Operating System Impact on Trace-Driven Simulation

Jason Casmira
John Fraser
David Kaeli
Waleed Meleis

Electrical and Computer Engineering
Northeastern University
Boston, MA 02115

Abstract

Trace-driven simulation is commonly used by the computer architecture research community to pursue answers to a wide variety of architectural design issues. Traces taken from benchmark execution have been extensively studied to optimize the design of pipelines, branch predictors, and especially cache memories. Today's computer designs have been optimized based on the characteristics of these benchmarks.

One important aspect that has been ignored in a majority of these trace-driven studies is the effect of the operating system interacting with the benchmark. It has been acknowledged that operating system overhead can introduce a level of interference that can limit the benefits of new designs. The reason why the operating system has been, for the most part, ignored in these studies is the lack of readily available tools that can generate kernel-laden traces.

In this paper we describe two tracing systems that allow the capture of operating system and application traces. We have captured traces of both benchmark and commercial applications using these tools. We show the effects of including operating system activity while studying various architectural design tradeoffs. We have found that the operating system can increase the number of instructions captured in the trace (for a given time sample) by as much as 100X, can significantly modify the instruction mix, and produce different ranges of reference locality. The results of this study show that the inherent characteristics of the applications will dictate the amount of overhead introduced by the operating system.

1 Introduction

Computer architecture designs have been studied extensively using trace-driven modeling [13, 14]. Traces of application execution are captured and fed into software models of a system. Various design tradeoffs can be evaluated using this methodology. The quality of the traces will directly affect the success or failure of any design tradeoff being investigated, so the modeler should carefully inspect the contents of these traces. One issue related to trace content is whether the trace contains operating system execution [16]. To capture operating-system-rich traces, *hardware-based tracing* is commonly employed [11, 15]. Hardware-based tracing involves the construction of a monitor which can capture trace records at machine speeds, while not perturbing the system under study. These tracing systems typically exploit some form of multiplexing across a set of memory arrays in order to match the speed of trace production with the speed of trace capture (which is bound by the speed of the trace storage device). The cost of constructing such a system is typically large, and many times can prohibit using this approach.

Another popular method for capturing traces which modifies either the program source or the object code is called *software-based tracing*. Hooks are placed into the executable and trace information is generated as the program is run. In this paper we describe two such software-based tracing tools which allow the capture of traces which include both application and operating system references. We describe how we use these tools to capture traces that include the operating system, and discuss some of the insights gained through analysis of these traces.

This paper is organized as follows. In Section 2 we provide an overview of previous work on tracing operating system activity. In Section 3 we describe the two tracing tools which we have developed. Section 4 discusses some of the implications of including operating system content in trace-driven evaluations. In Section 5 we conclude, summarizing the contributions of this work.

2 Operating System Tracing

Trace-driven simulation has been the method of choice to evaluate the merit of various pipeline and memory design tradeoffs [16, 29]. A record of execution is captured from the system under test and this recorded trace is replayed through a model of the proposed design. A number of trace capture methodologies have been proposed which capture both application and operating system references. These

tools include both hardware [11, 15, 20] and software [7, 18, 25, 26] based methods. Some of the issues involved with capturing operating system-rich traces include:

1. tracing overhead (introducing slowdown),
2. correctness (handling of real-time interrupts),
3. accuracy (perturbing the memory address space), and
4. completeness (capturing all desired information).

Hardware-based monitoring entails the addition or modification of the testbed hardware so that as a program is executed, a record of all instruction and/or data addresses is created. Trace capture can be accomplished using *sniffing* hardware [11, 15, 20]. Addresses are captured by the hardware, stored temporarily in a buffer, and off-loaded periodically. This additional hardware can be very expensive to build. Hardware-based methods are also usually tied to a specific system implementation and not flexible if changes are needed.

Certain environments necessitate the use of hardware monitoring. For instance, in [15, 17, 28], shared-bus multiprocessor environments were studied. Hardware snooping on the memory bus was used on these systems to capture application and operating system address traces. This type of tracing technique requires extensive modifications to be made to the system being traced, involving both hardware changes and operating system modifications.

Software-based tracing achieves many of the same goals as hardware, but instead of altering the system hardware, software is modified. Code can be modified at assembly time [24], link time [23] or execute time [19, 28]. This modified code captures the trace as the program is executed. The strength of software-based tracing is its modifiability. If the desired trace information changes, the trace mechanism is easily updated. In either case, the trace produced is fair representation of the workload and is not generally limited in length. Most software-based tracing methodologies (except Epoxie and ATOM) are not able to capture operating system traces.

