Tracing and Characterization of NT-based System Workloads

Jason Casmira David Kaeli

Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA

David Hunter

Software Partner Engineering Group
Digital Equipment Corporation
Palo Alto, CA

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 (e.g., SPEC, Bytemark, SPLASH) have been studied extensively 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.

As applications become more dependent on services and APIs provided by the hosting operating system, the overall application performance becomes more dependent on efficient operating system interaction. It has been acknowledged that operating system overhead can greatly affect the benefits provided by a new design feature. The reason why the operating system interaction has, for the most part, been ignored in past architectural studies is the lack of available tools that can generate kernel-laden traces.

In this contribution we describe the ongoing efforts to capture operating system rich traces on the DEC Alpha platform. We will describe the current version of the PatchWrx toolset, originally developed by Richard Sites. This tool allows us to obtain trace information of application and operating system activity, while introducing minimal overhead.

We will describe the current version of the tool, and demonstrate its capabilities by characterizing a number of applications. We will also contrast the fundamental differences between using simple benchmark programs, versus studying application based programs. This paper demonstrates that for real applications (MS CD Player, MS Visual C/C++, FX!32, etc.), the operating system execution can dominate the overall execution time of the application, contributing significantly to the contents of any captured trace.

1 Introduction

Computer architects spend a significant amount of time studying the characteristics of benchmark programs. The underlying assumption is that these programs are representative of what people use computers for. The SPEC benchmark suite [24] and the Bytemark benchmarks [5] present two sets of programs which are supposed to generate workloads representative of user applications. While the objectives of authors of these benchmarks may be genuine, the resulting workload is far from representative of many of the most commonly used desktop applications (e.g., MS Word, MS Visual C/C++).

A major hole in benchmark-based workloads is their lack of operating system execution. Today's applications make heavy use of Application Programming Interfaces (APIs), which in turn, execute many instructions in the operating system. We argue that for traces of today's workloads to be accurate, they must capture the operating system execution, as well as the native application execution. This need to capture complete program trace information has been a driving force behind the development and use of tools such as the one described in this paper.

PatchWrx was originally developed by Richard Sites and Sharon Perl of Digital Equipment's Systems Research Center. The tool, as implemented on the Microsoft NT 3.5 operating system, is described in [21]. The tool was originally developed to study database workloads on multiprocessor systems. Together with a development group at Digital Equipment's Software Partner Engineering Group, we have continued development of the tool, providing trace capability on Microsoft NT 4.0, as well as beginning to analyze the characteristics of popular PC-based applications.

This paper will review the PatchWrx toolset, comparing it to other tracing tools, while demonstrating its utility. Section 2 begins with an overview of many of the issues related to trace capture. Section 3 presents the PatchWrx toolset and discusses related tools. Section 4 provides analysis of PatchWrx traces captured on Microsoft NT 4.0. The premise of this section is to demonstrate the inherent differences between benchmark traces and application traces. Section 5 describes two issues with the current development of this tool. Section 6 will summarize the paper and suggest new directions for the toolset.

2 Tracing

Trace-driven simulation has been the method of choice to evaluate the merit of various architectural tradeoffs [16, 32]. Samples (i.e., traces) of program execution are 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 [13, 15, 17, 20] and software [6, 8, 18, 27, 28] 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 [13, 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, 31], 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 [26], link time [25] or execute time [31]. 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 a fair representation of the workload and is not generally limited in length. Most software-based tracing methodologies 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 [27, 28]. 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]; an early study on the VAX used this methodology (called ATUM). Very few microprocessors still utilize microcode. The DEC Alpha uses a form of microcode called PALcode [34]. PALcode was developed to provide VAX compatibility and efficient system utilities. We have exploited this interface to allow for efficient trace capture. Some examples of PALcode functions include:

- context swapping,
- privileged instructions, and
- VAX-style interlocked memory access,

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.

3 PatchWrx

PatchWrx is a static binary-rewriting instrumentation tool for capturing full instruction and data address traces on the DEC Alpha platform running Microsoft NT. The toolset modifies the binary image prior to execution. No dynamic modification or emulation is performed. Traces are captured as the program runs. PatchWrx allocates a fixed trace buffer in physical memory at system boot time (the size of the buffer, currently set to 45 MB, can be expanded, and is only bound by the size of the physical memory installed). Trace records are captured when the modified executable encounters a modified instruction at run time. We also have the option of capturing load and store data addresses using a similar methodology.

