Software Methods for System Address Tracing

J. Bradley Chen

1 August 1993
CMU-CS-93-188

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890

Appeared in the proceedings of
The Fourth Workshop on Workstation Operating Systems,
October 14-15, 1993, Napa, CA.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, and by an equipment grant from Digital Equipment Corporation,

Keywords: operating systems, address tracing

Abstract

This paper discusses the use of software methods to collect system trace for DEC Ultrix and 3.0 Mach on a DECstation 5000/200. We assert that software methods are a valuable tool for collecting system trace and understanding operating system and memory system behavior for modern workstation workloads. Software methods have some well documented shortcomings. We discuss how their impact was minimized in our system. We further support the validity of the software approach by comparing behavior predicted by our tracing/simulation system to measurements made with less intrusive methods.

1. Introduction

Prior research has established address tracing and trace-based simulation as important tools for understanding memory behavior of modern workstation workloads [1, 2, 3, 5, 7, 8, 11, 13, 16, 17, 18, 20]. Several studies have used hardware methods to collect system trace, and have demonstrated the importance of system activity in understanding overall behavior [1, 10, 20]. With one exception[1]. The current system is a direct descendent of the DEC WRL system. The Titan work was not published. software address tracing has not been used in studies of system behavior. This paper discusses our experience using software methods to trace DEC Ultrix and Mach 3.0 systems on a DECstation 5000/200. We will demonstrate that software methods are an important tool for understanding system behavior.

2. Some Answers

Many issues were raised during the development of the tracing system, both by skeptics of software tracing methods and skeptics in general. Now that the system has been built, the answers to some of their questions are clearer. Here we present a number of potential weaknesses of software methods, and how we have addressed them in our system.

workload	Description	Mach time	Ultrix time
egrep	The pattern search program run three times over a 27K input file.	2.01	1.90
gcc	The GNU C compiler converts a 17K (preprocessed) source file into optimized Sun-3 assembly code.	3.69	4.20
tomcatv	A vectorized mesh generation program, in Fortran.	139.42	155.44

Table 2-1: Example workloads.

Times are in seconds.

We use three example workloads during the following discussion. Table 2-1 describes the workloads. Table 2-2 gives some execution information.

[1]The exception is the software-based tracing system developed at DEC WRL for WRL Titan [5]

workload	instructions				data references			
	Ultrix	%sys	Mach	%sys	Ultrix	%sys	Mach	%sys
egrep	43488134	4.0	45063095	7.4	9398189	7.7	10065385	13.8
gcc	33212131	31.3	38793106	41.2	10192423	28.7	13539261	46.3
tomcatv	2006490556	1.0	2004860546	0.9	970307776	0.7	970804032	0.0

Table 2-2: Example Workloads: Instruction and Data Reference Counts

2.1. Why bother? Just trace with hardware.

Using direct measurement of hardware was relatively straightforward for the VAX 11/780, when measurement devices as fast as the CPU were not hard to obtain. For the VAX, speed and accuracy were significant advantages of hardware tracing methods. Computer architecture has changed in several fundamental ways since the VAX 11/780. Current trends suggest two fundamental limitations of hardware tracing:

- **The Relative Speed of Memory and CPU:** Any measurement system will be fundamentally limited by the speed at which measurements can be recorded. It follows that hardware methods are fundamentally limited by the speed of the memory device they use. Among measurement methodologies, the data requirements of address tracing are extreme. Though it should be possible to build a memory device as fast as the fastest existing processor, the cost and engineering talent required to build such a device increases as computer architectures become complex and as semiconductor processes become exotic. This has direct impact on use of hardware methods to collect address trace. In contrast, software methods scale with the performance of the subject machine. They benefit fully from the subject machine's memory hierarchy, while avoiding the engineering effort required to build it.

  A further advantage of software methods for address tracing is that trace can be made significantly more compact than trace from hardware methods.

- **On-Chip Structures:** Another aspect of modern architectures that interferes with hardware measurement is the movement of more functionality inside of sealed chip packages. This limits the signals that are realistically available for measurement to those appearing on the chip's output pins. A straightforward example of the problems this creates is on-chip caches. Normally an on-chip cache will prevent the majority of memory references from ever appearing on chip output pins. It follows that on-chip caches much be disabled when a hardware monitor is used to collect address trace[2]. Other structures, such as write buffers, instruction prefetch buffers, or translation buffers, may simply be inaccessible.

