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 paper we describe the ongoing efforts to capture operating system rich traces on the Digital Equipment Corporation's Alpha-based machines running the Windows NT operating system. We will describe the current version of the toolset and demonstrate its capabilities by characterizing a number of applications. We will also contrast the fundamental differences between using traces captured from common benchmark programs, versus those captured on commercial desktop applications. This paper demonstrates that for commercial desktop applications (Microsoft Word, Microsoft Visual C/C++, Microsoft Internet Explorer, etc.), the operating system execution can dominate the overall execution time of the application, and that the characteristics of the operating system instruction stream can be quite different than those typically found in benchmarking workloads.

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 [1] and the Bytemark benchmarks [2] present two sets of programs which are supposed to generate workloads representative of user applications. While the objectives of the authors of these benchmarks may be genuine, the resulting workload is far from representative compared to some of the most commonly used desktop applications (e.g., MS Word, MS Visual C/C++, MS Internet Explorer).

A major drawback of using 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 toolsets such as the one described in this paper.

PatchWrx was originally developed by Sites and Perl at Digital Equipment's Systems Research Center. The toolset, as implemented on the Microsoft Windows NT 3.5 operating system, is described in [3]. Working together with Digital Equipment's Software Partner Engineering Group, we have continued development of the toolset, providing trace capability on Microsoft Windows NT 4.0, as well as adding a number of features to the toolset.

This paper will review the PatchWrx toolset, comparing it to other tracing tools, while demonstrating its utility. Section 2 briefly reviews related tracing tools. Section 3 presents the PatchWrx toolset and describes new features we have added. Section 4 provides analysis of PatchWrx traces captured on the Microsoft Windows NT 4.0 operating system, both demonstrating the capabilities of the tool, while illustrating the importance of capturing operating system rich traces. Section 5 will summarize the paper, discuss current limitations of the toolset, and suggest new directions for development and study.

2 Trace Generation Tools

Trace-driven simulation has been the method of choice to evaluate the merits of various architectural tradeoffs [4, 5]. 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 tracing methodologies have been proposed which capture both application and operating system references. These tools include both hardware [6, 7, 8, 9, 10] and software [11, 12, 13, 14, 15] based methods. Some of the issues involved with capturing operating system-rich traces include:

1. tracing overhead (system slowdown),
2. accuracy (perturbation of the memory address space), and
3. completeness (capturing all desired information, e.g., operating system).

In Table 1 we have listed a number of the tracing tools that have been developed over the past 10-15 years. This list if far from complete, but provides a sample of the tools that have been used to generate input to a variety of trace-driven simulation studies. We have tried to characterize each tool based on the above criteria. We have also identified the target platform(s) for each tracing tool.

Note that many of these tools can not capture operating system activity. For those that can, their associated slowdown can significantly affect the accuracy of the captured trace. Of the tools that provide this capability, PatchWrx introduces the least amount of slowdown, while maintaining the integrity of the address space. Next we discuss the PatchWrx toolset.

Name	Avg. Slowdown	Addr. Perturb	OS Activity	Platform
ATOM [13]	10X - 100X	N	Y	Digital Alpha UNIX
ATUM [16]	20X	N	Y	Digital VAX VMS
EEL [17]	10X - 100X	Y	N	SPARC Solaris
Etch [18]	35X	Y	N	Intel x86 MS Windows NT 4.0
PatchWrx [3]	4X	N	Y	Digital Alpha MS Windows NT 4.0
Pixie [19]	10X - 100X	Y	N	Digital MIPS Ultrix
NT-Atom [20]	10X - 100X	N	N	Digital Alpha MS Windows NT 4.0
QPT [12]	10X - 100X	Y	N	SPARC Solaris
Shade [21]	6X	N	N	SPARC Solaris
SimOS [14]	10X - 50,000X	N	Y	Digital Alpha UNIX SGI IRIX SPARC Solaris

Table 1: Sample of Tracing Tools

3 PatchWrx

PatchWrx is a dynamic execution tracing toolset developed for use on the Alpha-based Microsoft Windows NT operating system. PatchWrx utilizes the PALcode of the Alpha microprocessor to perform tracing with minimal overhead. It can instrument (i.e., *patch*) all NT application and system binary images, including the kernel and operating system services, drivers, and shared libraries. The Privileged Architecture Library (PAL) facility of the Alpha is a set of architected functions and instructions that provide a consistent interface to a set of complex system functions [22]. These routines provide primitives for memory management, context switching, interrupts, and exceptions.

