

Trace-Driven Memory Simulation: A Survey^{*}

Richard A. Uhlig¹ and Trevor N. Mudge²

¹ Intel Microcomputer Research Lab (MRL), Hillsboro, Oregon

² Advanced Computer Architecture Lab (ACAL), Electrical Engineering and Computer Science Department, The University of Michigan, Ann Arbor, Michigan

Abstract. 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 paper 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.

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) 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 1970's, would have increased its workload run times by a factor of only 1.6 (Jouppi 1990), while disabling the cache of the HP 9000/735, a more recent machine introduced in the early 1990's, 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

^{*} 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.

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 ten 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. This chapter is an abridged version of Uhlig and Mudge (1997).

2 Scope, Related Surveys and Organization

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

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 reviewed before. This survey is broader in scope than the surveys by Pierce et al. and Cmelik et al., 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 multi-process 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. 1989b). Although trace-driven methods have been successfully applied to other domains of computer architecture, such as the simulation of super-scalar processor architecture, or the design of I/O systems, this survey will focus 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. While 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 main body of text. This organization enables a 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 Fig. 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 con-

nected through trace interfaces so that trace data can flow from one stage to the next.

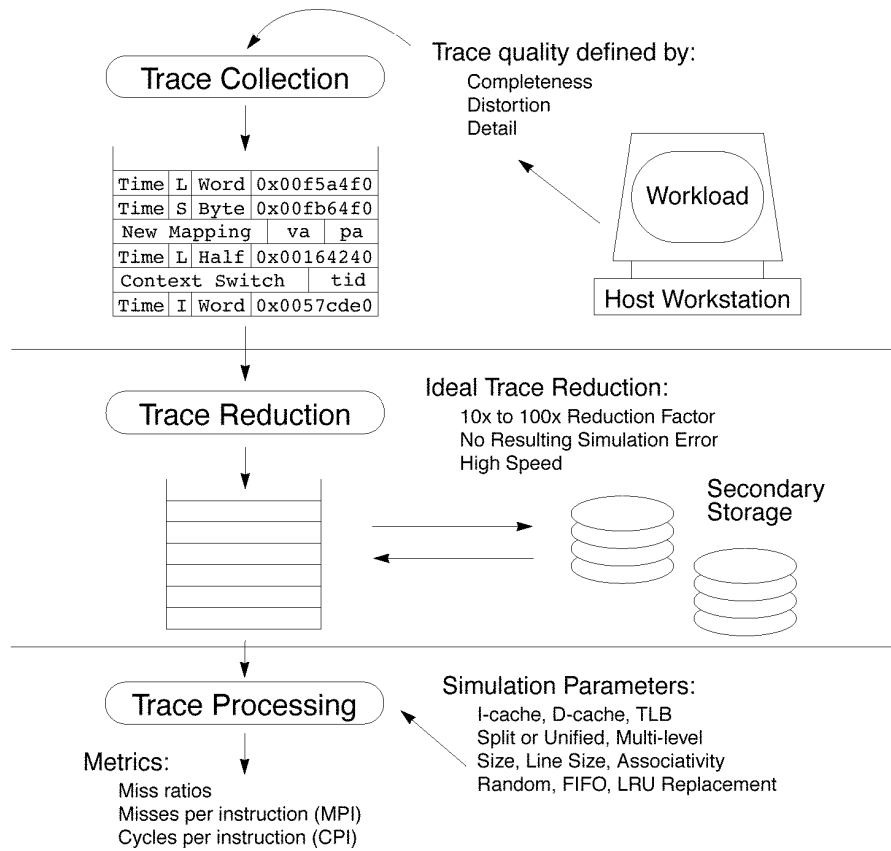


Fig. 1. The three stages of trace-driven simulation

In Sections 3–7, we shall examine each of the above 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

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

some performance metric such as miss ratio or misses per instruction:

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

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 are 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 shall 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 Bershady 1993; Nagle et al. 1993; Nagle et al. 1994).

When we look at trace reduction in Section 5 we will 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 (1) to measure the effects:

$$\text{Error} = \left[\frac{(\text{Measurements with Full Trace} - \text{Measurements with Reduced Trace})}{(\text{Measurements with Full Trace})} \right] \cdot 100\% . \quad (2)$$

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 non-blocking 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 from or processed on. Depending on the context, we compute slowdowns in a variety of ways:

$$\text{Slowdown} = \frac{\text{Address Collection Rate}}{\text{Host System Address Generation Rate}} \quad (3)$$

$$\text{Slowdown} = \frac{\text{Address Processing Rate}}{\text{Host System Address Generation Rate}} \quad (4)$$

$$\text{Slowdown} = \frac{\text{Total Simulation Time}}{\text{Normal Host System Execution Time}} \quad (5)$$

Because each of these definitions divide 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 will examine can reduce overall slowdowns. We report their effectiveness in terms of *speedups*, which divide slowdowns to obtain overall slowdowns:

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

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 have substantially different memory requirements to begin with. 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:

$$\text{Memory Overhead} = \frac{\text{Additional Memory Required}}{\text{Normal Host Memory Required}} \quad (7)$$

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 it passes 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*:

$$\text{Reduction Factor} = \frac{\text{Full Address Trace Size}}{\text{Reduced Address Trace Size}} \quad (8)$$

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, half word, 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 occurs 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 above, 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 compiler and operating-system levels (see Fig. 2). We organize the remainder of this section accordingly, starting at instruction-set emulation. For more details on microcode modification and external hardware probes see Uhlig and Mudge (1997).

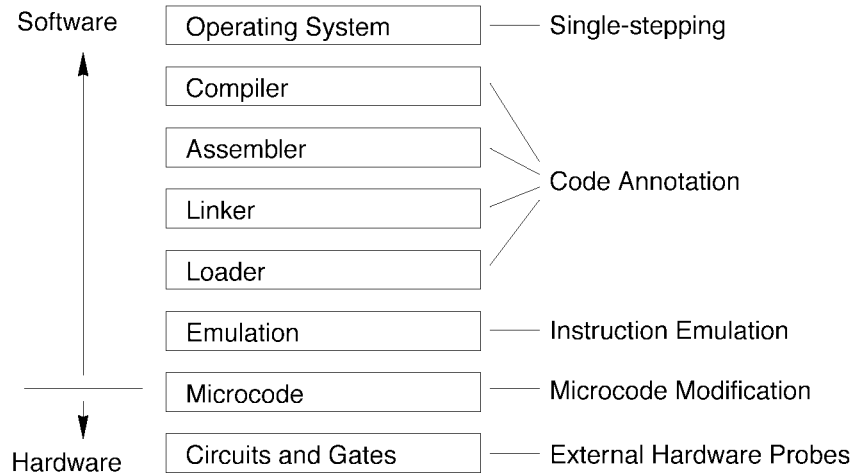


Fig. 2. Levels of system abstraction and trace collection methods

4.1 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 (Agarwal 1989a; Wall 1989; Borg et al. 1989; Stunkel et al. 1991; Flanagan et al. 1992). 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 above. 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 1).