  Even when counters or accurate timers are available to measure events or cycles for a given sequence of instructions, using the counters can be tricky. In general, system modifications/reconfiguration are required to delimit each code sequence of interest. The problem becomes more complex when contributions from various system components (i-cache, d-cache, write buffer, etc.) need to be isolated in the measurements. All such measurements are straightforward with a software simulator.

[2]Note that disabling caches will also have impact on the time to execute a traced workload. However, as the time to analyze trace tends to be an order of magnitude more than the time required to collect it, by hardware or software methods, the time to collect trace is not an important issue.

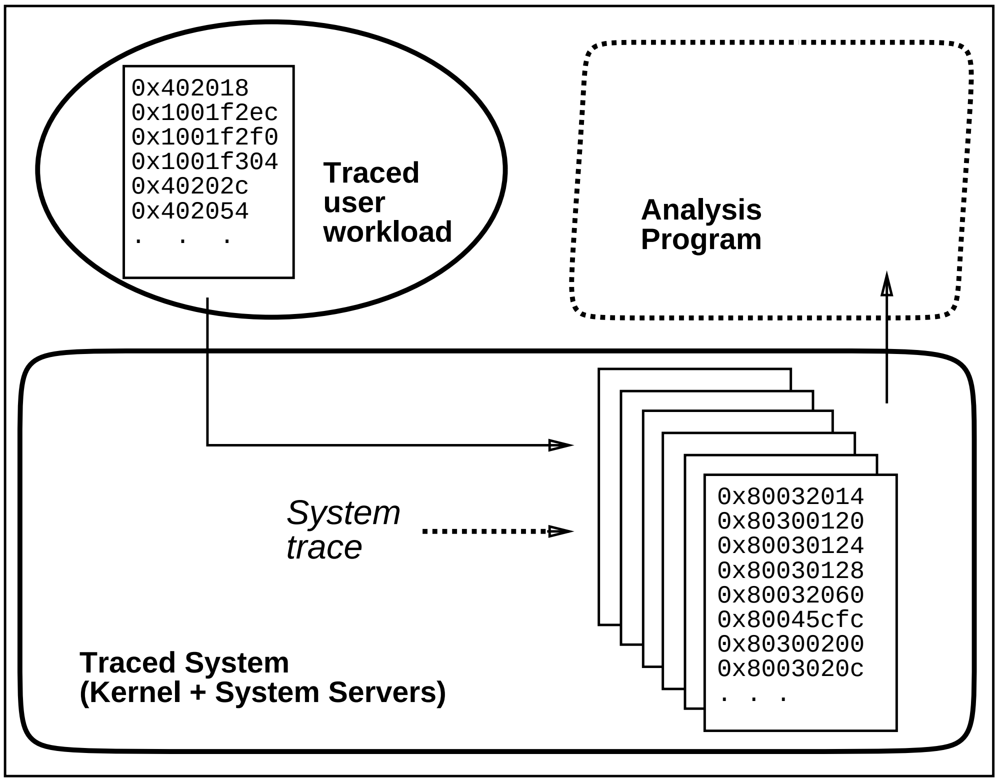

Figure 2-1: Overview of the tracing system.

2.2. Software methods don't work with operating system code.

Software methods have been used to trace DEC Ultrix and 3.0 Mach, running on a DECstation 5000/200. The tracing system is based on a design developed at DEC WRL, and is a direct descendent of earlier WRL tracing systems [5]. The system is based on an object-code rewriting tool called epoxie [21]. Epoxie rewrites programs such that they generate address trace as a side-effect of program execution. See Figure 2-1 for a high-level diagram of the tracing system.

There are three kinds of entities in the tracing system: traced user workloads, the traced kernel, and an analysis program to consume the trace. The kernel is also involved in controlling the tracing system. Appropriate mechanisms are employed to avoid tracing kernel activity that occurs on behalf of the tracing system.

