., AND BROWN, R. 1993. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, CA), IEEE, 27–38.

NAGLE, D., UHLIG, R., MUDGE, T., AND SECHREST, S. 1994. Optimal allocation of on-chip memory for multiple-API operating systems. In *Proceedings of the 21st International Symposium on Computer Architecture* (Chicago, IL).

PIERCE, J. AND MUDGE, T. 1994a. IDtrace—A tracing tool for i486 simulation. University of Michigan Tech. Rep. CSE-TR-203-94.

PIERCE, J. AND MUDGE, T. 1994b. 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.

PIERCE, J., SMITH, M. D., AND MUDGE, T. 1995. Instrumentation tools. In *Fast Simulation of Computer Architectures,* (T. M. Conte and C. E. Gimarc, Eds.), Kluwer Academic Publishers, Boston, MA (to appear).

PLESZKUN, A. 1994. Techniques for compressing program address traces. Tech. Rep., Department of Electrical and Computer Engineering, University of Colorado-Boulder.

PRIEVE, B. G. 1974. *A Page Partition Replacement Algorithm.* University of California at Berkeley.

PUZAK, T. 1985. Analysis of cache replacement algorithms. Ph.D. Dissertation, University of Massachusetts.

REINHARDT, S., HILL, M., LARUS, J., LEBECK, A., LEWIS, J., AND WOOD, D. 1993. 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, New York, 48–60.

REINHARDT, S., PFILE, R., AND WOOD, D. 1996. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* (to appear).

ROMER, T., LEE, D., VOELKER, G., WOLMAN, A., WONG, W., BAER, J., BERSHAD, B., AND LEVY, H. 1996. The structure and performance of interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, Oct.), (to appear).

ROSENBLUM, M., HERROD, S., WITCHEL, E., AND GUPTA, A. 1995. Complete computer simulation: The SimOS approach. *IEEE Parallel Distrib. Technol.* (Fall).

SAMPLES, A. 1989. Mache: No-loss trace compaction. In *Proceedings of 1989 SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* ACM, New York, 89–97.

SITES, R. L. AND AGARWAL, A. 1988. Multiprocessor cache analysis with ATUM. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Honolulu), IEEE, New York, 186–195.

SITES, R., CHERNOFF, A., KIRK, M., MARKS, M., AND ROBINSON, S. 1992. Binary translation. *Digital Tech. J. 4,* 4, 137–152.

SMITH, A. J. 1977. Two methods for the efficient analysis of memory address trace data. *IEEE Trans. Softw. Eng. SE-3*, 1, 94–101.

SMITH, A. J. 1982. Cache memories. *Comput. Surv. 14,* 3, 473–530.

SMITH, A. 1986. Bibliography and readings on CPU cache memories and related topics. *Comput. Arch. News 14,* 22–42.

SMITH, M. D. 1991. Tracing with pixie. Tech. Rep., Stanford University, Stanford, CA.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* (June), 196–205.

STEPHENS, C., COGSWELL, B., HEINLEIN, J., PALMER, G., AND SHEN, J. 1991. Instruction level profiling and evaluation of the IBM RS/6000. In *Proceedings of the 18th Annual International Symposium on Computer Architecture* (Toronto, Canada), ACM, New York, 180–189.

STONE, H. 1993. *High-Performance Computer Architecture.* Addison-Wesley, Reading, MA.

STUNKEL, C. AND FUCHS, W. 1989. 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, New York, 70–78.

STUNKEL, C., JANSSENS, B., AND FUCHS, W. K. 1991. Collecting address traces from parallel computers. In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences* (Hawaii), 373–383.

SUGUMAR, R. 1993. Multi-configuration simulation algorithms for the evaluation of computer designs. Ph.D. Dissertation, University of Michigan.

TALLURI, M. AND HILL, M. 1994. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA), ACM, New York.

TEKTRONIX. 1994. *Test and Measurement Product Catalog.* Wilsonville, OR.

THEKKATH, C. AND LEVY, H. 1994. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth Interna-*

tional Conference on Architectural Support for Programming Languages and Operating Systems (San Jose, CA), ACM Press, New York, 110–119.

THOMPSON, J. AND SMITH, A. 1989. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Trans. Comput. Syst. 7,* 1, 78–116.

TORRELLAS, J., GUPTA, A., AND HENNESSY, J. 1992. Characterizing the caching and synchronization performance of multiprocessor operating system. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA), 162–174.

UHLIG, R., NAGLE, D., MUDGE, T., AND SECHREST, S. 1994. Trap-driven simulation with Tapeworm II. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA), ACM Press (SIGARCH), New York, 132–144.

UHLIG, R., NAGLE, D., MUDGE, T., SECHREST, S., AND EMER, J. 1995. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, June) (to appear).

UPTON, M. D. 1994. Architectural trade-offs in a latency tolerant gallium arsenide microprocessor. Ph.D. Dissertation, The University of Michigan.

VEENSTRA, J. AND FOWLER, R. 1994. MINT: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS),* 201–207.

WALL, D. 1989. Link-time code modification. DEC Western Research Lab Tech. Rep. 89/17.

WALL, D. 1992. Systems for late code modification. DEC Western Research Lab. Tech. Rep. 92/3.

WANG, W.-H. AND BAER, J.-L. 1990. 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, New York, 27–36.

WIECEK, C. A. 1982. 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.

WILKES, M. 1969. The growth of interest in microprogramming: A literature survey. *Comput. Surv. 1,* 3 (Sept.), 139–145.

WINSOR, D. 1989. Bus and cache memory organizations for multi-processors. Ph.D. Dissertation. University of Michigan.

WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Philadelphia, May).

WOOD, D., HILL, M., AND KESSLER, R. 1991. A model for estimating trace-sampled miss ratios. In *Proceedings of the 1991 SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 79–89.

ZIV, J. AND LEMPEL, A. 1976. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23,* 75–81.

Received November 1995; accepted December 1996