Another tracing technique is to use *emulation*. Here, a layer is constructed between the host machine and the operating system under evaluation. A good example of such a system is the SimOS project at Stanford [25, 26]. The layer only emulates enough of the system to allow the operating system to run correctly. While this system provides a flexible interface for collecting operating-system rich traces, the accuracy of the captured trace is suspect. Since emulation is performed, execution will be perturbed. This can significantly reduce the accuracy of the trace-driven simulation.

A fourth approach to tracing utilizes microcode modification to capture trace information, introducing minimal slowdown. This idea is not new [1, 2, 3]; an early study on the VAX used this methodology. Very few microprocessors still utilize microcode. The DEC Alpha uses a form of microcode called PALcode [31]. PALcode was developed to provide VAX compatibility and efficient system utilities, while also providing a vehicle for efficient trace capture. Some examples of PALcode functions include:

- context swapping,
- privileged instructions,
- VAX-style interlocked memory access,
- power-up initialization, and
- emulation of instructions with no explicit hardware support.

The PALcode environment enables the code to have complete control of the machine state, disable interrupts, and access low-level system hardware. These capabilities are needed to be able to efficiently capture traces.

In the work discussed here, we use the ATOM and PatchWrx toolsets to capture application and operating system traces, as run on a DEC Alpha platform. Both of these tools are extensions to earlier toolsets.

The ATOM toolset was originally developed by Alan Eustace and Amitahb Srinivastva, at DEC's Western Research Laboratory. ATOM allows the user to build a variety of customized execution-driven program analysis tools [30]. The PatchWrx toolset was originally developed by Richard Sites, also at DEC WRL [21]. We have extended this work and currently have trace capability on NT 4.0. We will describe these tools and how to use them to capture application and operating system traces.

3 ATOM

ATOM is a link-time code modification tools which inserts procedure calls to user defined analysis routines at link time. ATOM provides the ability to instrument a program at the procedure level, basic block level, or the instruction level. ATOM is accompanied with a rich set of analysis routines [30]. These routines are called during program execution time, and various simulations can be performed. This type of trace-driven simulation is called *execution-based simulation*. Execution-based simulation reduces much of the overhead associated with capturing traces, since traces need not be stored to disk or tape. They are instead passed directly to the simulation model.

ATOM has been used extensively by the computer architecture research community to capture application traces as input to execution-based analysis. ATOM introduces limited overhead while still providing a great degree of flexibility. Execution-driven simulations for single processes (i.e., of user application execution) are quite easy to generate. Up until recently, ATOM could not capture operating system or multiple processes easily. The latest release of ATOM, however, does provide this capability [7]. We built upon the work of Eustace and Chen as described in [9], using ATOM to study the effects of the operating system on cache performance.

3.1 Using ATOM for Kernel Tracing

ATOM provides a method to instrument programs at link time with user defined functions, generating runtime program analysis. Srivatava and Eustace describe the base ATOM tool in [23].

The latest version of ATOM allows the OSF/1 kernel to be instrumented as well, providing a means to monitor the execution of the operating system. There are certain limitations associated with working with the kernel, however. There is no simple way to insert a procedure call at program termination. Certain kernel procedures cannot be instrumented, but they are a minor portion of the kernel (**locore**, **lockprim**, and **spl**, which comprise 132 out of 10,678 procedures). Floating point numbers cannot be used within kernel analysis programs, nor can one use any system calls (e.g., I/O, memory allocation). The lack of file I/O makes it difficult for run time declaration of simulation parameters (they must be statically defined). Finally, the model for dynamic allocation of memory is not accurate within the kernel, so any analysis involving memory allocation is suspect.

We use the ATOM toolset, version 2.20 and the WRL Enhancement Kit, which is included in the Platinum release of OSF/1. We built our traceable kernel on a DEC Alpha AXP 3000 model 300 running DEC OSF/1 version 3.2A. The root partition was expanded on the system to accommodate the larger instrumented kernel, in addition to the normal root resident programs. The critical factor in our ability to use this instrumented kernel was memory. The memory present on the machine was enough to allow smaller benchmarks to be run, but larger programs crashed the system. This was a dedicated system, so no other users or processes were running during tracing.