At any instant during tracing the system is in one of two modes: trace-generation or trace-analysis. During trace-generation, references from a given user workload goes into a per-process buffer. When that buffer becomes full, a kernel trap occurs and the per-process trace is transferred to the large in-kernel buffer. When the in-kernel buffer becomes full, the system switches from trace-generation to trace-analysis, during which an analysis program (such as a memory system simulator) digests the trace. Analysis continues until all pending trace has been analyzed and the in-kernel buffer is empty, at which time trace-generation resumes. To preserve the interleaving of trace from kernel and user processes, transfer of trace into the kernel buffer also occurs each time the kernel is activated.

2.3. Locore is too complicated and delicate to trace.

Everything is traced. Epoxie instrumentation is used for all user workloads and for all system code written in C. Some system code written in assembler is instrumented by hand.

Tracing interrupt and exception handling code is tricky but it's not impossible. When epoxie instrumentation cannot be used, code is instrumented by hand. The high-level strategy is to execute as much of the routine as is necessary to put the machine in a 'clean' state, then write enough machine state to the in-kernel trace buffer to regenerate all memory references. Four or five words of trace was sufficient for most of the hand traced routines.

`Wbflush()`, which flushes the write buffer, is one routine that could not be instrumented by epoxie. The code for `wbflush()` looks something like this:

```
LEAF(wbflush)
1:
        bc0f    1b
        nop
        j       ra
        nop
        END(wbflush)
```

This routine has two basic blocks of two instructions each. In the first basic block, the first instruction (`bc0f`) tests if the write buffer is empty. The `nop` that follows fills a branch delay slot. Epoxie instrumentation inserts a call to the basic block trace routine at the beginning of the first basic block. Unfortunately, the basic block trace routine does two writes, so when it returns and the `bc0f` instruction checks the state of the write buffer, it is never empty.

Another situation in which epoxie instrumentation fails is when the machine runs with caches isolated. Cache isolation is used during cache flush routines. In this mode, cache misses fail without going to memory. Although it may have been possible to fix the trace routines to reference memory through the uncached segment when the machine runs with caches isolated, instrumenting them by hand was straightforward and probably an easier solution.

A further case where epoxie instrumentation can't be used is the general exception handler. The overall high-level strategy applies. A further problem occurs because the machine could have been executing either epoxie-instrumented or hand-instrumented code at the time of the exception. This creates in a certain amount of additional state that must be maintained.

For the traced Ultrix kernel there are 866 lines of hand-instrumented assembler. The traced Mach kernel required 711 lines of hand-instrumented assembler.

2.4. An operating system instrumented by software methods will introduce excessive distortion into trace data. The resulting system will be so complicated, you can't tell if the trace is good or not.

There are two principle sources of distortion when tracing the system with software methods: memory dilation and time dilation. Memory dilation occurs because object code expands when it is instrumented. For user programs and system servers, this text growth can result in increased I/O, paging and TLB miss rates [9]. Memory dilation is not an issue for kernel text, as the kernel is loaded at boot time, and runs in unmapped, unpaged memory [12].

The instrumentation tools were modified extensively to minimize text expansion. The text growth factor is observed to range between 1.9 and 2.3. This compares very favorably with other instrumentation tools. Text commonly grows by a factor of five with the original epoxie. When used for address tracing, the pixie tool from MIPS Computer Systems [19] commonly increases text size by a factor of six[3]. Text growth for QPT [4, 14] ranges from four to six [15]. It should be noted that, excepting the modified epoxie, minimal text growth was not a design objective for any of the these tools.

Limiting text growth reduces the impact of memory dilation. The impact of memory dilation is further minimized by collecting trace on a machine with sufficient memory to avoid paging, and by simulating calls to the user TLB miss handler, rather than tracing TLB misses with expanded text.

A second source of distortion is time dilation, which causes an apparent speedup for behavior independent of CPU speed: clock interrupts, scheduler policy and I/O delays. Instrumented code runs slower than uninstrumented code by a factor of approximately fifteen, so any activity whose latency is independent of CPU speed appears to complete about fifteen times faster. For clock interrupts, the clock interrupt rate was slowed by a factor of fifteen. Scheduler policy is more difficult to adapt, as it depends significantly on behavior and slowdown of individual workloads. To avoid this source of distortion, we concentrate on workloads where scheduler activity is dominated by client-server relationships, such that scheduler policy has no impact. Similarly, no special modifications were made to adapt I/O delays. While it would be possible to artificially slow down I/O devices during tracing, I/O delays are spent in the idle loop for our restricted class of workloads, and thus have limited impact on memory system behavior.