PatchWrx uses a similar tracing mechanism as was earlier developed with ATUM for trace generation on the DEC VAX architecture [2]. ATUM was one of the first tracing systems which allowed for capture the operating system execution. ATUM modified the microcode of the VAX processor to provide trace hooks with a minimal amount of overhead. Agarwal describes the benefits of capturing operating system content using this strategy [1].

PatchWrx also employs microcode modification. The DEC Alpha processor uses a form of microcode called PAL (Priviledged Architecture Library). The Alpha architecture defines PALcode instructions, which trap to system level routines [34]. PALcode for the Alpha NT platform was modified and extended to support PatchWrx. Table 3 lists the PAL subroutines used by PatchWrx (taken from the original description of PatchWrx [21]).

PAL routine	Action/Trace record
PalReset	Reserve trace memroy
InterruptStackDispatch	next address, interrupt target
SoftwareInterrupt	next address, interrupt target
DispatchMmFault	next address, page fault target
UNALIGNED	next address, alignment fault target
RFE	return from exception target
CALLSYS	system call target
RETSYS	return from system call target
SWPCTX	new process ID
pwrdent	read trace entry from buffer
pwctrl	initiate tracing (toggle on/off)
pwbsr	branch entry
pwjsr	jump/call/return entry
pwldst	load/store base register entry
pwbrt	conditional branch taken
pwbrf	conditional branch fall through
pwpeek	(reserved for debugging)

Table 1: PAL subroutines used to support PatchWrx. The first set are modifications to existing PAL routines. The second set are new PAL routines.

PatchWrx modifies a binary image by identifying all branch points in the executable image, and then *patches* these branches, turning them into unconditional branches into a *patch section*. If the current instruction is either a BSR or JSR, then a BSR is used to branch to the patch section (this is necessary in order to preserve the return address register field). Otherwise, a BR instruction is used to effect the branch to the patch section. The patch section is additional code generated by PatchWrx, that is appended to the end of the executable. The patch section issues a PALcall (the type of call depends on the type of instruction that was patched), collecting the trace log entries which are later stored in the trace buffer. There are four possible formats for these entries, as as shown in Figure 1.

After a trace entry is stored to the buffer by the PALcode, the PALcode returns to the patch section at an address where a copy of the original branch instruction resides. Execution resumes at the target of the branch in the executable.

The PatchWrx trace records are written into the trace buffer section of memory as a patched program or system is run. The trace contains points in the execution which will later be used to reconstruct a full trace of the execution. These points in execution include conditional branches, unconditional jumps, task switches and interrupts, and loads and stores. Every unconditional jump will generate a trace record using the format shown in the first record in Figure 1. This format is also used to log interrupts. Load and store instructions will generate the second form of trace record, saving the base register value which will later be used to reconstruct the data address stream. The third record format in Figure 1 captures process switching, saving the new process id associated with the new context. The bottom format captures the results of up to 60 conditional branches, all in a single record. This provides for dense encoding of trace information.

The 45 MB buffer can capture 5-20 seconds of real time, depending on the type of patching specified and the characteristics of the the patched application. A single trace can capture up to 650 MB of

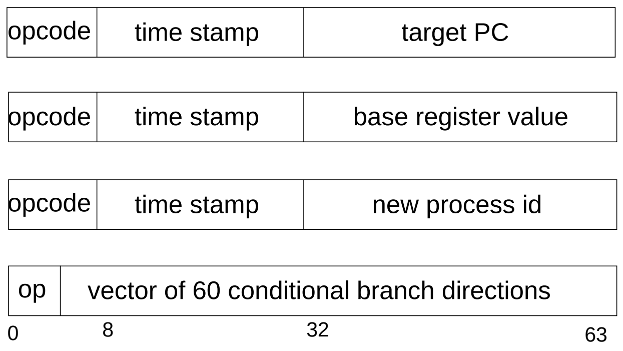

Figure 1: PatchWrx trace record formats. Each record is 64 bits long.

dynamic execution. By using main memory to store the trace, very little overhead will be introduced into the captured trace. If we instead attempt to dynamically save the trace to disk, this could sacrifice the integrity of the trace. A number of these issues are discussed in [16].

3.1 Operating System Rich Traces

As applications become more integrated with the operating system upon which they run, the need to capture operating-rich traces become greater. To accurately assess the effects of different design tradeoffs, application traces which contain not only the application code execution, but the operating system, library, etc., will need to be captured.