The last major problem encountered is the fact that tracing is being performed in a multi-threaded environment. If the analysis code written for ATOM fails to take this into account, problems can occur. For example, as a kernel process executes, it will invoke the ATOM analysis program. However, if the ATOM analysis code encounters an interrupt while it is running, a kernel routine will interrupt the analysis program. This kernel routine will also call the ATOM analysis code, leaving the previous invocation of the analysis code incomplete. The state of any variables in the analysis code end up in some indeterminate state (this problem is typically referred to as *reentrant code*. To counter these problems, we set a flag upon entry to the ATOM analysis program. This flag is a global variable, and is reset upon exit from the analysis program. If the analysis program ever encounters the flag already set upon entry, the analysis program will immediately exit. This ensures that once we are in the analysis routine that interrupts will not be traced until we have completed the current invocation of the analysis routine. While there are still holes in this strategy (*livelock* can occur), we have used this scheme effectively in our tracing.

Using ATOM, we instrument the benchmark programs just as we would if we were only interested in generating application traces. The slowdown when capturing traces is on the order of 10X. Adjustments to timers and system services have been made in order to account for this slowdown. The only difference is how we access the data structure in the kernel. Due to the limited space in this paper, we point the reader to [12] for a complete description of operating system tracing using ATOM.

4 PatchWrx

The PatchWrx Tracing toolset was developed to capture activity of Microsoft NT workloads and kernel execution, as run on a DEC Alpha platform. The goal was to capture a rich set of traces without introducing a large overhead to the tracing environment. Currently the system is able to capture traces on NT 4.0, as run on a DEC 21064 platform (tracing is processor-version dependent).

The tracing philosophy is to modify an NT executable or DLL (dynamic link library), *patching* all branches. The patches would then generate a trace record which could later be used to reconstruct the execution flow. A similar approach was used in the Pixie toolset for the MIPS platform [27].

To patch an executable or DLL, branches are overwritten with an unconditional jump to a patch section of code (appended to the end of the image). The patch section issues the appropriate PALcall (based on the type of branch that was overwritten). The PALcode captures the trace information and stores it in a reserved area of storage. The PALcode then returns to the patch area, where a copy of the overwritten branch is located. This branch is executed, and program execution resumes. Load and store instructions can also be patched to obtain full data reference traces as well.

This provides a very efficient methodology for quickly capturing selected points in programs, DLLs and the kernel, which after execution, can be used to reconstruct the entire instruction stream.

Instrumentation of the code can be done on a single executable or DLL, or the entire system can be patched. The system runs normally while patched, only incurring a 2X slowdown with tracing disabled, and a 4X slowdown with tracing enabled (there is the option to enable/disable tracing on the patched system).

Tracing involves three step:

1. patching the code,
2. tracing the code, generating trace records, and
3. reconstructing the executed stream.

We have been able to patch and trace a variety of applications, including simple benchmarks taken from the Bytemark benchmark suite, the MS CD Player, and Oracle 7.3. While the tool is currently in beta testing, we have validated the accuracy and representativeness of the captured programs. Next we report on the characteristics of the traces captured using ATOM and PatchWrx.

5 Characteristics of Operating System Rich Traces

We will present results from the two different tracing platforms, comparing results across a single platform. Our goal is to demonstrate two things:

1. show the influence of the operating system in application traces, and
2. show the relationship between different classes of applications (i.e., benchmark versus real application) and their interaction with the operating system.

We begin by presenting instruction and data reference breakdowns for the ATOM/OSF environment. We include analysis of a very simple program (the Hello World program), and compare it to programs taken from the SPEC benchmark suite. The purpose of including such a simple program is to illustrate just how the operating system can dominate execution of even the simplest of programs. Tables 1 and 2 show the amount of operating system interaction for HelloWorld and four SPEC applications. Table 1 lists the number of instruction reads, data reads and data writes, without any operating system interaction. Table 2 lists the same data for just the operating system overhead when running the same five programs.

HelloWorld spends most of its time in the operating system, while the SPEC benchmarks require minimal interaction. Espresso and Alvinn interact the least, since they do not extensively use system services. Compress uses the file system, requiring more interaction with the operating system. GCC requires the most amount of interaction, since it also presents the most complex application behavior.