Table 1. Instruction-Set Emulators that Support Trace Collection. An instruction-set emulator is a program that directly reads executable images written in one ISA (the *target*) and emulates it using another ISA (the *host*). In general, the target and host ISAs need not be the same, although they may be. We only consider instruction-set emulators that also generate address traces (for a more complete survey of instruction-set emulators in general, see Cmelik and Keppel (1993) and Cmelik and Keppel (1994)). The leftmost column (*Method*) indicates the general method used by the emulator (see Fig. 3), but it should be noted that not all emulators fit neatly into one category or the other. The table includes additional characteristics that help to define the methods used by these emulators. *Register State*, for example, can be held either by the registers of the host machine, in memory (as part of the emulator's data structures), or in both places via a *hybrid* scheme. When emulators predecode or translate target instructions, some do so *all-at-once*, when the workload begins executing, while others use a *lazy* policy, predecoding or translating when an instruction is first executed. Finally, some emulators attempt to reduce the overhead of the dispatch loop by clustering groups of instructions together by *chaining* or *threading* individual instructions together. The same effect can be achieved by translating entire *blocks* of instructions at a time. *Note*: slowdowns may include additional overhead that is not strictly required for collecting address traces.

Method	Reference	Name	Target(s)	Host(s)	Other Characteristics			
					Register State Held In	Predecode/Translation Policy	Chain, Thread or Block	Slowdown
Iterative Interpretation	Cmelik & Keppel (1993)	Spa (Spy)	SPARC	SPARC	Host Registers	N/A	No	40–600
	Davies et al. (1994)	Mable	MIPS-I, MIPS-III	MIPS-I	Memory	N/A	No	20–200
Predecode Interpretation	Larus (1991)	SPIM	MIPS-I	SPARC, 680x0, MIPS, x86, HP-PA	Memory	All-at-once	No	25
	Magnusson (1993)	gsim	88100	HP-PA, SPARC	Memory	Lazy	Threading	45–75
Dynamic Translation	Bedichek (1995)	Talisman	88100	SPARC	Memory	Lazy	Threading	100–150
	Veenstra & Fowler (1994)	MINT	R3000	R3000	Hybrid	All-at-once	Block	20–70
Dynamic Translation	Cmelik and Keppel (1994)	Shade	SPARC-V8,	SPARC-V8	Memory	Lazy	Chaining	9–14
			SPARC-V9, MIPS					

Spa (Cmelik and Keppel 1993) and *Mable* (Davies et al. 1994) are examples of emulators that use straightforward iterative interpretation (see top of Fig. 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 instruction has been decoded, it is emulated (*dispatched*) by updating 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,² 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 slowdowns 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 Fig. 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 lookup 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 di-

² *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, while 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.

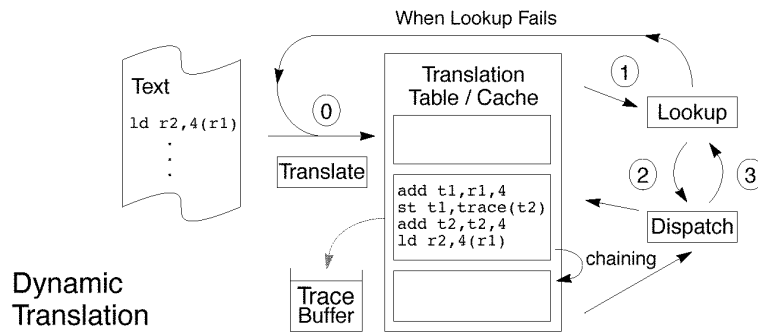
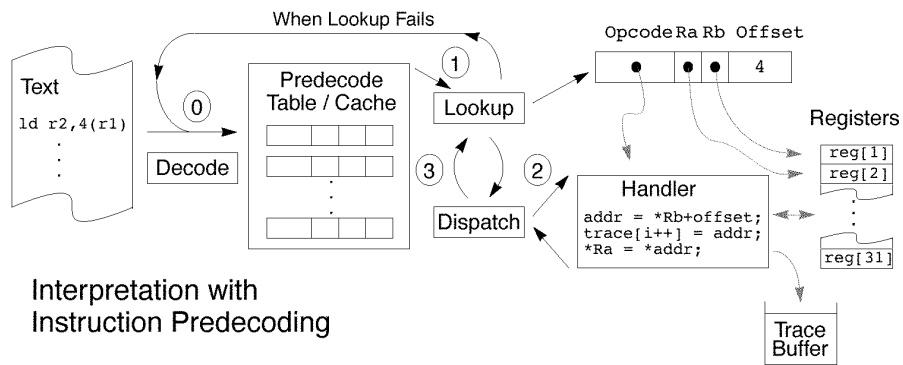
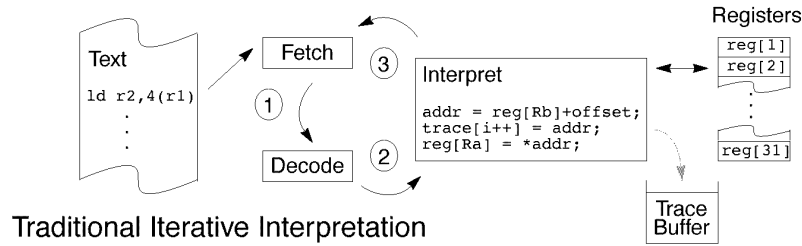


Fig.3. Some emulation methods. Traditional emulators fetch, decode and interpret each instruction from a workload's text segment in an iterative loop (top figure). To avoid the cost of re-decoding instructions each time they are encountered, some faster emulators pre-decode instructions and store them in a table for rapid lookup and dispatch (middle figure). A further optimization is to translate target instructions from the emulated workload into equivalent sequences of host instructions that can be executed directly (bottom figure). In all three cases, code can be added to emit addresses into a trace buffer as the workload is emulated

rectly 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, 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 handlers 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 20–70 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 Fig. 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 *gim* 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 1 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, in indeed, be effectively used 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 multi-threaded applications and emulation of operating system code, but this statement should be interpreted carefully. All of these emulators run in their own user-level 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 paper was written (Cmelik and Keppel 1993; Cmelik and Keppel 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 1, 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; Cmelik and Keppel 1994). Increased memory usage means that these systems must be equipped with additional physical memory to handle large workloads.