3.1 PatchWrx and the Alpha PALcode

The PatchWrx tool is made possible by the use of the Privileged Architecture Library (PAL) [22] used in DEC Alpha microprocessors. PAL routines have access to physical memory and internal hardware registers, and operate with interrupts disabled. PALcode is loaded from disk at system boot time. The shrink-wrapped Alpha PALcode on our 21064-based system was modified and extended to support the PatchWrx operations. The new and modified PatchWrx PAL routines serve two major purposes: 1) reserving the trace buffer at system boot time, and 2) logging trace entries at trace time.

One way that PatchWrx maintains a low operating overhead is to store the captured trace in a physical memory buffer, which is reserved at boot time. The size of the buffer is only dependent upon the amount of physical memory installed on the system. Since we use PALcode routines to reserve this memory, the operating system is not aware that it even exists, as the PALcode performs all low-level system initialization before the operating system is started.

All trace entries are logged in this trace buffer. Writing trace entries directly to physical memory has several advantages. First, writing to memory is much faster than writing to disk or tape. Second, using physical memory allows tracing of the lowest levels of the operating system (i.e., the page fault handler) without generating page faults. Third, using physical memory allows tracing across multiple threads running in multiple address spaces without worrying about which address space is currently running.

In order to capture all system activity, nine of the standard Alpha PALcalls were modified, as

described in [3]. In addition to modifying existing PALcalls, new routines were added to enable tracing of operating system and application images. The new PALcalls, and their corresponding descriptions, can be found in Table 2. The first three PALcalls are used for reading the trace entries from the buffer and turning tracing on and off. The remaining five new routines are used to log trace entries based upon the type of instruction instrumented.

PALcall	**Function**
pwrdent	read a trace entry from trace memory
pwpeek	read an arbitrary location (for debug)
pwctrl	initialize, turn tracing on/off
pwbsr	record a branch to subroutine
pwjsr	record a jump/call/return
pwldst	record a load/store base register value
pwbrt	record a conditional branch taken bit
pwbrf	record a conditional branch fall-through bit

Table 2: New PALcalls

3.2 PatchWrx Image Instrumentation

Next, we describe how we use PatchWrx to instrument Microsoft Windows NT images. The first phase of *patching* the operating system is the instrumentation of all the binary images, including the applications, operating system executables, libraries, and kernel. Once patching is complete, trace entries will be logged via PALcalls as images execute.

We define a *patched instruction* as an instruction within an image's code section that is overwritten with an unconditional branch to a *patch*. We define a *patch* as the PALcall which logs a trace entry corresponding to the type of instruction patched and the return branch to the original target.

PatchWrx does not modify the original binary images but instead generates new images containing patches. This preserves the original images on the system in case they need to be restored for any reason. Instrumentation involves replacing all branching instructions of type conditional/unconditional branch, branch to subroutine, function call, function return, jump, and jump subroutine within an image's code section with unconditional branches into a *patch section*. The patch section is an additional section added to the rewritten image that contains all the patches for the image. Examples of patched instructions are shown in Figure 1. We replace only branch instructions within an image to reduce the number of entries logged in the trace buffer. Using these traced branches, we can later reconstruct the basic blocks they represent.

Unconditional branches (BR) and jumps (JMP) are replaced with unconditional branches that transfer control to the patch section. The original branch or jump is repeated in the patch section for purposes of recording the value of the target register (if necessary) into the trace buffer when the patched image is executed. This register value is necessary as it serves as a means to reconstruct the traced instruction stream. Jumps to subroutines and branches to subroutines are replaced with branch to subroutine patches. This is done to preserve the RA register field value, which contains the return address for the subroutine. Again, the original instruction is repeated in the patch section for register value recording during tracing to help facilitate reconstruction.