Additional trace analysis techniques provide a high degree of confidence in the quality of the trace generated by the instrumented system. Using a kernel with a user TLB miss counter, we compared the TLB miss counts predicted by the simulator to TLB miss counts from an uninstrumented system (See Table 2-3).

	predicted		measured	
workload	Mach	Ultrix	Mach	Ultrix
egrep	6430	176	6122	191
gcc	53389	32617	48355	30574
tomcatv	340968	317839	359976	314950

Table 2-3: TLB misses, measured and predicted.

One source of error in the TLB miss predictions is explicit TLB writes from the kernel. The kernel sometimes avoids a user TLB miss by writing the TLB explicitly, using `tlbdropin()` in Ultrix or `tlb_map_random()` in Mach. In the simulator, which doesn't know about these writes, all TLB fills are caused by TLB misses. Kernel instruction reference counts for *gcc* showed about 1800 calls to `tlbdropin()` for Ultrix, and 3700 calls to `tlb_map_random()` for Mach. Overall, the miss rates predicted by the simulator are reasonable.

[3]For a `gcc` binary with 688128 bytes of text, `pixie -t gcc` grows program text to 4131968 bytes. `Epoxie -t gcc` grows text to 3780608 bytes. QPT expands *gcc* text by a factor of 5.5 [15]. The modified epoxie grows text to 1515520 bytes.

We also used a high resolution timer to measure execution times of the workloads. These times are compared to predicted times from the simulator in Table 2-4.

	predicted		measured	
workload	Mach	Ultrix	Mach	Ultrix
egrep	1.95	1.82	2.01	1.90
gcc	2.66	1.84	3.69	4.20
tomcatv	134.88	151.36	139.42	155.44

Table 2-4: Run Times, measured and predicted, in seconds.

The simulation system measures delay cycles from the memory system, but does not count pipeline stalls from the floating point unit. The simulator predictions for *tomcatv* includes 17.08 seconds of pipeline stalls, as estimated by pixie.

When predicting time from simulator events, I/O is reflected by instructions executed in the idle loop. For long running programs that do little I/O, the predictions from the simulator show good correlation, as in the estimates for *tomcatv*. When the ratio of I/O to computation is high, as for *egrep* and *gcc*, the simulator tends to underestimate the execution time. To understand why, note that for a given I/O operation, the delay will be the same for instrumented and uninstrumented systems, but in the instrumented system the idle loop will execute (approximately) 1/15 as many instructions. As traced instruction and data references are the basis for the simulator's estimate of elapsed time, low estimates result.

Note that for *gcc*, the simulator predicts that the Ultrix run will be faster, but in measured time the contrary is true. This is due to I/O time distortion. For *gcc*, counts of idle loop instructions show about 1.9 million instructions for Mach, as compared to 4 million instructions for Ultrix. This is due in part to differences in buffer-cache implementations. Assuming a factor of fifteen slowdown for instruction execution, this implies that, for an untraced system, the *gcc* run executes about (1.9×15)=28 million idle-loop instructions for Mach, as compared to 60 million idle-loop instructions for Ultrix. Using an idle-loop CPI of 1, we get that the simulator should underestimate Mach I/O time by about 1 second, and Ultrix I/O time by about 2.2 seconds. Adding these corrections to the predicted times we get 3.7 seconds for Mach and 4.1 seconds for Ultrix. With corrections for I/O time distortion, the predicted run times become quite good.

Ultrix		Mach 3.0	
inst. count	routine	inst. count	routine
2332170	idle	1942951	idle_thread_continue
872736	splclock	1663346	aligned_block_copy
727430	spl0	553547	mipscache_Iflush
642095	bcopy	454014	TRAP_exception
501032	kn01_clean_icache	425088	pmap_zero_page
489288	bzero	308968	TRAP_exception_exit
376999	memall	280808	vm_fault
305919	sz_start	225723	ipc_kmsg_copyin
275892	pagein	222159	vm_fault_page
223608	ascintr	212926	pmap_enter

Table 2-5: The Ten Most Active Kernel routines for *gcc*.