4.2 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 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 Fig. 4). Static code annotation can be performed at the source (assembly) level, the object-module level, or the executable (binary) level (see Fig. 2 and Ta-

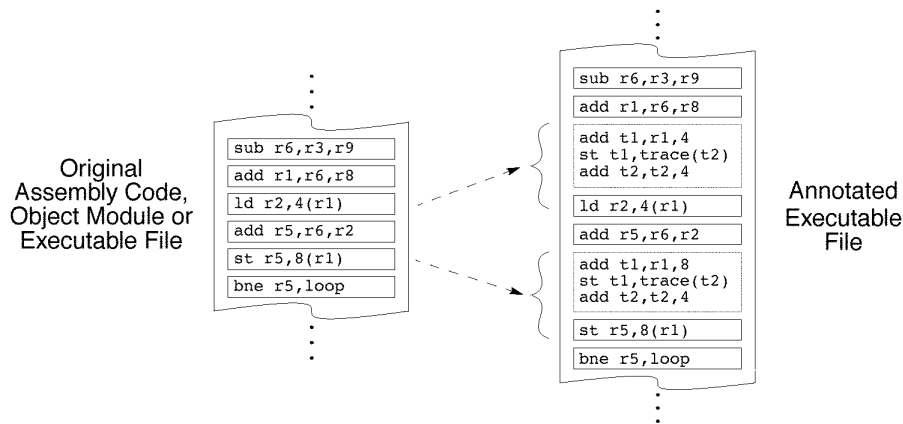


Fig. 4. Static code annotation. In this example, memory references made by a workload are traced by inserting instructions ahead of each load and store operation in the annotated executable file. The inserted code computes the load or store address in register `t1`, saves it in a trace buffer, and then increments the trace buffer index, which is held in register `t2`. Notice that registers `t1` and `t2` are assumed to be not live during this fragment of code. If the code annotator is unable to determine, via static analysis, whether this assumption is true, then it may be forced to temporarily save and restore these registers to memory, thus increasing the size of the annotation code. Annotations can also be inserted at the beginnings of basic blocks to trace instruction-memory references

ble 2), with different consequences for both the implementation and the end user (Stunkel et al. 1991; Wall 1992; Pierce and Mudge 1994a).

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

Table 2. Static code annotators. Code-annotation tools add instructions to a program at the *Source*, *Object* or *Binary* level to create an *annotated* program executable file that outputs address traces as a side effect of its execution. In the table below, *Slowdown* refers to the time it takes both to run the annotated program and to produce the full address trace, while *Time Dilation* refers only to the time it takes to run the annotated program. Usually these are the same, but some annotated programs generate only a minimal trace of significant events which must be post-processed to reconstruct the full trace. *Memory Dilation* refers to the additional space used by the annotated program relative to an un-annotated program.

Method	Reference	Name	Slow-down	Time Dilation	Memory Dilation	Completeness		Processor	Analyzer Interface
						Multi-process	OS Kernel		
Source	Stunkel & Fuchs (1989)	TRAPEDS	20-30	20-30	8-10	No	No	iPSC/2	Linked into Process
	Eggers et al. (1990)	MPtrace	1,000+	2-3	4-6	No	No	i386	File + Post Process
	Larus (1990)	AE	20-65	2-5	—	No	No	MIPS, SPARC	File + Post Process
	Goldschmidt & Hennessy (1993)	TangoLite	45	45	4	No	No	MIPS	Memory Buffers
Object	Borg et al. (1989)	Epoxie	8-12	8-12	5	Yes	No ^a	Titan	Global Buffer
	Chen (1993)	Epoxie2	15	15	2	Yes	Yes	R3000	Global Buffer
	Srivastava & Eustace (1994); Eustace & Srivastava (1994)	ATOM	6-13	6-13	—	No	Yes	Alpha	Linked into Process
	Smith (1991)	Pixie	10	10	4-6	No	No	MIPS	File / Pipe
Binary	Stephens et al. (1991)	Goblin	20	20	10	No	No	RS/6000	Linked into Process
	Pierce & Mudge (1994a)	IDtrace	12	12	12	No	No	i486	File / Pipe
	Larus (1993)	Qpt	10-60	2-5	3	No	No	MIPS, SPARC	File + Post Process
	Larus (1995)	EEL	—	—	—	No	No	MIPS, SPARC	—

^aKernel tracing was implemented, but was not fully debugged.

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 easily be 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 post-processing program that reconstructs the full trace. 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 multi-processor 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 argues that annotating code at this level is only slightly more difficult because data-relocation tables and symbol tables are still available (Wall 1992). An early example of this form of code annotation is *Eporie*, implemented for the DEC Titan (Borg et al. 1989; Borg et al. 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 ranges 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 1994a). 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 1994a), 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 compares 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 report that addresses for cache simulation can be collected from *ATOM*-annotated SPEC92 benchmarks with a slowdowns of between 6 and 13 (Eustace and Srivastava 1994). *EEL* is a similarly-flexible executable editor that is the basis of a new version of *qpt* as well a high-speed cache simulator named *Fast-cache* (Lebeck and Wood 1995), which we will discuss in Section 8.

In general, code annotators are not capable of monitoring multi-process³ workloads or the operating system kernel, but there are some exceptions. Borg and Mogul describe modifications to the Titan operating system, *Tunix*, that support tracing of multiple workload processes by *Epoixie* (Borg et al. 1989; Borg et al. 1990; Mogul and Borg 1991). *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

³ Many of the tracing tools discussed on this section were designed to monitor multi-threaded workloads running on a multi-processor 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.

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 1993; Chen 1994). 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 system-wide, multi-process 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 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 multi-process 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 multi-process 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 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) (Borg et al. 1989; Chen 1993).

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, like Shade. Second, all of these tools suffer from the disadvantage that all workload components must be prepared prior to being run. Usually this is not a major concern, but it can be a time-consuming and tedious process if a workload consists of several source or object files. Even for the tools that avoid source or object-file annotation, it can be difficult to locate all of the executables that make up a complex multi-process 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).

4.3 Summary of Trace Collection

Table 3 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 complete 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

Table 3. Summary of Trace-Collection Methods. This table summarizes the characteristics of five common methods for collecting address traces. For the descriptions of trace quality (*completeness*, *detail* and *distortions*) a *Maybe* entry means that the method has inherent difficulty providing data with the given characteristics, but there are examples of tools in the given category that overcome these limitations. The ranges given in the *slowdown* row exclude times for excessively bad implementations.