Most software-based tracing environments only capture the application code execution [10, 18, 23, 30]. Most hardware-based tracing systems allow for the capture of the entire execution [17], but the cost of such systems may be very high. A few software-based tracing system allow for the collection of operating system execution [8, 25, 27]. We discuss these systems next.

ATOM was developed by Srivastava and Eustace at DEC's Western Research Lab [25]. This tool allows for the selected instrumentation of executable binaries. The tool uses the Object Modifier technology developed at DEC. ATOM is a link-time code modification tool which inserts procedure calls to user defined analysis routines at link time. ATOM provides the ability to instrument a program at the procedure level, the basic block level, or the instruction level. ATOM is accompanied with a rich set of analysis routines [25]. 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*.

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 [8]. More recent work, which builds upon the work of Eustace and Chen as described in [11], extends ATOM to study the effects of the operating system impact on cache performance [14]. Some of the limitations of this work were:

1. the operating system could only run in single user mode,
2. substantial slowdown (10-100X) was introduced,
3. portions of the OSF kernel could not be traced,
4. floating point numbers or system calls can not be used in analysis routines, and
5. issues related to re-entrant code needed to be addressed.

These issues are addressed in [6] in detail.

The SimOS environment [27] is an emulation-based system, which models enough of the system to allow the operating system and applications to run correctly. While this system provides a flexible interface for collecting operating system-rich traces, the accuracy of the traces is suspect due to significant emulation overhead. SimOS is a simulation layer that runs on top of general-purpose Unix multiprocessors (i.e., Silicon Graphics Challenge series), simulating the SGI machine in enough detail to support running a real operating system (IRIX). Unaltered IRIX executables can then be run on top of the SimOS layer. Overhead rates of 10X-35X have been reported [28] (the amount of overhead is directly proportional to the complexity and accuracy of the simulated hardware layer).

3.2 Trace Accuracy

With respect to the accuracy of the traces captured with the PatchWrx toolset, there are a number of issues related to instruction addresses. Since patched program images are larger (due to a patch section which is appended to the bottom of the executable), shared-libaries will be loaded at a different address than they would normally (i.e., on an unpatched system). This can have an effect on the accuracy of cache simulations which use these traces. A second side-effect is that extra page faults may be incurred due to the increase in size of a patched executable. This effect should appear infrequently.

Since there is application slowdown due to the overhead associated with the trace capture patches, real-time events will occur more frequently (i.e., 4 times more frequently when capturing instruction traces). Studies by Sites and Perl on NT 3.50 found a 1% increase in the number of instructions executed [21]. This slowdown will also affect the queuing of external devices interacting with the patched system (e.g., disks, adapters, etc.). This can change when an interrupt service routine shows up in our reconstructed trace. Again, the effects of this seem to be minimal.

4 Characteristics of PatchWrx Traces

Next we will present the characteristics of traces obtained using the PatchWrx toolset. Our purpose here is not necessarily to answer specific architectural questions, but instead to demonstrate the benefits of providing this level of trace capability, and its importance as we move to studying applications run on NT-based platforms.

Lee et al. have recently reported on the characteristics of NT-based execution on the Intel x86 platform [19]. To capture these traces they used the Etch toolset [23]. While Etch is able to instrument most NT binaries, it is not able to instrument some DLLs, nor any of the NT kernel image. Lee et al. attempted to compare the characteristics of commercial desktop applications (e.g., MS Word, PowerPoint, Netscape), with execution from the SPEC'95 benchmark suite. The outcome of this study was to identify two distinguishing characteristics of desktop applications which were quite different than the benchmark suite:

1. the instruction working set size grows due to the fact that more calls are made to distinct functions, and

2. many more indirect calls are issued.

While we agree that these are valid conclusions, we argue that the picture here is incomplete. Since desktop applications spend a significant amount of time in the operating system and DLLs, we must evaluate this interaction to provide a complete picture. Next we present some of the characteristics of traces obtained with PatchWrx, and discuss the implications of these results.

4.1 PatchWrx Traces

To demonstrate the need for a tool like PatchWrx, we have captured traces from a variety of applications (in the final version of this paper we plan to have MS Word and possibly one other MS application included), and discuss various characteristics contained in them. The set of applications we report on are:

1. three desktop benchmarks (float - a small program which uses emulated floating point, neural - a small back-propagation network simulator, and bitfield - a set of bit-twiddling routines), from the Bytemark benchmark suite,

2. the MS Visual C/C++ compiler, compiling and linking a 3000 line C source code file,

3. the MS CD Player playing a music CD, and

4. the FX!32 program running the sample x86 arm2.exe executable. The x86 program is an openGL graphics-intensive application of a 3D solid model robot arm performing rotation motion simulation.