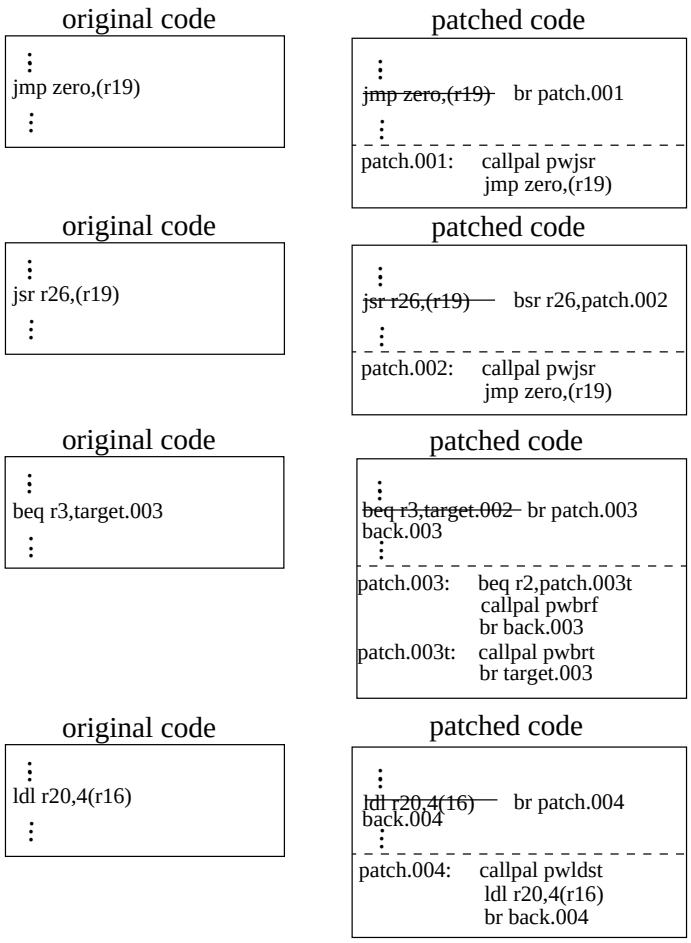

Figure 1: Instruction patch examples for a jmp, jsr, beq and ldl instruction

Conditional branches have a larger and more complex patch than the other branch types. This is because the original condition is duplicated and resolved within the patch. The taken or fall-through path generates a bit value when logged within the taken/fall-through trace entry. The return branch in the patch section is just a replica of the original conditional branch. Load and store instructions are replaced with unconditional branches to the patch section, where the original load (or store) is copied. A return branch is also needed to return control flow to the instruction subsequent to the original load. When this patch is encountered, the register value of the original load or store instruction is recorded in the trace log. For all patches, the original branch is replaced with a *patch* unconditional branch. Since Alpha instructions are all equal in size, this allows branch patching without causing any increase in code size within the image. Although the code size remains unchanged, the *image size* will increase in size proportional to the number of patches added. This becomes an issue for dynamically linked library images.

Microsoft Windows NT provides a memory management system that allows sharing between processes [23]. For example, if two processes edit text files, they can share the text editor application image that has been mapped into memory. When the first process invokes the editor, it is loaded into memory and that process's virtual address space is mapped to it. When the second process is invoked, rather than loading another editor image, the second process's virtual address space is mapped to the physical pages containing the editor. Of course, both processes contain local storage for private data.

DLLs are loaded into memory and shared in this manner. When adding patches to a DLL, the size of the image increases (causing *image bloat*). When this image is mapped to physical memory (as per its preferred base load address), due to its increased size, this larger image may overlap with another image having a base address within the new range. This can prevent the operating system from booting properly, as some of the environment DLLs will conflict in memory. This becomes an issue because some DLLs perform calls directly into other DLLs at fixed offsets. To resolve this issue, we *rebase* [24] the preferred base load addresses of the patched DLLs. Doing this modifies the base load addresses to eliminate conflicts. This will affect the address accuracy of the patched system, though we are able to account for rebasing during trace reconstruction.

The original version of PatchWrx was developed on Microsoft Windows NT version 3.50. When the 3.51 version, and later 4.0 version, were released, the linking convention used by Microsoft to create the system images changed. The new linker allows for data to reside within the code regions (in an effort to condense the number of memory pages occupied by an image). In developing the 3.51 and 4.0 compatible versions of PatchWrx, this issue had to be addressed. One change affecting how we patch was the placement of the Import Address Table (IAT) into the front of the initial code section of executable binary images. This table is used to lookup the addresses of DLL procedures used (i.e., imported) by the executable binary. In the development of the current generation of PatchWrx, modifications had to be made to utilize image header fields that had previously remained unused or reserved, indicating the executable code sections that contained data areas.

Another issue that was addressed in the recent modifications to PatchWrx was long branches. In the original version of PatchWrx, all branch, jump, call, and return instructions were replaced with either branches or branch-subroutines to the patch section. Since the PatchWrx tool has no information about machine state during the patching phase, it is impossible to utilize other branching instructions (e.g., jump or jump-subroutine instructions), to provide this branch-to-patch transition. Register and register-indirect branching instructions would require perturbing the machine state. Therefore, only PC-based offset branching instructions could be used.