In Table 3 simulation results of modeling 8KB direct mapped instruction and data caches are shown for the four SPEC benchmarks. A 32 byte line size is assumed. The second column lists the instruction cache miss rate when considering the operating system and the application run together, versus the third column which lists the application-only miss rate. Similar information is provided in columns four and five for data cache miss rates. While we see that the instruction cache miss rates are only mildly affected (only GCC experiences a significant increase), we do see a dramatic change in the data cache miss rates. This effect is fairly consistent, independent of application. A full set of cache simulations, including results for capturing multiple programs and the operating system, can be found in [12]

Next we present results for Microsoft NT 4.0 workloads, as captured by the PatchWrx tool. Note that the length of the traces that we have captured to date is only bound by the limited storage provided on our tracing system. PatchWrx allows for nearly unlimited length traces, given sufficient memory.

For the NT environment we will discuss the following applications:

- 5 programs from the Bytemark benchmark suite,
- the Microsoft CD Player, and

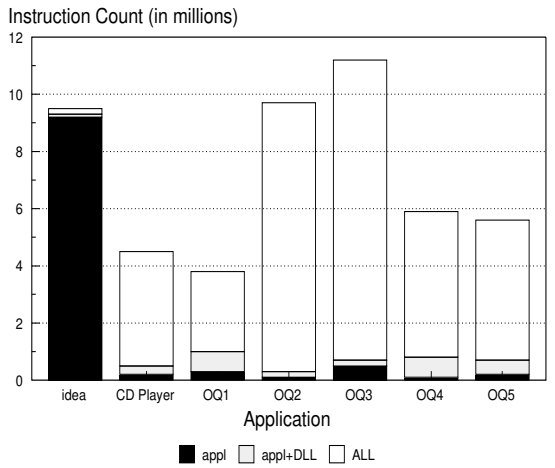

Figure 1: Instruction Counts

- and Oracle 7.3 running five queries from the TPCD benchmark.

Figure 5 shows a breakdown of the amount instructions captured in: 1) the application only, 2) the application and associated dynamic link libraries (DLL's), and 3) the application, associated DLL's, and the operating system. As we can see, the amount of operating system interaction is heavily application dependent. The Bytemark benchmarks studied contain very little operating system interaction, and thus do not provide a realistic view of NT application behavior (with respect to operating system interaction). The CD Player and the five Oracle traces (OQ1-OQ5 are 5 transactions associated with the TPCD benchmark) all exhibit how operating system activity can dominate overall execution. To further support this point, we found that capturing application-only traces of Oracle took on the order of minutes to fill the 45 MB trace buffer, while capturing operating system-laden traces took on the order of only a few second to fill the buffer.

Figure 5 shows the average basic block size obtained when considering the application alone, versus including DLL and OS activity. For the benchmark programs, the basic block size does not change significantly (mainly due to the small amount of DLL and OS activity captured in the trace). As we can see, the basic block sizes of the DLL's generally increase the basic block size, but when we include the operating system, the average basic block size is reduced. While we do not have sufficient information to generalize this result, the fact is that DLL and OS interaction can have a dramatic effect on basic block lengths.

In Figure 5 we present instruction mixes. The five categories are based on the a breakdown of the Alpha AXP instruction set architecture [31]. The five categories:

1. OTHER - alu and floating point operations
2. LD/ST - loads and stores

Application Name	Instruction Reads	Data Reads	Data Writes
HelloWorld	1,247	207	135
Compress	87,045,969	22,412,010	8,521,661
GCC	160,240,175	50,197,333	19,074,845
Espresso	977,787,899	225,779,331	59,867,421
Alvinn	5,233,222,045	1,415,013,630	487,428,474

Table 1: The number of references captured (benchmark only)

Application Name	Instruction Reads	Data Reads	Data Writes
HelloWorld	337,491	84,403	51,332
Compress	5,567,602	1,518,924	802,242
GCC	18,705,569	5,130,601	2,613,506
Espresso	29,093,428	9,107,479	3,585,537
Alvinn	197,365,478	60,413,211	25,986,851

Table 2: The number of reference captured (operating system only)

Application Name	I-Miss Rate Appl + OS	I-Miss Rate Appl Only	D-Miss Rate Appl + OS	D-Miss Rate Appl Only
Compress	.38	.02	16.1	13.2
GCC	4.2	3.2	8.0	6.2
Espresso	1.2	0.9	7.5	6.2
Alvinn	.24	.15	6.7	5.6

