Operating-System Level Tracing Tools for the DEC AXP Architecture

Jason P. Casmira
John Fraser
David R. Kaeli

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA
casmira,kaeli@ece.neu.edu

Abstract

Trace-driven simulation is commonly used by the computer architecture community to answer a wide range of design questions. Traces taken from benchmark program execution (commonly from the SPEC95 suite) have been used extensively to study instruction scheduling, branch prediction, and cache design. Today's computer designs have been optimized based on the workload characteristics of these benchmarks.

One important issue which has been ignored in these traces is the lack of operating system activity. It has been acknowledged by a number of researchers that operating system interaction can severely affect the validity of any trace-driven simulation study. The major reason why most studies have elected to ignore this fact is due to the difficulty of obtaining such traces.

In this contribution we describe two tools which have been developed at Digital Equipment Corporation, in collaboration with Northeastern University's Computer Architecture Research Laboratory, which capture operating-system rich traces. These tools can be used for capturing trace information on an DEC Alpha-based system, running either the DEC Unix or Microsoft Windows NT operating system.

1 Introduction

Trace-driven simulation has been extensively used to evaluate the merit of a wide range of system design features. A trace provides the modeler with a repeatable and representative sample of a program's execution. An issue which has frequently been ignored is the lack of operating system execution in the traces used. Excluding the overhead associated with operating system execution can have a dramatic effect on performance [1, 3, 2]. Only recently has this issue received attention [6, 9], though most of these studies have used either hardware-based tracers or simulated the effects of task switch using synthetic means to capture the effects of the operating system in an application trace.

For example, including the effects of the operating system in a cache simulation can cause the designer to rethink how to capture the temporal and spatial locality present in a system workload. By periodically interrupting a program in order to allow other processes to run, the lifetime of a program footprint in cache memory is severely reduced. Some modelers have attempted to estimate this effect by flushing the cache after a set number of trace records have been recorded. It has been reported that his is poor estimate of true program behavior and more accurate measures are needed [1].

2 Operating System Level Tracing With ATOM

In work previous to ours, Chen et al. captured traces of applications and operating system using a modified version of the ATOM tool [4]. We have extended this work, providing the capability to capture multiple programs being executed concurrently, along with operating system interaction, all in a single trace. A more detailed description of this work can be found in [7].

ATOM was developed at DEC Western Research Laboratories [10], and provides an efficient mechanism to instrument programs at link time. Instrumentation is performed by inserting procedure calls to analysis routines at user defined instrumentation points. Various architectural features can be modeled in these analysis routines, and the instrumentation granularity can be varied.

The current release of ATOM allows DEC Unix to be instrumented, and provides a means to monitor the execution of the operating system [5]. There are certain limitations on tracing the kernel using this tool. Certain procedures can not be instrumented, but these comprise a minor portion of the kernel. Dynamic memory allocation within the kernel can perturb the accurate modeling of different memory allocation strategies in an analysis program. Besides these minor inconveniences, we have been successful in instrumenting and tracing programs on top of DEC Unix version 3.2A.

To be able to accommodate our instrumented kernel, we had to increase our root partition to 85 MB, increase swap space to 323 MB, and expand the /usr partition to 694 MB. Since we wanted to capture traces containing both the kernel and user programs, and input these to a common architectural model, a method was established to map a portion of memory to be shared by multiple programs. Also, we had to deal with the issue of reentrant code, since the operating system may need to be entered from the analysis routine.

Based on the size and complexity of our instrumented kernel, we had to run DEC Unix in single user mode. While this will limit the representativeness of our traces as captured in this environment, we are more focused on the application-dependent interaction with the operating system. Because we are slowing down our kernel by a factor of 10-100x, we also had to scale real-time timers and interrupts.

To illustrate the utility of this toolset, we show some program characteristics of a set of benchmark programs. In Table 1 we show the number of references generated for 5 benchmark programs (4 SPEC benchmarks and the hello.c program). Frequencies for the application alone, as well as the operating system along, are presented. For a simple program like hello.c (which also uses the operating system), the operating system can dominate overall execution. The amount of operating system overhead varies among programs, and depends on the underlying OS requirements of the application.