As discussed previously, when a control flow instruction is replaced with a PatchWrx patch branch, a BR or BSR instruction is used with the offset field set to branch to the corresponding patch within the image's patch section. The Alpha architecture branching instructions use the following format:

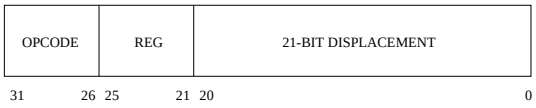

Figure 2: Alpha BR/BSR Format

The branch target virtual address computation for this format is:

$$\text{newPC} = (\text{oldPC} + 4) + (4 * \text{sign-extended}(\text{21-bit branch displacement})) \quad (1)$$

The register field is used to hold the return address for BSRs. With this branch format and target virtual address computation, the Alpha architecture provides a branch target range of 4MB from an instruction's current PC.

Several of the applications that run today on Microsoft NT 4.0 are sufficiently large that the displacement between a control flow instruction to be patched and the patch location within the patch section exceeds this 4MB limit. (Recall that since we want to avoid moving code or data sections, the patch section is placed at the end of the image.) To address this problem, two new branch instructions were developed for use with PatchWrx. These new branches were not implemented in the instruction set architecture of the Alpha. Instead, we used the PALcode facilities to implement them. The two new branches are designated Long Branch (LBR) and Long Branch Subroutine (LBSR). The format of these two instructions is:

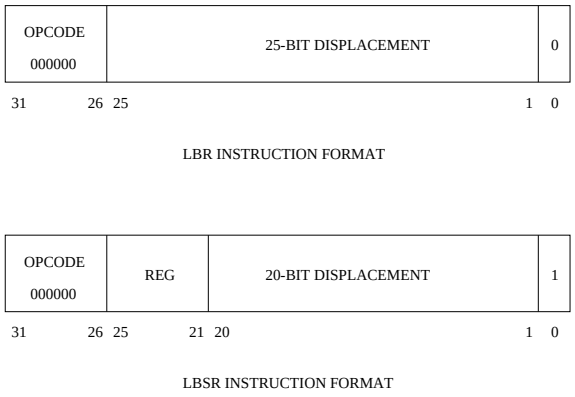

Figure 3: Long Branch Formats

and the computation of the target virtual address is:

$$\text{newPC} = (\text{oldPC} + 4) + (4 * \text{sign-extended}(\text{25-bit branch displacement})) \quad (2)$$

for LBR branches and

$$\text{newPC} = (\text{oldPC} + 4) + (32 * \text{zero-extended}(\text{20-bit branch displacement})) \quad (3)$$

for LBSR branches.

LBRs are used when patching any control flow instruction that has a displacement greater than 4MB. LBSRs are used similarly for control flow instructions that must preserve the register field value. Figure 3 shows that the two branches share the same PALcode opcode (000000). When an LBR or LBSR is executed within the image code section, a trap to PALcode occurs. Normally, Alpha PALcalls have one of several defined function fields which cause a corresponding PAL routine to be executed. The long branches have *function fields* that do not belong to any of the defined PALcalls, and therefore force an illegal instruction exception within the PALcode. This PALcode flow has been modified to detect if a long branch has been encountered.

As shown in Figure 3, both branch types have the same opcode value. In order to distinguish between the two, the least significant bit in the instruction word is set to zero for LBRs and to one for LBSRs. This bit is not included as a usable bit for the displacement fields of either branch type. This leaves a 25 bit displacement field for LBRs and a 20 bit field for LBSRs. With a 25 bit usable displacement field, the PALcode performs the above LBR target address computation, allowing a $\pm$ 64MB range.

Although the LBSR branch has only a 20 bit displacement field (versus the original Alpha architecture branch displacement field of 21 bits), the target instruction address computation is performed differently than for standard branches within the PALcode. As shown above, the 20 bit displacement is multiplied by 32 rather than 4. Notice that the 20 bit displacement is always zero extended. This computation provides the LBSR instruction with a displacement of +32MB. There are two implications of this computation procedure. First, LBSRs can only be used to branch from an image code section to an image's patch section. As discussed previously, patch sections are always placed at the end of an image. Also, branches into the patch section are always either branches or branches to subroutines (or their long displacement counterparts); only branches (or long branches) are used to return from the patch section to the original branch target (branches to subroutines are never used) within a code section. Therefore, restricting LBSRs to use positive displacements does not present a problem.