	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
Detail	Dynamically-Linked Code	Yes	Yes	Yes	Maybe	No
	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 Dilatation	No	10-20	No	2-30	N/A
	Memory Dilatation	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	

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 (multi-process 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 applicability 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 multi-process 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 4). 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 4 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. Ta-

ble 4 shows when and how these assumptions imply restrictions on the use of the reduced trace. For more details see Uhlig and Mudge (1997).

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 K-bytes to 64 K-bytes (in powers of two), line sizes ranging from 1 word to 16 words (in powers of two), and associativities ranging from 1-way to 4-way, contains 100 possible 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 multi-level hierarchy. Because of these interactions and because different memory components often compete for scarce resources such as chip-die area, the different components cannot be considered in isolation. This leads to a further, combinatorial expansion of the design space. Researchers have explored two basic approaches to dealing with this problem: (1) parallel distributed simulations, and (2) multi-configuration 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 under-utilized 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 desk, it is useful to have a method for suspending or moving a compute-intensive simulation task that has been started on his 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 a multi-configuration simulation algorithms in this survey (see Table 5). First, it is de-

Table 4. Methods for Address Trace Reduction. The trace reduction factor is the ratio of the sizes of the reduced trace and the full trace. *Decompression Slowdown* is only relevant to methods that reconstruct the full trace before it is processed. Most of these methods pass the reduced trace directly to the trace processor which is able to process this data much faster than the full trace (see *Simulation Speedup*). Simulations with a reduced trace usually result in some simulation error and can be performed only in a restricted design space (see *Exact, Error and Restrictions*).

Method	Reference	Reduction Factor	Decompression Slowdown	Simulation Speedup	Exact?	Error	Restrictions
Trace Compression	Samples (1989)	10-100	100-200	1	Yes	N/A	None
Significant-Event Traces	Larus (1990);	10-40	20-60	1	Yes	N/A	None
	Larus (1993)						
	Eggers et al. (1990)	—	1,000+	1	Yes	N/A	None
Stack Deletion Filter	Smith (1977)	5-100	0	4-50	No	< 4-5%	Fully-associative Memories
Snapshot Filter	Smith (1977)	5-100	0	4-50	No	< 4-5%	Fully-associative Memories
Cache Filter	Puzak (1985)	10-20	0	—	Yes	N/A	Fixed-line-size Caches
	Wang and Baer (1990)	10-20	0	7-15	Yes	N/A	Fixed-line-size Caches
Block Filter	Agarwal and Huffman (1990)	50-100	0	—	No	< 12%	Fixed-line-size Caches
Time Sampling	Laha et al. (1988)	5-20	0	< 5-20	No	< 5%	Small Caches (< 128 K-byte)
	Kessler (1991)	10	0	< 10	No	< 10%	Small Caches (< 1 M-byte)
Set Sampling	Puzak (1985)	5-10	0	< 10	No	< 2%	Set Sample Not General
	Kessler (1991)	10	0	< 10	No	< 10%	Constant-bits Set Sample

sirable 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 per instruction (CPI). The *overhead* of performing a multi-configuration 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-configurations simulations.

6.1 Stack Processing

Mattson et al. 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 (Mattson et al. 1970). 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 an least-recently-used (LRU) replacement policy includes all of the contents of a similar cache with only $(n - 1)$ entries.

When inclusion holds, a range of different-sized, fully-associative caches can be represented as a stack as shown in Fig. 5. 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.

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:

$$hits_n = \sum_{i=0}^{n-1} hits[i] . \quad (9)$$

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:

$$misses_n = totalReferences - hits_n , \quad (10)$$

$$missRatio_n = misses_n / totalReferences , \quad (11)$$

$$MPI_n = misses_n / totalInstructions . \quad (12)$$

Table 5. Multi-configuration Memory Simulators. Multi-configuration memory simulators can determine the performance for a range of memory configurations in a single pass of an address trace. Each of these simulators is, however, limited in the way that memory-configuration parameters can be varied (see *Range of Parameters*) or in the performance metrics that they can produce (see *Metrics*). Most multi-configuration algorithms cannot vary total cache size directly. Instead, they vary the number cache sets or associativity, and thus vary total cache size as determined by the equation: $Size = Sets * Assoc * Line$. *Overhead* is the extra time that it takes to perform a multi-configuration simulation relative to a single-configuration simulation (as reported by the authors of each simulator). This overhead is usually an underestimate of the true processing overhead because values reported in papers typically do not include the time to read input traces from a file.

Reference	Name	Range of Parameters			Metrics	Overhead		
		Sets	Line Assoc	Write Policy				
				Sector				
Mattson et al. (1970)	Stack Processing	Fixed	Fixed	Vary	None	No	Misses, Miss Ratio, MPI	—
Hill (1987)	Forest Simulation	Vary	Fixed	1-way	None	No	Misses, Miss Ratio, MPI	< 5%
Hill (1987)	All-Associativity	Vary	Fixed	Vary	None	No	Misses, Miss Ratio, MPI	< 30%
Thompson and Smith (1989)	—	Fixed	Fixed	Vary	W-back	Yes	Misses, Write Backs	< 100%
Wang and Baer (1990)	—	Vary	Fixed	Vary	W-back	No	Misses, Write Backs	< 65%
Sugumar (1993)	Cheetah	Fixed	Vary	1-way	W-thru	No	Misses, WB Stalls	< 120%

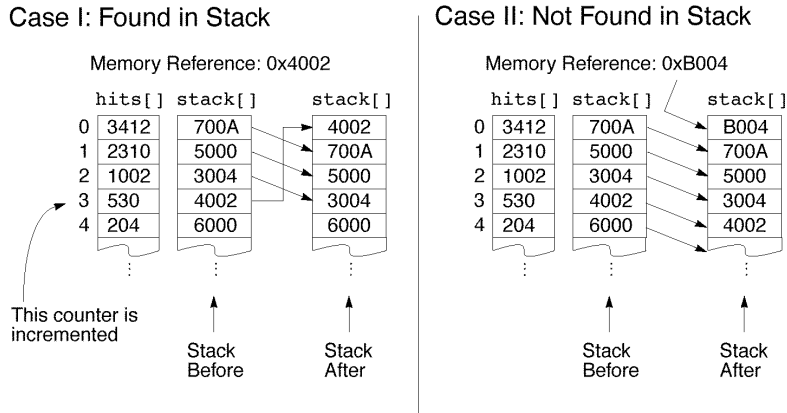


Fig. 5. Data structures for stack simulation. In Case I, the address is found at stack depth 3, so the hits[3] counter is incremented, and the entry at this depth is pulled to the top of the stack. In Case II, the address is not in the stack, so it is pushed onto the top, and no counter is incremented

Mattson et al. 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-on 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 multi-configuration simulation algorithms. Many researchers have advanced multi-configuration simulation by proposing various enhancements and by reporting simulation times for actual implementations of these improvements. We focus on a selection of recent papers that extend the range of multi-configuration parameters, and that characterize the current state-of-the-art in this form of simulation (see Table 5).