Figures 1 and 2 plot the percentage of references from the operating system and the in-

Benchmark	Instruction Fetches	Data Reads	Data Writes	Total Data	Total References
Hello World	1,247	207	135	342	1,589
OS	337491	84,403	51,332	135,735	473,226
Total	338,738	84,610	51,467	136,077	474,815
Compress	87,045,969	22,412,010	8,521,661	30,933,671	117,979,640
OS	5,567,602	1,518,924	802,242	2,321,166	7,888,768
Total	92,613,571	23,930,934	9,323,903	33,254,837	125,868,408
GCC	160,240,175	50,197,333	19,074,845	69,272,178	229,512,353
OS	18,705,569	5,130,601	2,613,506	7,744,107	26,449,676
Total	178,945,744	55,327,934	21,688,351	77,016,285	255,962,029
Espresso	977,787,899	225,779,331	59,867,421	285,646,752	1,263,434,651
OS	29,093,428	9,107,479	3,585,537	12,693,016	41,786,444
Total	1,006,881,327	234,886,810	63,452,958	298,339,768	1,305,22 1,095
Alvinn	5,233,222,045	1,415,013,630	487,428,474	1,902,442,104	7,135,66 4,149
OS	197,365,478	60,413,211	25,986,851	86,400,062	283,765,540
Total	5,430,587,523	1,475,426,841	513,415,325	1,988,842,166	7,4 19,429,689

Table 1: Benchmarks with Operating System References

crease in the number of references caused by including the operating system, respectively. We can see that for even simple benchmark applications, that considering the operating system can comprise a significant portion of the overall trace (e.g., more than 10% of the references for GCC). Figure 3 shows the relative distribution of each reference type within the workload for both the program and its operating system overhead. Both the program and operating system references have about the same distribution, with roughly 70% instruction fetches. This result is consistent with those presented in [1].

While work is still in progress to improve this toolset, we currently have the ability to generate and distribute traces to research groups.

2.1 PatchWrx

In present work we are developing a new version of the PatchWrx toolset. This work is performed in conjunction with DEC's Advanced Software Development Laboratory, and builds upon the framework provided by Perl and Sites[11]. The concept behind PatchWrx is to develop a tool to study performance of a commercial operating system, running commercial applications. The PatchWrx tool is currently implemented on Alpha Windows NT version 3.50, and we are in the process of completing the necessary modifications for tracing NT version 4.0.

PatchWrx allows for the capture of all instructions and memory references, while running a real operating system and real applications (e.g., NT executing MS Word, Lotus 123, etc.). The PatchWrx tool provides the capability to capture low-level trace information, without severely perturbing the system. While tracing is performed, the system runs at 1/8 to 1/2 of its normal speed. This is a difficult feat to accomplish while capturing content rich instruction-level traces. Most trace-generation systems introduce a 10-100x slowdown during tracing.

PatchWrx follows similar concepts as described in the ATUM work by Agarwal [1]. ATUM provided tracing via instrumenting using the microcode of the DEC VAX processor. PatchWrx accomplishes tracing by inserting PALcall instructions at various points in the executable. PALcalls are architected Alpha instructions, which trap to one of the Privledged Architecture Library (PAL) subroutines, and do so without disturbing any programmer-visible state.

In the PAL subroutine, trace information is captured and execution is returned to the program upon completion of recording the trace record. These PAL calls incur about a 40-50 instruction overhead, but their effect is small since we only need to instrument every 5-7 instructions on average (to collect instruction traces).

In its present version, PatchWrx can capture traces on Window NT 3.0. Both instruction and memory accesses can be captured. Sites et al. have reported on using this tool to capture traces of Microsoft's SQL server database engine running the TPC-B benchmark on an Alpha Windows NT 3.50 platform [11]. Multiprocessor traces can also be captured on this platform.

We are currently nearing the completion of tracing capability on the Windows NT 4.0 platform. Once this toolset is fully functional, the system will be ported to an MP platform. Traces from this work will be made available to the research community in the near future.

3 Conclusions

We have described two efforts underway at Northeastern University's Computer Architecture Research Laboratory, which are focused on studying the interaction of applications and operating systems. These project are in collaboration with researchers at DEC's research laboratories in Palo Alto. The outcome of this collaboration will be to provide a set of for the DEC Alpha architecture which can capture operating-system rich traces. The hope is that with the availability of improved tracing tools and traces which the architecture community can teach students to consider the effects of the operating system interaction when evaluating architectural features in a trace-driven study. For more information, see the Northeastern University Computer Architecture Research Laboratory URL: http://www.ece.neu.edu/info/architecture/nucar.html.

References

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

[2] J. Chen and B. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Proc. of the 14th ACM Symposium on Operating System Principles*, December 1993.

[3] 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.

[4] J. Chen and A. Eustace, "Kernel Instrumentation Tools and Techniques," Technical Report 26-95, Center for Research in Computing Technology, Harvard University, Cambridge, MA, Nov. 1995.

[5] A. Eustace and B. Chen, "ATOM Kernel Instrumentation Guide Version 0.4 ", unpublished, Sep 1995.

[6] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest and J. Emer, "Instruction Fetching: Coping with Code Bloat," *Proc. of the 28th International Symposium on Computer Architecture*, June 1995.