However, the LBSR displacement multiplier value of 32 (versus the original value of 4) does present some restrictions. The multiplier value of 4 used in the original Alpha ISA represents the ISA's instruction word length of 4 bytes. This means that normal branch instruction target addresses must be aligned on a 4 byte boundary. By using the multiplier value of 32 for LBSR instructions, LBSR target addresses are restricted to align on a 32 byte (i.e., 8 instruction) boundary. Since all LBSR targets reside within the patch section, this restriction does not pose a problem. If an LBSR is to be inserted into the image code section, and the next available patch target address is not aligned properly, NOOP instruction words can be inserted (and the next available patch target address advanced) until the necessary alignment is achieved. The NOOPs are never actually executed, they are inserted only for alignment purposes. Although inserting these NOOP instructions increases the amount of image bloat, several optimizations have been implemented into the instrumentation algorithm to keep this bloat to a minimum. For example, a queue is used hold LBSRs that do not align. As LBR patches are committed, the queue is probed to determine if any LBSRs align from their origin to the newly available patch target offset.

3.3 Trace Capture

The PatchWrx toolset allows the user to turn tracing on and off. The tracing tool can be started and stopped at any point in time, allowing capture of any portion of workload execution. This piece of the toolset is also responsible for utilizing the *pwrdent* PatchWrx PALcall to copy trace entries from the physical memory buffer to disk. Copying the trace buffer to disk is done after tracing has stopped so that the time required to perform the copy does not introduce any overhead during trace capture.

Each patch logs a trace entry as it is encountered during program execution. As instructions are executed within the code section, an unconditional PatchWrx branch is encountered. Instead of branching to the original target, the patched branch transfer control to the image's patch section. Within the patch section, a PatchWrx PALcall traps to the PALcode routine corresponding to the patch type, and logs a trace entry to the trace buffer. The PALcode then returns to the instruction following the PALcall. An unconditional branch is used to transfer control from the patch section back to the original target within an image code section. It is during the execution of the PatchWrx PAL routine that necessary machine state information is recorded and logged in the trace buffer. This allows for the capture of register contents, process ID information, etc., that is later used during trace reconstruction.

The trace capture facility captures the dynamic execution of a workload running on the system. To reconstruct the trace after it has been captured, a snapshot of the base load addresses of all active images on the system must also be captured. This snapshot serves as the virtual address map (VA map) used in reconstructing the trace. Each active process and its associated libraries are loaded into a separate address space, which may be different than the preferred load address as specified statically in the image header. If each image was loaded into memory at its preferred base address, we would not need the VA map to perform reconstruction. Instead we could map target addresses from the trace buffer using the base address values contained in the static image headers.

The type of trace record logged into the trace buffer is dependent upon the type of branch or low level PAL function being traced. The trace record formats are shown in Figure 4. The first three trace entry formats consist of a 8-bit opcode and a 24-bit timestamp. The timestamp is the low-order 24 bits of the CPU cycle counter. The 32-bit field of these three formats depends upon the type of trace entry logged. The first format is used for target virtual addresses for all unconditional direct and indirect branches, jumps, calls, returns, interrupts, and returns from interrupts. The 32-bit field of the second format is used to record the base register value for traced load and store instructions and stack pointer values that are flushed into the trace buffer during system calls and returns. The 32-bit field of the third format is used for logging the current active process ID at a context swap.

The fourth trace entry type is used for tracing conditional branches. It uses a 3-bit opcode, followed by up to 60 taken/fall-through bits. A "start bit" (SB) is used to determine how many bits are active. A "1" is recorded if a conditional branch is taken, a "0" for not-taken. This allows a compact encoding of conditional branch trace entries. During trace reconstruction, conditional branch trace entries are used to reconstruct the correct instruction flow when conditional branches are encountered, as well as providing concise information about when to deliver interrupts in loops.

Figure 4: Trace Entry Formats

3.4 Trace Reconstruction

The reconstruction phase is the final step in generating a full instruction stream of traced system activity. Trace reconstruction requires several resources in order to generate an accurate instruction stream of all traced system activity, as shown in Figure 5. First, the heading of the captured trace is read. This heading is initialized by the trace capture tool, and includes a timestamp, the name of the user who captured the trace, and any important system configuration information (OS version number, etc.).

Next, the first four raw trace records are read. These first four records are automatically entered whenever tracing is turned on. They contain the first target virtual address, the active process ID, the value of the stack pointer, and the first taken/fall-through (TF) record to be used (TF records always precede the branches they represent). Using this information, the necessary data structures of the reconstruction process are initialized.