Table 3: Cache miss rates, with and without operating system overhead

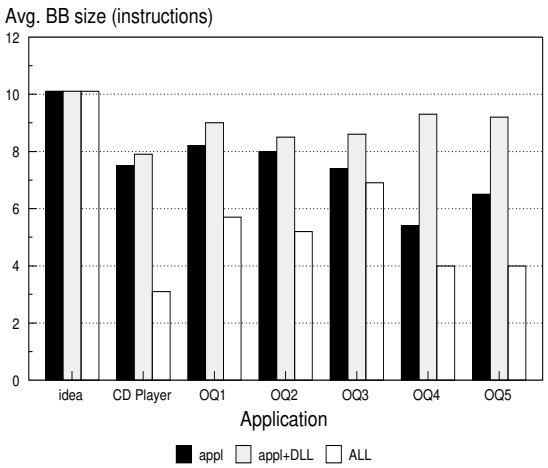

Figure 2: Basic Block Length

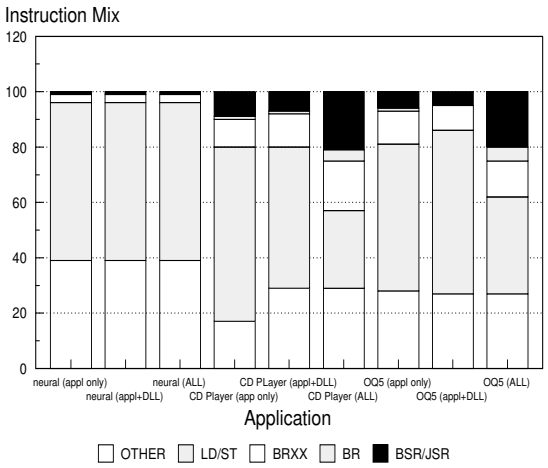

Figure 3: Instruction Mix

3. BRXX - conditional branches

4. BR - unconditional branches

5. BSR/JSR - branch and jump to subroutine

Again we compare the application alone, application and DLL's, and the entire system including the NT kernel. As we can see, the mix for the benchmarks is not significantly affected (again, due to the limited execution in the kernel). For the more system-oriented applications, we can see a dramatic change in the instruction mix. Since instruction mix has a dramatic impact on architectural choices (e.g., branch prediction, cache design, pipeline design), the operating must be included to provide accurate input to the design tradeoff process.

Our main conclusion is that we need to include operating system interaction for these applications if we want an accurate picture of program execution. By using application-only execution traces, many misleading conclusions will be drawn from simulation results.

6 Summary

Trace-driven simulation is the chosen methodology for performing quantitative analysis of architectural design tradeoffs. In the past, many analyses have chosen to ignore the interaction with the operating system. This can significantly reduce the accuracy of any trace-driven simulation. Architects and modelers must carefully inspect the input to their simulation models in order to insure that their conclusions will truly lead to a faster or more cost-effective design.

We plan to continue our work on PatchWrx, as we anticipate the growth in popularity of the Microsoft NT platform. Currently we are looking to capture multiprocessing workloads, as well as moving to the 21164 and readying for NT 5.0. We are also interested in providing a similar capability on the Intel X86 architecture.

7 Acknowledgments

This research was supported by a fellowship from the Charles Stark Draper Laboratory, by a grant from Microsoft Research and by NSF CAREER Grant No. MIP-9501172. We would like to acknowledge David Hunter, Richard Sites, and Sharon Smith for the opportunity to participate in the development of the PatchWrx tool. The authors also wish to thank Brad Chen, Alan Eustace, and Greg Lueck for their considerable assistance working with ATOM.

References

[1] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Kluwer, 1989.

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming Workloads," *ACM Transactions on Computer Systems*, Vol. 6 No. 4, Nov 1988, pp. 393-431.

[3] A. Agarwal, M. Horowitz, and J. Hennessy, "An Analytical Cache Model," *ACM Transactions on Computer Systems*, Vol. 7, No. 2, May 1989, pp. 184-215.

[4] P. Argade, D. Charles, and C. Taylor, "A Technique for Monitoring Run Time Dynamics of an Operating System and a Microprocessor Executing User Applications," *ACM SIGPLAN Notices*, Vol. 29, No.11, Nov 1994, pp. 122-131.

[5] A. Borg, R. Kessler, and D. Wall, "Generation and Analysis of Very Long Address Traces," *Proc. of the 18th International Symposium on Computer Architecture*, Vol. 18, No. 2, Jun. 1990, pp. 270-279.