All traces were captured on a Alpha 21064 containing a 45 MB trace buffer, leaving 35 MB of available physical memory for the application, with only the single application of interest running during tracing. All applications were run in the foreground, and multiple traces were capture to establish the representativeness the of captured traces (which were found to be highly repeatable).

In Figure 2, we show the percent of execution time spent in the application (within the application .*EXE*), dynamic link libraries (only the DLL's associated with the traced .*EXE*), and NT kernel (and associated NT kernel DLLs, system services and drivers). Time is measured here in the number of instructions executed in each execution domain. As we can see, the three Bytemark benchmarks spend a majority of their time in their own .*EXE*, while the three real applications exhibit a very different characteristic. The Visual C/C++ compiler shows a fairly even split between .*EXE*, associated DLLs and the NT kernel. The CD player is dominated by the kernel. The FX!32 program spends most of the time in its own DLLs.

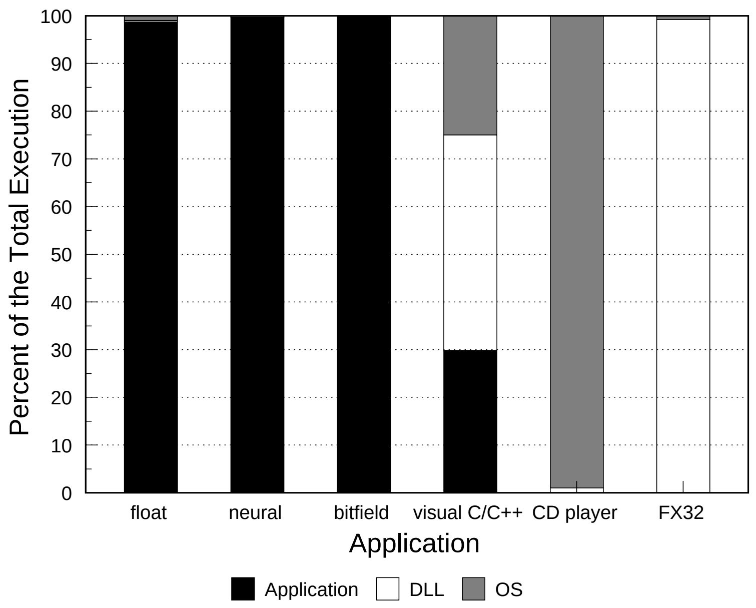

Figure 2: Execution breakdown, as a percent of the total number of instructions executed

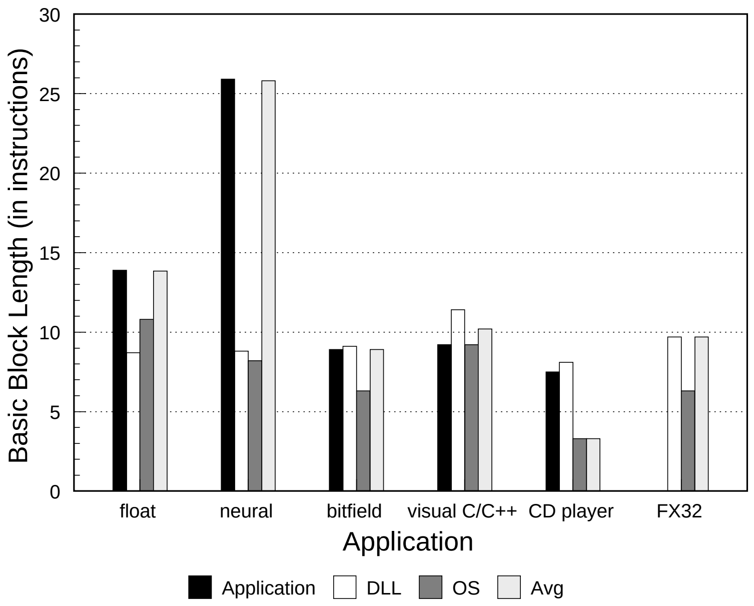

Figure 3: Average basic block length (in instructions)