Using the first target virtual address and process ID pair from the captured trace, the virtual address map is consulted to determine which image the instruction falls within (based upon its dynamic base load address), and where that image is physically located on the system. The patched image is consulted to determine the actual instruction at the target address. This instruction is recorded and the next instruction is then read from the patched image. This process is continued until either a conditional or unconditional branch is encountered. If a conditional branch is encountered, the first active bit of the current TF entry is consulted to determine subsequent control flow, and the process is continued at that address. If an unconditional branch is encountered, it is recorded and checked against the next captured trace entry. If the two match, the recorded instructions are output to an instruction stream file, the captured trace entry is consulted for the next target instruction virtual address, and the process is repeated. This continues until the entire captured trace is processed.

Since interrupts and other low-level system activities (e.g., page faults) are captured in the

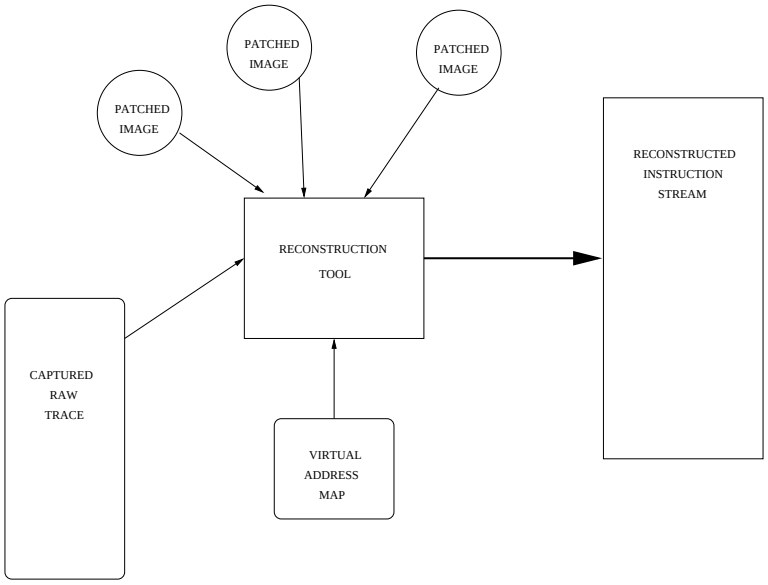

Figure 5: Instruction Stream Reconstruction Resources

trace, they must also be reconstructed. When an interrupt is logged into the trace buffer, the corresponding target virtual address in the captured record represents the address of the first instruction *not* executed when the interrupt was taken. The currently active TF entry is flushed to the memory buffer and a new TF entry is initialized. This new TF entry will be responsible for the conditional branches encountered beginning with the interrupt service routine. The address of the first instruction within the interrupt service routine is then logged in the trace.

During reconstruction, we will look for the interrupt's first unexecuted instruction address to know which instruction to stop at when reconstructing the instruction stream. We can then begin reconstructing the instruction stream, including the interrupt handler stream. If the unexecuted instruction is within a loop, we can utilize our TF entry convention. We know that when the interrupt was taken, the active TF record was flushed and another was started. This allows us to simply continue to reconstruct iterations of the loop until we exhaust all of the TF bits.

4 OS-Rich Workload Characterization

As presented in Lee et al.'s study [18], desktop applications and benchmarks share some workload characteristics, but applications alone do not represent full system behavior. In order to investigate and address system design issues, operating system-rich traces should be used.

To illustrate this point we will present a sample of various workload characteristics present in a selected a set of benchmark and desktop applications to study the differences in the use of the operating system and related services. The first characteristic we will discuss is the amount of time each benchmark or desktop application spends within its own executable application, within its associated DLLs, and within the operating system (utilizing some service). This provides insight into the workload's use of the each domain. We will also examine DLL and system service usage on an image basis for each workload. From this we can more clearly identify the dependence between

the workload and the system services provided by NT.

We also present the instruction mix of each workload with and without the inclusion of the operating system execution. Understanding the differences in instruction composition in the presence of system activity will further highlight the behavior we miss in application-only traces, such as increases in branches and memory instructions, when compared to application-only workloads. Average basic block lengths are presented for each domain of execution (application-only, DLLs, OS) separately, and then in combination. This metric reveals which workload domain dominates the branching behavior. We point the interested reader to [25] for a more complete description of these differences across a wider set of workload characteristics.