6.2 Forest and All-Associativity Simulation

Hill noted that the original stack algorithm of Mattson et al. requires the number of cache sets and the line size to be fixed (Hill 1987). 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 multi-configuration 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 developed extensions that count the number of writes to main memory for different-sized caches that implement a write-back

write policy (Thompson and Smith 1989). They also studied multi-configuration algorithms for sector or sub-block caches. Wang and Baer combined the work of Mattson et al. (1970), Hill (1989) and Thompson and Smith (1989) to compute both miss ratios and write backs in a range of caches where the both the number of sets and the associativity is varied. In his dissertation, Sugumar developed algorithms for varying line size with direct-mapped caches of a fixed size, and also for computing write-through stalls and write traffic in a cache with a coalescing write buffer (Sugumar 1993).

6.3 Summary of Trace Processing

There are several points to be made about multi-configuration algorithms in general. First, for all of the examples considered, the overhead of simulating multiple configurations in one trace pass is reported to be less than 100%, which means that one multi-configuration simulation of two or more configurations would perform as well as or better than collections of two or more single-configuration simulations. 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 multi-configuration is very low. If, however, the trace input times are relatively low, then the multi-configuration overheads will be much higher. This is the case with the Sugumar's *Cheetah* simulator which appears to have very high overheads relative to Hill's *Tycho* simulator (Hill 1987; Sugumar 1993) (see Table 5). *Cheetah*'s overall simulation times are, however, approximately eight 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 multi-configuration algorithms several times to cover an entire design space. Hill gives an example design space of 24 caches, with a range of sizes, line sizes and associativities where the minimal number of trace passes required by stack simulation is 15 (Hill 1987). 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 multi-configuration 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've 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 or may not 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 Fig. 6). Some simulators rely on mechanisms provided by the host operating system (*files* or *pipes*), while others implement communication on their own using standard procedure calls or regions of memory shared between the trace collector and the trace processor. We shall examine each of the possibilities in turn.

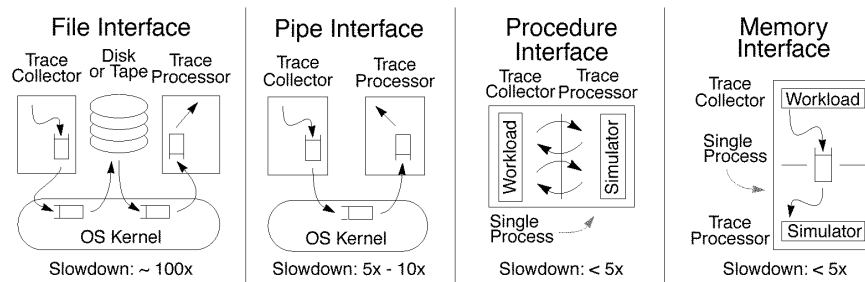


Fig. 6. Trace interfaces. The slowdowns for each of these trace-interface options were estimated by measurements performed on a DECstation 5000/133 with a 33-MHz processor and a SCSI-connected disk running Ultrix

Because they are backed by secondary storage devices, files provide the advantages of deep and non-volatile 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 re-applied during each simulation run and thus provide little or no advantage over simply processing the full address trace.

Both files and pipes are inter-process communication mechanisms provided by an OS filesystem. As such, their use incurs a certain amount of operating system overhead for copying or mapping data from one address space to another, and from context switching between 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 same process address space (see Table 2). 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 multi-process 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 6 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 6 because each corresponds to the simulation of different memory configurations⁴ 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 6 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 45–75 for a relatively simple workload (an optimized version of the Drystone benchmark). 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 60–80. 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) is being simulated. Note that the overheads of the two multi-configuration 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 4.1, perform quite well, providing further evidence that instruction-set emulation is a very viable technique for memory-system evaluation.

⁴ 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.

Table 6. Slowdowns for Some Complete Trace-Driven Memory Simulation Systems. This table gives some typical slowdowns for a complete trace-driven simulation system. The number of configurations considered in a single pass of the trace are given under the *Trace Processing* column. *Slowdowns* are for a single simulation run, while *Effective Slowdowns* are computed by dividing by the number of configurations (given in parenthesis) simulated during that run. In each row, slowdowns were taken (or computed) directly from the referenced paper. For entries that have an asterisk by the reference, slowdowns do not come from the paper, but were determined by our experiments on a DECstation 5000/240.

Sets	Reference	Trace Collection	Trace Reduction	Trace Processing	Interface Method	Slowdown	Effective Slowdown
Pixie + Cache2000	MIPS (1988)*	Annotation	None	Single Config	Pipe	60–80	60–80
Monster + Cheetah	—	Probe-based	Time Sample	Multi (8)	File	419	52
Pixie + Cheetah	Sugumar (1993)*	Annotation	None	Multi (44)	Pipe	183	4
Pixie + Tycho	Gee et al. (1993)	Annotation	None	Multi (44)	Pipe	6250	142
gsim	Magnusson (1993)	Emulation	None	Single Config	Procedure	45–75	45–75
Talisman	Bedichek (1994, 1995)	Emulation	None	Single Config	Procedure	100–150	100–150
TangoLite	Goldschmidt & Hennessy (1992, 1993)	Annotation	None	Single Config	Memory	765	765
Epoxie + Panama	Borg et al. (1989)	Annotation	None	Single Config	Memory	100	100

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 Fig. 7, illustrate that most of the slowdowns are due to trace processing. This observation is supported by reported experiences with other tools as well. Goldschmidt reports 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 (Goldschmidt and Hennessy 1992) (compare the TangoLite entries in Table 2 with those of Table 6). Borg et al. report a similar observation, noting that their Epoxie-driven Panama simulations spend far more time processing address references than collecting them (Borg et al. 1989).