As a further means of verifying the tracing system, we used an analysis program that did reference counts for all the instructions in the kernel. This made it straightforward to identify and correct situations where tracing code was erroneously generating trace. Additionally, the data gives a feeling for kernel behavior by identifying active routines. Table 2-5 shows the ten most active routines in Ultrix and Mach during a run of *gcc*.

This data suggests a number of observations. Ultrix spends more time than Mach waiting in the idle loop, again a reflection of buffer-cache implementation. Most of the Ultrix calls to `splclock()` and `spl0()` occur during the idle loop. Mach executes many more instructions in the general exception handler than Ultrix. A closer look at the data shows that 3563 exceptions occur during the Ultrix run and 11399 exceptions occur during the Mach run. Most of the additional exceptions can be attributed to Mach system call emulation, with three exceptions per system call rather than one. Both system do similar amounts of zeroing and copying of pages, although Mach executes far more instructions for block copies.

The instruction reference counts of Table 2-5 are one example of analysis that is straightforward with address trace data. Using a cache simulator, we can do a similar analysis for cache misses. Figure 2-2 shows cache temperature for system (kernel+server) instruction misses during a run of *gcc*. Dark squares indicate "hot spots," cache lines where a disproportionate number of misses occur. In the Ultrix diagram, there is a distinct dark area on the right side, six lines from the bottom. In this region, `memall()`, `swap()`, and `tlbmiss()` all collide in the cache. They were called 660, 253, and 1487 times, respectively. Similarly, several dark regions are visible in the top line of the Mach diagram. `TRAP_exception()`, `ipc_mqueue_send()`, `vm_map_lookup()` and `vm_fault()` all overlap in this part of the cache. These concentrations of cache misses suggest situations where instruction cache behavior could be improved through better text placement.

2.5. You’ll never get accurate results for I/O-bound workloads. You can’t duplicate scheduler behavior. For these reasons, software methods aren’t applicable to complicated multitasking workloads.

These are problems that have not yet been resolved in the current system. They may eventually be addressed, although at present we feel there is much to be learned by thoroughly understanding the limited class of workloads for which the system works well.

For other classes of workloads, several observations are immediate. To the degree that performance of I/O bound workloads is function of I/O device speed, the impact of memory behavior is not significant. For competing compute-bound processes, prior studies [16] have demonstrated that performance degradation will occur.

3. Epilogue

Publications are in preparation for our experiments on memory system behavior. The tracing system has already helped us understand performance anomalies and discover performance bugs in both Ultrix and Mach. Readers interested in a comprehensive report should watch for future publications [6].

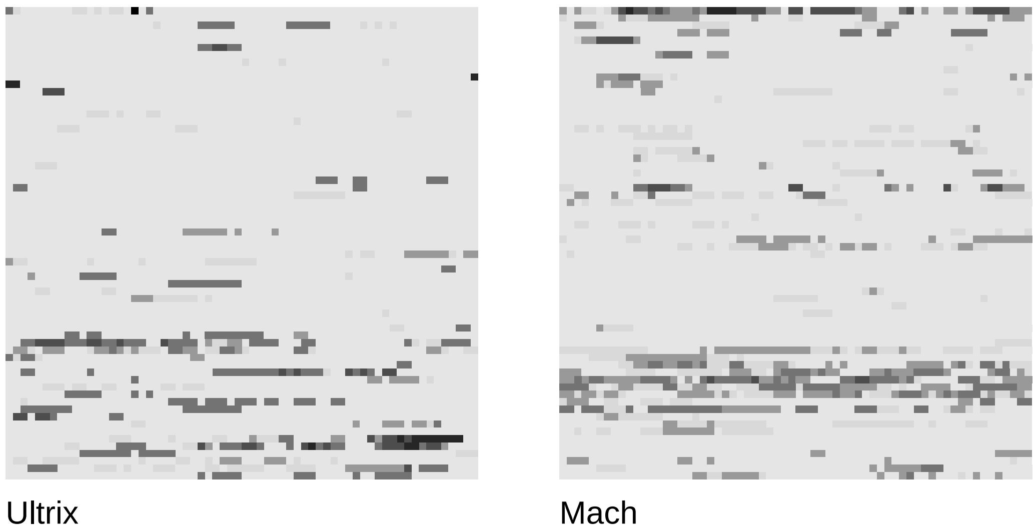

Figure 2-2: Cache Temperatures for system instruction cache misses.