[7] J. Fraser, *Cache Analysis in a Multiprocess Environment Using Execution Driven Simulation*, MS Thesis, Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, 1996.

[8] N. Gloy, C. Young, J.B. Chen, M. Smith, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads," *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996, pp. 12-21.

[9] M. Evers, P.-Y. Chang and Y. Patt, "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches," *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996, pp. 2-11.

[10] A. Srivastava and A. Eustace, "ATOM: A system for Building Customized Program Analysis Tools," *Proc. of PLDI'94*, June 1994, pp. 196-206.

[11] S.E. Perl and R.L. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces," *Proc. of the OSDI'96*, Seattle, October 1996.

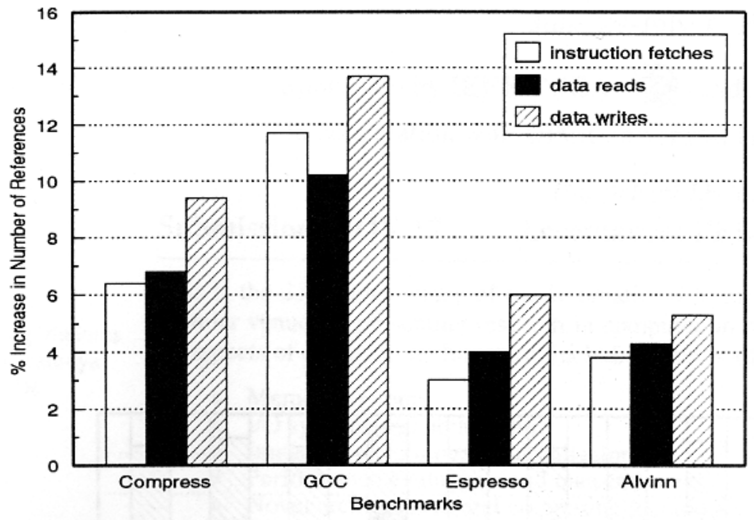

Figure 1: Percent of Total References From Operating System

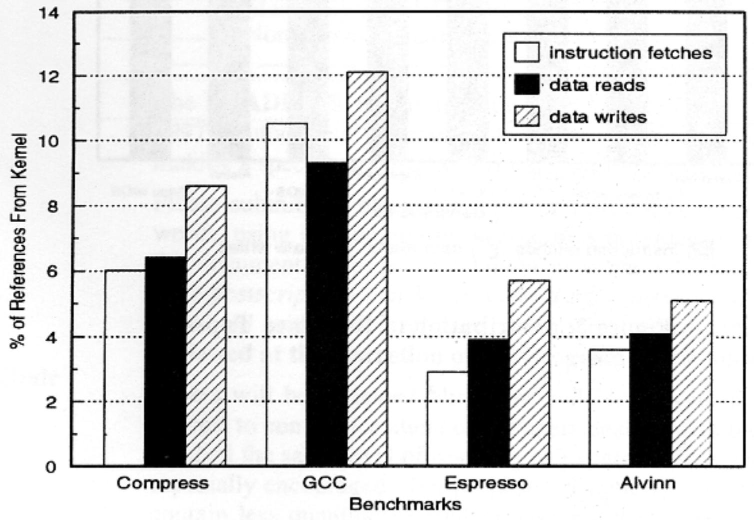

Figure 2: Percent Increase in Number of References by Including Operating System

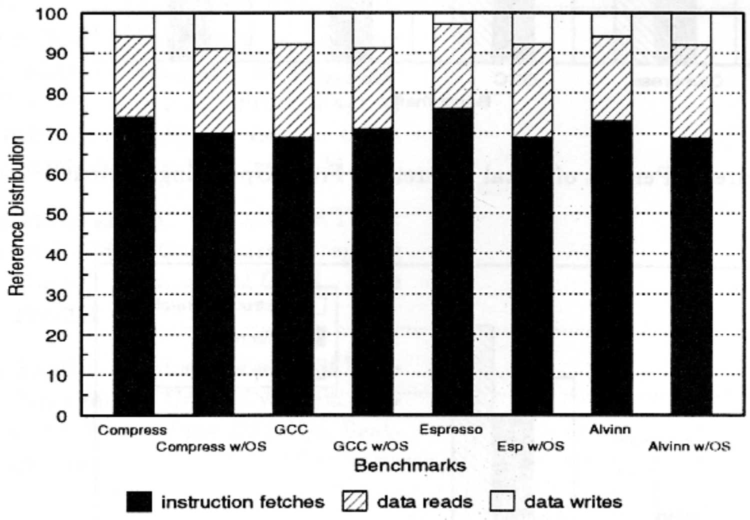

Figure 3: Distribution of Reference Types