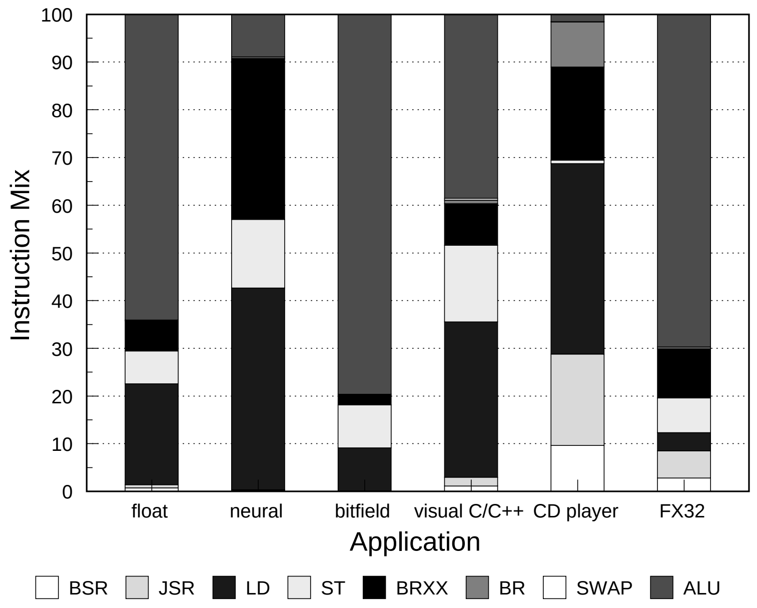

Figure 4: Instruction mix breakdown

The differences observed between the benchmark and the application programs are only significant if we see a marked difference in the characteristics of application versus DLL versus kernel execution. In Figure 3, we plot the average basic block length (measured in instructions) for the six applications. We break down the execution again into three categories, and see that the length of basic blocks in application code is much smaller than those found in operating system code (this is especially evident in the Bytemark benchmarks). (Note that the FX!32 traces did not contain any instructions from the *.EXE* associated with the application.

In Figure 4, we show the instruction mix for the six programs studied. This figure includes execution from all three domains. The instruction mix breakdown is as follows:

1. BSR - Branch to Subroutine
2. JSR - Jump/Return to/from Subroutine
3. LD - Loads
4. ST - Stores
5. BRXX - Conditional Branches
6. BR - Unconditional Branch
7. SWAP - SWAP Instructions
8. ALU - Arithmetic and Logical Instructions, user PALcalls

As we can see, each application exhibits its own characteristic mix. One general trend is that the real applications contain a lot more BSRs and JSRs. This is consistent with the results present by Lee

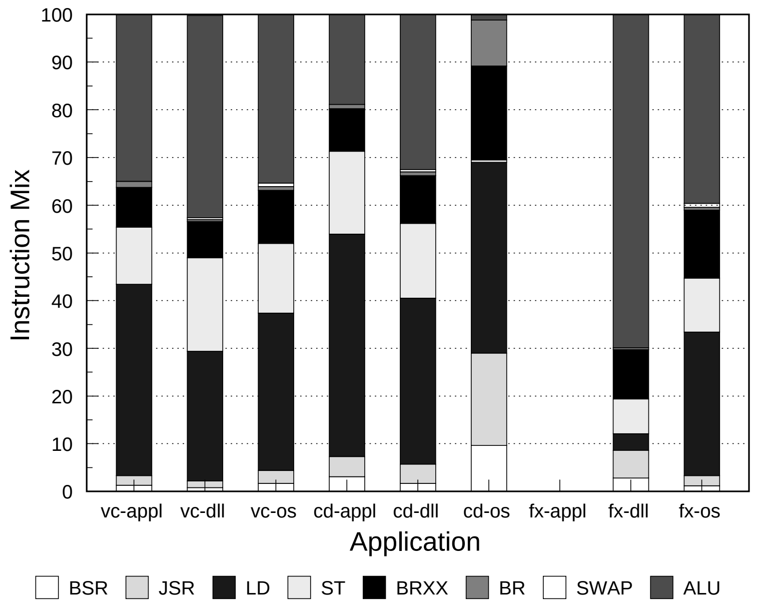

Figure 5: Instruction mix breakdown for the real applications. appl = application-only, dll = the DLL of the associated application, and os = the NT kernel

et al. [19]. In Figure 5 we show the instruction mix for the three real applications, further classified by the three execution domains (1) application-only, (2) application DLLs, and (3) the NT kernel (and OS-related images). What we can see is that, while both the DLLs and the kernel contain a significant number of indirect jumps and calls, the applications (at least for Visual C/C++ and the CD player) contain a significant number of indirects as well.