This figure illustrates system misses in a 64 Kbyte instruction cache during a run of the *gcc* benchmark. Each square represents a sixteen byte cache line, with a darker square indicating a relatively large number of misses for that line. The gray level for a given square is determined by the percent of total misses occurring in that line. The darkest squares indicate cases where more than 1% of total misses occurred in that line. There are six gray levels, representing 1%, 0.5%, 0.25%, 0.125% 0.0625% and 0.03125%.

Acknowledgements

Thanks to Doug Tygar, David Steere and Alan Eustace for their useful comments on this paper. Brian Bershad, Alessandro Forin, Daniel Stodolsky, and Mary Thompson helped me with Mach. The tracing system is based on Anita Borg's original system for the DEC WRL Titan, and her contributions continued throughout the project. Thanks to David Wall for providing epoxie, the tool which made this work possible. Thanks also to Digital Equipment Corporation for their generous support.

References

1. Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. 13th Annual Symposium on Computer Architecture, June, 1986, pp. 119-127.

2. Anant Agarwal, John Hennessy, and Mark Horowitz. "Cache Performance of Operating System and Multiprogramming Workloads". *ACM Transactions on Computer Systems 6*, 4 (November 1988), 393-431.

3. Anant Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Ph.D. Th., Stanford University, May 1987. Technical Report No. CSL-TR-87-322.

4. Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. Principles of Programming Languages, January, 1992.

5. Anita Borg, R.E. Kessler, Georgia Lazana, and David Wall. Long Address Traces from RISC Machines: Generation and Analysis. WRL Research Report 89/14, Digital Equipment Western Research Laboratory, 1989.

6. J. Bradley Chen. *Memory System Behavior in Modern Operating Systems*. Ph.D. Th., Carnegie Mellon University, 1994. to appear.

7. Douglas W. Clark. "Cache Performance in the VAX-11/780". *ACM Transactions on Computer Systems 1*, 1 (February 1), 24-37.

8. Douglas W. Clark and Joel S. Emer. "Performance of the VAX 11/780 Translation Buffer: Simulation and Measurement.". *ACM Transactions on Computer Systems 3*, 1 (February 1985).

9. M. DeMoney, J. Moore, and J. Mashey. Operating System Support on a RISC. Proceedings of the 31st Computer Society International Conference (Spring Compcon '86), March, 1986, pp. 138--143.

10. J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud. BACH: BYU Address Collection Hardware; The Collection of Complete Traces. Proceedings of the 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, 1992.

11. Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith. Cache Performance of the SPEC Benchmark Suite. University of Wisconsin-Madison, 1991.

12. Gerry Kane. *Mips RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1987.

13. R.E. Kessler and Mark D. Hill. "Page Placement Algorithms for Large Real-Indexed Caches". *ACM Transactions on Computer Systems 10*, 4 (November 1992).

14. James R. Larus. "Abstract Execution: A Technique for Efficiently Tracing Programs". *Software Practices and Experience) 20*, 12 (December 1990), 1241-1258.

15. James R. Larus. personal communication.

16. Jeffrey C. Mogul and Anita Borg. The Effects of Context Switches on Cache Performance. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.

17. Steven A. Przybylski. *Cache Design: A Performance-Directed Approach.* Morgan-Kaufmann, San Mateo, CA, 1990.

18. Alan Jay Smith. "Cache Memories". *ACM Computer Surveys 14*, 3 (September 1982), 473-530.

19. Micheal D. Smith. Tracing with Pixie. Stanford University, November, 1991.

20. Bart C. Vashaw. *Address Trace Collection and Trace Driven Simulation of Bus Based, Shared Memory Multiprocessors*. Ph.D. Th., Carnegie Mellon University, 1992. Department of Electrical and Computer Engineering.

21. David W. Wall. Systems for Late Code Modification. In *Code Generation --- Concepts, Tools, Techniques*, Springer-Verlag, 1992, pp. 275-293.

Table of Contents

List of Figures

List of Tables