4.1 Workload Descriptions

All of our experiments were performed on a Digital Equipment Corporation Alpha platform running the Microsoft Windows NT 4.0 operating system. Traces were captured on a 150 MHz 21064 Alpha processor. The system was configured with 80 megabytes of physical memory. The workloads we consider in this paper are listed in Table 3.

Workload	**Description**
fourier	BYTEmark; numerical analysis routine for calculating series approximations of waveforms
neural	BYTEmark; a small, functional back-propagation network simulator
go	SPEC95 Go! game benchmark
li	SPEC95 Lisp interpreter benchmark
cdplayer	Microsoft CD player playing a music CD
fx32	FX!32 V1.1 interpreting/translating included openGL sample x86 application
ie	Microsoft Internet Explorer V2.0 following a series of web page links
vc50	Microsoft Visual C++ 5.0 compiling a 3000 line C program
word	Microsoft Word97 V7.0, spell-checking a 15 page document

Table 3: Workload Description

The fourier and neural benchmarks are from the BYTE benchmark suite. Neural is a small array-based floating-point test. Fourier is designed to measure transcendental and trigonometric FPU performance. Both go and li are from the SPEC95 integer benchmark suite. Go is a simulation of the game *Go!*, with the computer playing itself. Li is a Lisp interpreter. All of the benchmarks use the standard inputs provided with the benchmarks, and are compiled with the default optimization level using the native Alpha version of Microsoft Visual C++ version 5.0.

The CD player is the Microsoft CD player included on the release of Microsoft Windows NT 4.0 distribution. It was traced while playing a music CD using default playing options (e.g., playing all songs in order).

FX!32 is the emulator/translator provided by Digital Equipment Corporation's Alpha Migration Tools Group [26]. Version 1.1 was used in this work. The robot arm openGL sample Intel-based application was run in the foreground during trace capture.

The Microsoft Internet Explorer workload is the standard 2.0 version included on the Microsoft Windows NT 4.0 distribution. It was traced while traversing several links through the Sony home

page, arriving finally at the Sony PlayStation Store page. Four links were followed in all. The trace was captured on 5/4/98 (pages may have changed since this date). The history cache and the web link cache were both empty when the trace was captured.

The Microsoft Visual C++ V5.0 compiler was traced while compiling a 3000 line C source code file. The command line interface was used. Defaults for optimization levels and other parameters were used, as they should best represent the common usage of the compiler.

Microsoft Word version 7.0 of the MS Office97 desktop application suite for the Alpha was used to capture a manual spell check of a 15 page Word-format document. The standard Microsoft Word dictionary was used.

In order to provide a clear and representative comparison of workload behavior, several traces were captured. For all scenarios, full traces of each workload captured approximately 5-10 seconds of execution, filling the 45MB trace buffer. To characterize workload behavior, each experiment was run with the benchmark or application as the only activity on the system. Each workload was run in the foreground.

To insure that the traces captured were representative of the overall workload behavior, several traces were captured. Different points during execution were chosen for tracing to allow comparison between different portions of the selected scenarios. In order to investigate the variability present in selected workloads, additional scenarios were traced. A second trace was captured of Microsoft Word performing an autoformat operation of the same document used in the first trace of the spell check operation and for Microsoft Internet Explorer, a second trace was captured of repeating the Sony links but with the links cached. A second trace was captured of FX!32 using the included *boggle* sample game (for comparison against using the openGL application input). Additionally, the FX!32 translator was traced while it optimized a native Intel x86 application's profile. Refer to [26] for an explanation of FX!32's emulation and translation/optimization procedures. Discussion of these and additional scenarios can be found in [25].

4.2 Domain Mix

To illustrate the inherent differences between benchmark and desktop application behavior, we breakdown the captured trace in terms of three mutually-exclusive *domains*. These domains are the 1) application, 2) DLLs, and 3) operating system. The application domain represents the set of executed instructions that are within the traced application's executable image. The DLL domain represents the instructions executed by the application of interest's process, but excludes the application's executable image. This domain is made up of the DLLs, system services, and drivers that the application may access during its execution. The operating system domain includes instructions executed by the kernel or other system support service executable images, and all associated DLL and driver images. These are the processes, images, and libraries that are always present and running on the system. Figure 6 displays the breakdown of instructions into these three domains. On the x axis the workloads are listed, and on the y axis the percent composition of the captured trace is presented. We see that the benchmarks (i.e., fourier, neural, go, li) spend at least 95% of their execution within their application image. Both fourier and neural spend about 99% of their execution within their application image. Go and li do exhibit some operating system activity, but this is due to the I/O generated as go displays output as it progresses, and as li reads input from its standard input file.