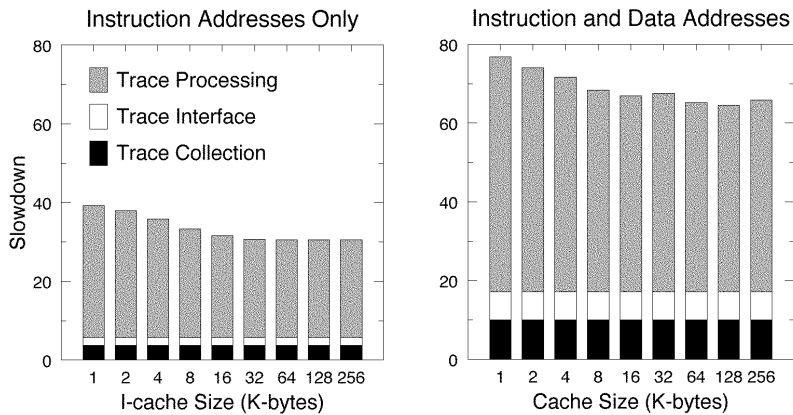


Fig. 7. The components of trace-driven simulation slowdowns. These two plots show the components of trace-driven slowdowns for a complete trace-driven memory simulator constructed by driving the Cache2000 trace processor with Pixie-generated traces via the pipe interface under Ultrix. The left plot shows slowdowns for I-cache simulations, while the right plot shows the slowdowns when simulating both I- and D-caches concurrently

7.3 Summary of Complete Trace-Driven Simulation Systems

As Table 6 and Fig. 7 show, the generation, transfer and processing of trace data for memory-system simulation is 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 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 7). Although the mechanisms that they use differ, each of these tools works by finding special cases where a memory reference has no affect 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 4.2. 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 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

Table 7. Beyond Traces: Some Recent Fast Memory Simulators. Each of the simulators in this table improve performance by reducing or eliminating the cost of processing memory references that do not cause a change of cache state (e.g., cache hits). The cost of cache or TLB hits (*Cycles per Hit*) and misses (*Cycles per Miss*), as well as their relative numbers determine *Overall Slowdown*. Because cache misses are dependent on the configuration (size, associativity) of the cache or TLB being simulated, overall slowdowns can vary widely, so we report them as ranges of values. *Type of Simulation* and *Completeness* summarize the range of simulations supported. Although some systems (e.g., SimOS and WWT) support simulation of multiprocessor memory systems, we report only their uniprocessor slowdowns here.

Method	References	Name	Cycles per Hit	Cycles per Miss	Overall Slowdown	Miss-Detection Mechanism	Type of Simulation	Completeness	
								Multi Process	OS Kernel
Software-Based	Marionosi et al. (1992, 1993)	MemSpy	25	320–510	10–20	Annotation	D-cache	No	No
	Lebeck & Wood (1995)	Fast-Cache	4	55	2–7	Annotation	D-cache	No	No
	Rosenblum et al. (1995); Witchel & Rosenblum (1996)	SimOS + Embra	10	—	7–21	Emulation	D-cache, I-cache, TLB	Yes	Yes
Hardware-Based Miss Detection	Nagle et al. (1993)	Tapeworm	1–2	100–650	0.5–4.5	TLB Miss	TLB	Yes	Yes
	Reinhardt et al. (1993)	WWT	1–2	2,500 ^a	1.4–46 ^a	ECC	D-cache	No	No
	Uhlig et al. (1994)	Tapeworm II	1–2	300	0–10	ECC	I-cache, TLB	Yes	Yes
	Lee (1994)	Tapeworm486	1–2	3,600–4,000	0–14	Page Fault	TLB	Yes	Yes
	Taluri & Hill (1994)	Foxtrot	1–2	1,500–4,000	—	TLB Miss	TLB	No	No

^aMiss costs and slowdowns for WWT are from Lebeck and Wood (1994).

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 4.2, to annotate each workload instruction that makes a memory reference with 9 additional instructions that lookup 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 (7–21) 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 multi-process workloads as well as operating-system kernel code.

8.2 Hardware-Based Miss Detection

Simulators like *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 as memory-management units 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 doesn't add any instructions to the workload that it monitors, non-trapping memory references proceeded 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 has implemented a trap-driven TLB simulator on a 486-based PC running Mach 3.0 (Lee 1994). 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 et al. uses similar techniques in a trap-driven TLB simulator that runs on SPARC-based workstations under the *Foxtrot* operating system to study architectural support for superpages (Talluri and Hill 1994). 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 described above is that they are not easily extended to cache simulation. This is because the mechanisms that they use to cause kernel traps operate at the granularity of a memory page. The first trap-driven simulator that overcame this limitation is the *Wisconsin Wind Tunnel* (WWT), which caused kernel traps by modifying the error-correcting code (ECC) check bits in a SPARC-based CM-5 (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 noted above, 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 et al. reports that WWT exhibits slowdowns of greater than 30 or 40 for caches smaller than 32KB (Lebeck and Wood 1994). 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 multi-process 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 than with 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, multi-process 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.

While 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 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 (Lebeck and Wood 1995). 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 multi-task 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 that Tapeworm II has difficulty compensating for.

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 investigated time sampling by adding an additional check to MemSpy's annotations that enabled and disabled monitoring at regular intervals (Martonosi et al. 1993). 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 two 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 trade-offs between these new memory-system simulators are complex, and neither the software-based or 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 multi-configuration simulations include more overhead than typically reported. Most importantly, 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 addresses 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 fastest known cache simulator may not be of much value if it can only be used to study single-process workloads. A slower, but more complete system, capable of capturing multi-process 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, non-blocking accesses to the memory system.

Acknowledgements

Many thanks to Stuart Sechrest, Peter Bird, Peter Honeyman and Mike Smith, as well as the anonymous reviewers from *Computing Surveys* for their helpful

comments on earlier versions of this paper. A special thanks to Andre Seznec and IRISA for supporting this work during its final stages.