Even from this small sample of applications, we can see that there are marked differenes in the execution profile between the benchmark and real applications, as well as differences of captured execution within the application, versus execution patterns in the DLLs and NT kernel. The value of obtaining a more complete (and accurate picture) of entire program execution cannot be ignored. The results of any trace-driven simulation are only as good as the traces used as input to the evaluation. The PatchWrx tool provides an accurate picture of execution, so that the effects of the operating system execution need not be modeled as an afterthought [12].

5 Discussion

There are a couple of issues which we are presently working on to extend the utility of the PatchWrx toolset. We describe these next.

5.1 Long Branches

When we attempt to patch a branch (which is done by using a BR or BSR instruction) we only have a 21-bit signed displacement field provided (as defined in the Alpha ISA) to branch to the patch section of code. If we attempt to patch an executable that is larger than 4 MB (as is the case with some NT desktop applications), we need to employ a different strategy for patching the executable.

To address this issue, two new PatchWrx PALcalls were developed. Using these new instructions, calls are made directly from the original executable to the PALcode, passing information in the instruction format regarding the displacement into the patch area. These PALcalls provide the ability to patch executables larger than 4 MB. For non-call patching, we use a new PALcall which can specify a signed 25-bit displacement field. This gives us the ability to patch executables that need to branch to a patch section which is 32 MBs away in the address space. For call patching, we do not have the same luxury since we need to preserve the 5-bit register field for the return address register. To handle this, we block branch targets within the patch section on 32 byte boundaries, which provides a displacement of 32 MB for call patching. While blocking will create larger executables, the effect is not great enough to be prohibitive. This technique can be expanded to use a larger blocking factor if executables grow to larger than 32 MB.

5.2 Portability

One issue that limits the ease of use of PatchWrx is the fact that the tracing mechanism is processor version dependent. This is only true because the PALcode is also processor version dependent. We are currently working towards a Alpha 21164 version of PatchWrx which will allow us to trace on two versions of the processor.

A second issue is that when we move to a new version of the NT operating system, some porting of the patching code is necessary. This is true mainly to changes that occur in the non-documented portions of the NT kernel. We anticipate that many of the issues will be resolved as we work more closely with Microsoft in the future.

6 Summary

In this paper we have described the PatchWrx toolset. We have compared it to existing tools, and demonstrated the need for operating system rich traces by showing the amount of the total execution spent in the kernel and DLLs. We have also shown that existing desktop benchmarkss do not exercise the kernel and DLL enough to provide meaningful indicators of desktop performance.

In future work we hope to report on a wider range of desktop applications. We also plan on moving this work to the Alpha 21164 and Microsoft NT 5.0.

We would like to acknowledge the help and advice of the following people: Richard Sites, Sharon Smith, Geoff Lowney, and Joel Emer, all from DEC and Robert Davidson from Microsoft Research.

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] 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.

[4] 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.

[5] See the following URL for more information on the Bytemark benchmark suite: http://www.byte.com/bmark/bmark.htm

[6] J. Casmira, D.R. Kaeli, and D. Hunter, “Operating System Impact on Trace-Driven Simulation,” *Proc. of the 31st Annual Simulation Symposium*, Boston, MA, April 1998, to appear.

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

[8] 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.

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

[10] R.F. Cmelik and D. Keppel, “Shade: A Fast Instruction-Set Simulator for Execution Profiling,” *Proc. of ACM Sigmetrics*, May 1994, pp. 128-137.

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

[12] M. Evers, P.-Y. Chang, and Y.N. Patt, “Using Hybrid Branch Predictors to Improve Branch Predication Accuracy in the Presence of Context Switches,” *in the Proc. of the 23rd Annual International Symposium on Computer Architecture,* Philadelphia, PA, May 1996, pp. 3-11.

[13] 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.

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

[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] D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad, “Execution Characteristics of Desktop Applications on Windows NT,” *To appear in the Proc. of the 25th International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[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] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy and B. Bershad, “Instrumentation and Optimization of Win32/Intel Executables Using Etch,” *Proc. of USENIX Windows NT Workshop,* August 1997, pp. 1-8.

[24] SPEC Newsletter, September 1995.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] A.J. Smith, “Two Methods for the Efficient Analysis of Memory Address Trace Data,” *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, January 1997, pp. 94-101.

[30] M. Smith, “Tracing with Pixie,” *Technical Report*, Stanford University, Stanford, CA, 1991.

[31] 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.

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

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

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