[6] B. Chen, *The Impact of Software Structure and Policy on CPU and Memory System Performance*, PhD Thesis, Carnegie Mellon, CMU-CS-94-145, 1994.

[7] B. Chen and B. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Operating Systems Review,* Vol. 27, No. 5, Dec. 1993, pp. 120-133.

[8] B. Chen, D. Wall, and A. Borg, "Software Methods for System Address Tracing: Implementation and Validation," DEC WRL Research Report, 94/6, 1994.

[9] A. Eustace and B. Chen, *ATOM Kernel Instrumentation Guide Version 0.4,* DRAFT (unpublished), Sept. 1995.

[10] J. Feldman, J. and C. Retter, *Computer Architecture: A Designers Text Based on a Generic RISC,* McGraw Hill, 1994.

[11] K. Flanagan, J. Archibald and K. Grimsrud, "BACH: BYU Address Collection Hardware, the Collection of Complete Traces," *6th International Conference on Modeling Techniques and Tools for Computer Evaluation*, 1992, pp. 128-137.

[12] J. Fraser, *Cache Modeling of Operating System Workloads*, MS Thesis, Northeastern University, 1996.

[13] J. Gee, M. Hill, M. and D. Pnevmatikatos, "Cache Performance of the SPEC92 Benchmark Suite," IEEE Micro, Vol. 13, No. 4, Aug. 1993, pp. 17-27.

[14] N. Jouppi, "Cache Write Policies and Performance", Computer Architecture News, Vol. 21, No. 2, June 1993, pp. 191-201.

[15] D. Kaeli, O. LaMaire, W. White, P. Hennet and W. Starke, "Real-Time Trace Generation," *International Journal on Computer Simulation*, Vol. 6, No. 1, 1996, pp. 53-68.

[16] D. Kaeli, "Issues in Trace-Driven Simulation," *Lecture Notes in Computer Science, No. 729, Performance Evaluation of Computer and Communication Systems*, L. Donatiello and R. Nelson, eds., Springer-Verlag, 1993, pp. 224-244.

[17] D. Kaeli, L. Fong, D. Renfrew, K. Imming, and R. Booth, "Performance Analysis on a CC-NUMA Prototype," *IBM Journal of Research and Development, Special Issue on Performance Tools,* 41, No. 3, May 1997, pp. 205-214.

[18] J. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs", *Univ. of Wisconsin-Madison Technical Report*, 1990.

[19] J. Larus, "Efficient Program Tracing," IEEE Computer Magazine, May 1993, pp. 52-60, 1993.

[20] D. Nagle, R. Uhlig and T. Mudge, "Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures," Univ. of Michigan Technical Report CSE-TR-147-92, 1992.

[21] S. Perl and R. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," *2nd USENIX Symposium on Operating System Design and Implementation,*, pp. 169-183, 1996.

[22] D. Pnevmatikatos and M. Hill, "Cache Performance of the Integer SPEC Benchmarks on a RISC," *Computer Architecture News,* Vol. 18, No. 2, June 1990, pp. 53-68.

[23] A. Srivastava and A, Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *ACM SIGPLAN Notices*, Vol. 29, No. 6, June 1994, pp. 196-205.

[24] C. Stunkel and K. Fuchs, "TRAPEDS: Producing Traces for MultiComputers via Execution-Driven Simulation," *Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Berkeley, CA, pp. 70-78, 1989.

[25] M. Rosenblum, S.A. Herrod, E. Witchel and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *to appear in the IEEE Journal of Parallel and Distributed Technology*, 1998.

[26] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Simulation*, Vol. 7, No. 1, January 1997, pp. 78-103.

[27] M. Smith, "Tracing with Pixie," *Technical Report*, Stanford University, Stanford, CA, 1991.

[28] J. Torrellas, A. Gupta and J. Hennessy, "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System," *ACM SIGPLAN Notices*, Vol. 27, No. 9, Sept. 1992, pp. 162-174.

[29] R. Uhlig and T. Mudge, "Trace-Driven Memory Simulation: A Survey," ACM Computing Surveys, to appear.

[30] *Program Analysis Using Atom Tools*, Digital Equipment Corporation, March 1996.

[31] *Alpha AXP Architecture Handbook*, Digital Equipment Corporation, No. EC-QD2KA-TE, October 1994.