References

- Agarwal, A.: Analysis of cache performance for operating systems and multiprogramming. Ph.D. dissertation, Stanford. 1989
- Agarwal, A., Horowitz, M., and Hennessy, J.: An analytical cache model. *ACM Transactions on Computer Systems* 7 (2): 184–215, 1989
- Agarwal, A. and Huffman, M.: Blocking: Exploiting spatial locality for trace compaction. In *Proc. of the 1990 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Boulder, CO, ACM, 48–57, 1990.
- Baker, M.: Cluster Computing Review. Northeast Parallel Architectures Center (NPAC) Technical Report SCCS-748, November, 1995
- Becker, J. and Park, A.: An analysis of the information content of address and data reference streams. In *Proc. of the 1993 SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems*, Santa Clara, CA, 262–263, 1993
- Bedichek, R.: The Meerkat multicomputer: Tradeoffs in multicomputer architecture. Ph.D. dissertation, University of Washington Department of Computer Science Technical Report 94-06-06, August 1994
- Bedichek, R.: Talisman: fast and accurate multicomputer simulation. In *Proc. of the 1995 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 14–24, 1995
- Borg, A., Kessler, R., Lazana, G., and Wall, D.: Long address traces from RISC machines: generation and analysis. DEC Western Research Lab Technical Report 89/14, 1989
- Borg, A., Kessler, R., and Wall, D.: Generation and analysis of very long address traces. In *Proc. of the 17th Ann. Int. Symp. on Computer Architecture*, IEEE, 1990
- Chen, B.: Software methods for system address tracing. In *Proc. of the Fourth Workshop on Workstation Operating Systems*, Napa, California, 1993
- Chen, B. and Bershad, B.: The impact of operating system structure on memory system performance. In *Proc. of the 14th Symp. on Operating System Principles*, 1993
- Chen, B.: Memory behavior of an X11 window system. In *Proc. of the USENIX Winter 1994 Technical Conf.*, 1994
- Clark, D. W., Bannon, P. J., and Keller, J. B.: Measuring VAX 8800 performance with a histogram hardware monitor. In *Proc. of the 15th Ann. Int. Symp. on Computer Architecture*, Honolulu, Hawaii, IEEE, 176–185, 1985
- Cmelik, R. and Keppel, D.: Shade: A fast instruction-set simulator for execution profiling. University of Washington Technical Report UWCSE 93-06-06. 1993
- Cmelik, B. and Keppel, D.: Shade: A fast instruction-set simulator for execution profiling. In *Proc. of the 1994 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Nashville, TN, ACM, 128–137, 1994
- Cvetanovic, Z. and Bhandarkar, D.: Characterization of Alpha AXP performance using TP and SPEC Workloads. In *Proc. of the 21st Ann. Int. Symp. on Computer Architecture*, Chicago, IL, IEEE, 1994
- Davies, P., Lacroute, P., Heinlein, J., Horowitz, M.: Mable: A technique for efficient machine simulation. Stanford University Technical Report CSL-TR-94-636, October, 1994

- Davis, H., Goldschmidt, S., and Hennessy, J.: Multiprocessor simulation and tracing using Tango. In Proc. of the 1991 Int. Conf. on Parallel Processing, 99–107, 1991
- Digital: Alpha Architecture Handbook. USA, Digital Equipment Corporation, 1992
- Eggers, S., Keppel, D., Koldinger, E., and Levy, H.: Techniques for efficient inline tracing on a shared-memory multiprocessor. In Proc. of the 1990 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, Boulder, CO, 37–47, 1990
- Emer, J. and Clark, D.: A characterization of processor performance in the VAX–11/780. In Proc. of the 11th Ann. Symp. on Computer Architecture, Ann Arbor, MI, IEEE, 301–309, 1984
- Eustace, A. and Srivastava, A.: ATOM: a flexible interface for building high performance program analysis tools. In Proc. of the USENIX Winter 1995 Technical Conf. on UNIX and Advanced Computing Systems, New Orleans, Louisiana, 303–314, January, 1995
- Flanagan, J. K., Nelson, B. E., Archibald, J. K., and Grimsrud, K.: BACH: BYU address collection hardware, the collection of complete traces. In Proc. of the 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation, 128–137, 1992
- Gee, J., Hill, M., Pnevmatikatos, D., and Smith, A. J.: Cache performance of the SPEC92 benchmark suite. IEEE Micro (August): 17–27, 1993
- Goldschmidt, S. and Hennessy, J.: The accuracy of trace-driven simulation of multiprocessors. Stanford University Computer Systems Laboratory Technical Report CSL–TR–92–546, September 1992
- Goldschmidt, S. and Hennessy, J.: The accuracy of trace-driven simulation of multiprocessors. In Proc. of the 1993 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, 146–157, May 1993
- Hammerstrom, D. and Davidson, E.: Information content of CPU memory referencing behavior. In Proc. of the 4th Int. Symp. on Computer Architecture, 184–192, 1977
- Hill, M.: Aspects of cache memory and instruction buffer performance. Ph.D. dissertation, The University of California at Berkeley. 1987
- Hill, M. and Smith, A.: Evaluating associativity in CPU caches. IEEE Transactions on Computers **38** (12): 1612–1630, 1989
- Holliday, M.: Techniques for cache and memory simulation using address reference traces. Int. Journal in Computer Simulation **1**: 129–151, 1991
- IBM: IBM RISC System/6000 Technology. Austin, TX, IBM, 1990
- Jouppi, N.: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proc. of the 17th Ann. Int. Symp. on Computer Architecture, Seattle, WA, IEEE, 364–373, 1990
- Kaeli, D.: Issues in trace-driven simulation. In Proc. of the 22rd Ann. Pittsburgh Modeling and Simulation Conf., Vol. **22**, Part 5, May, 2533–2540, 1991
- Kessler, R.: Analysis of multi-megabyte secondary CPU cache memories. Ph.D. dissertation, University of Wisconsin-Madison. 1991
- Laha, S., Patel, J., and Iyer, R.: Accurate low-cost methods for performance evaluation of cache memory systems. IEEE Transactions on Computers **37** (11): 1325–1336, 1988
- Larus, J. R.: Abstract execution: A technique for efficiently tracing programs. Software Practice and Experience, **20** (12):1241–1258, December, 1990
- Larus, J.: SPIM S20: A MIPS R2000 Simulator. University of Wisconsin-Madison Technical Report, Revision 9. 1991
- Larus, J. R.: Efficient program tracing. IEEE Computer, May: 52–60, 1993

- Larus, J. R. and Schnorr, E.: EEL: Machine independent executable editing. In Proc. SIGPLAN Conf. on Programming Language Design and Implementation, June, 1995
- Lebeck, A. and Wood, D.: Fast-Cache: A new abstraction for memory-system simulation. University of Wisconsin-Madison Technical Report 1211, 1994
- Lebeck, A. and Wood, D.: Active Memory: A new abstraction for memory-system simulation. In Proc. of the 1995 SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems, May, 220–230, 1995
- Lee, C.-C.: A case study of a hardware-managed TLB in a multi-tasking environment. University of Michigan Technical Report. 1994
- Magnusson, P.: A design for efficient simulation of a multiprocessor. In Proc. of the 1993 Western Simulation Multiconference on Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS-93), 69–78, La Jolla, California, 1993
- Martonosi, M., Gupta, A., and Anderson, T.: MemSpy: Analyzing memory system bottlenecks in programs. In Proc. of the 1992 SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems, ACM, 1992
- Martonosi, M., Gupta, A., and Anderson, T.: Effectiveness of trace sampling for performance debugging tools. In Proc. of the 1993 SIGMETRICS Conf. on the Measurement and Modeling of Computer Systems, Santa Clara, California, ACM, 248–259, 1993
- Mattson, R. L., Gecsei, J., Slutz, D. R., and Traiger, I. L.: Evaluation techniques for storage hierarchies. IBM Systems Journal **9** (2): 78–117, 1970
- Maynard, A. M., Donnelly, C., and Olszewski, B.: Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In Proc. of the Sixth Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, ACM, 145–156, 1994
- MIPS: RISCCompiler Languages Programmer's Guide. MIPS, 1988
- Mogul, J. C. and Borg, A.: The effect of context switches on cache performance. In Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, ACM, 75–84, 1991
- Nagle, D., Uhlig, R., and Mudge, T.: Monster: A tool for analyzing the interaction between operating systems and computer architectures. University of Michigan Technical Report CSE-TR-147-92. 1992
- Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T., and Brown, R.: Design tradeoffs for software-managed TLBs. In Proc. of the 20th Ann. Int. Symp. on Computer Architecture, San Diego, California, IEEE, 27–38, 1993
- Nagle, D., Uhlig, R., Mudge, T., and Sechrest, S.: Optimal allocation of on-chip memory for multiple-API operating systems. In Proc. of the 21st Int. Symp. on Computer Architecture, Chicago, IL, 1994
- Pierce, J. and Mudge, T.: IDtrace – A tracing tool for i486 simulation. University of Michigan Technical Report CSE-TR-203-94. 1994
- Pierce, J., Smith, M. D., and Mudge, T.: “Instrumentation tools,” in Fast Simulation of Computer Architectures (T. M. Conte and C. E. Gimarc, eds.), Kluwer Academic Publishers: Boston, MA, 1995
- Pleszkun, A.: Techniques for compressing program address traces. Technical Report, Department of Electrical and Computer Engineering, University of Colorado-Boulder. 1994
- Puzak, T.: Analysis of cache replacement algorithms. Ph.D. dissertation, University of Massachusetts. 1985

- Reinhardt, S., Hill, M., Larus, J., Lebeck, A., Lewis, J., and Wood, D.: The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In Proc. of the 1993 SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems, Santa Clara, CA, ACM, 48–60, 1993
- Reinhardt, S., Pfile, R., and Wood, D.: Decoupled hardware support for distributed shared memory. To appear in Proc. of the 23rd Ann. Int. Symp. on Computer Architecture, 1996
- Romer, T., Lee, D., Voelker, G., Wolman, A., Wong, W., Baer, J., Bershad, B., and Levy, H.: The structure and performance of interpreters. To appear in the Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, October, 1996
- Rosenblum, M., Herrod, S., Witchel, E., and Gupta, A.: Complete computer simulation: the SimOS approach, In IEEE Parallel and Distributed Technology, Fall 1995
- Samples, A.: Mache: no-loss trace compaction. In Proc. of 1989 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, ACM, 89–97, 1989
- Sites, R., Chernoff, A., Kirk, M., Marks, M., and Robinson, S.: Binary translation. Digital Technical Journal **4** (4): 137–152, 1992
- Smith, A. J.: Two methods for the efficient analysis of memory address trace data. IEEE Transactions on Software Engineering SE-**3** (1): 94–101, 1977
- Smith, A. J.: Cache memories. Computing Surveys **14** (3): 473–530, 1982
- Smith, M. D.: Tracing with pixie. Technical Report, Stanford University, Stanford, CA. 1991
- Srivastava, A. and Eustace, A.: ATOM: A system for building customized program analysis tools. In Proc. of the SIGPLAN '94 Conf. on Programming Language Design and Implementation, 196–205, June 1994
- Stephens, C., Cogswell, B., Heinlein, J., Palmer, G., and Shen, J.: Instruction level profiling and evaluation of the IBM RS/6000. In Proc. of the 18th Ann. Int. Symp. on Computer Architecture, Toronto, Canada, ACM, 180–189, 1991
- Stunkel, C. and Fuchs, W.: TRAPEDS: producing traces for multicomputers via execution-driven simulation. In Proc. of the 1989 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, Berkeley, CA, ACM, 70–78, 1989
- Stunkel, C., Janssens, B., and Fuchs, W. K.: Collecting address traces from parallel computers. In Proc. of the 24th Ann. Hawaii Int. Conf. on System Sciences, Hawaii, 373–383, 1991
- Sugumar, R.: Multi-configuration simulation algorithms for the evaluation of computer designs. Ph.D. dissertation, University of Michigan. 1993
- Talluri, M. and Hill, M.: Surpassing the TLB performance of superpages with less operating system support. In Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, ACM, 1994
- Thompson, J. and Smith, A.: Efficient (stack) algorithms for analysis of write-back and sector memories. ACM Transactions on Computer Systems **7** (1): 78–116, 1989
- Uhlig, R., Nagle, D., Mudge, T., and Sechrest, S.: Trap-driven simulation with Tape-worm II. In Proc. of the Sixth Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, California, ACM Press (SIGARCH), 132–144, 1994
- Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., and Emer, J.: Instruction fetching: coping with code bloat. To appear in Proc. of the 22nd Int. Symp. on Computer Architecture, Santa Margherita Ligure, Italy, June, 1995
- Uhlig, R., and Mudge, T.: Trace driven memory simulation: A survey. Computing Surveys **29** (2) 128–170, 1997.

- Upton, M. D.: Architectural trade-offs in a latency tolerant gallium arsenide microprocessor. Ph.D. Dissertation, The University of Michigan, 1994
- Veenstra, J. and Fowler, R.: MINT: A front end for efficient simulation of shared-memory multiprocessors. In Proc. of the 2nd Int. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication systems (MASCOTS), 201–207, 1994
- Wall, D.: Link-time code modification. DEC Western Research Lab Technical Report 89/17. 1989
- Wall, D.: Systems for late code modification. DEC Western Research Lab Technical Report 92/3. 1992
- Wang, W.-H. and Baer, J.-L.: Efficient trace-driven simulation methods for cache performance analysis. In Proc. of the 1990 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, Boulder, CO, ACM, 27–36, 1990
- Witchel, E. and Rosenblum, M.: Embra: fast and flexible machine simulation, In Proc. of the 1996 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, Philadelphia, May, 1996