The operating system dominates the execution in the CD player. This application is I/O bound,

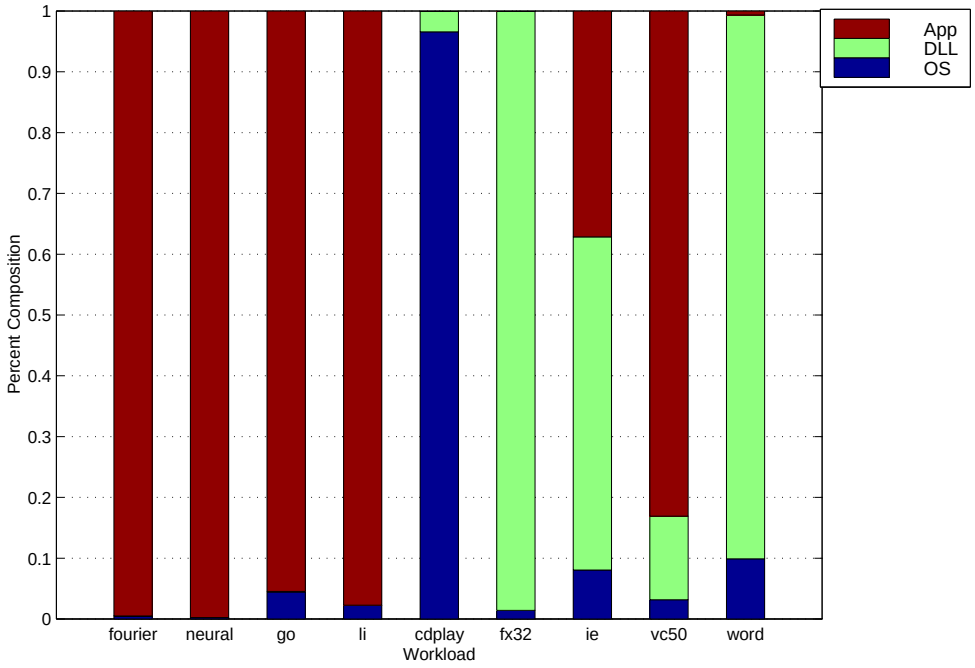

Figure 6: Domain Execution Mix

relying heavily on the necessary services provided by the OS and DLLs to access the CD hardware. While waiting for I/Os to complete, the system activity is composed almost completely of the kernel idle loop performing *busy waiting* (recall that each workload investigated is the only application running on the system, so there is no other work to be done during these periods).

FX!32 spends nearly all of its time operating within DLLs. The *robot arm* Intel x86 openGL sample that FX!32 is interpreting heavily exercises the graphics display libraries and console display services.

Internet Explorer (IE) is more evenly distributed across the three domains. The moderate amount of OS activity is due to the network and screen display I/O, as well as IE caching the pages it touches to local disk files. The DLL activity is generated by operating system services for screen and file I/O, and network service library routines. The application image coordinates the usage of these routines, and network and display I/O, which is frequently encountered during the operations of selecting and opening web links. This accounts for IE's high percentage of application domain execution.

MS Visual C++ spends nearly all of its time within its application image. This phase of the compiler is responsible for performing the parsing and lexing (front end) of the source code file. There is some use of DLLs, invoking library routines to load included header files. The operating system activity, although small, is present; all I/O must be accessed via a system service.

Microsoft Word's spell-checking service is provided by means of a DLL included with the application. The search through the document and the successive dictionary lookups are both handled by this same DLL. Operating system services are required for accessing portions of the file residing on disk (not in memory pages), displaying the search and compare results to the user, and user-driven I/O for accepting/rejecting word replacement choices (prompted by the spell-checker

tool).

Although there is variability in the instruction mix domain of the desktop application workloads, we notice a more definite pattern followed by the benchmarks. Although there is slight OS activity for go and li (attributable to I/O services), we see that the benchmarks spend practically all of their execution within their application images. There is also no DLL use visible. Clearly these benchmarks do not utilize system services to the level observed in the commercial desktop workloads. With the exception of the CD player, DLLs are more heavily used than the OS. This is especially true in FX!32 and Word, which carry out the tasks captured in the trace via DLL routines.

4.3 Characterization of Image Usage

To investigate the domains present in the trace at the next lower level, we identify the top five most heavily used images (based on number of instructions executed in each image). First, an explanation of some of the more frequently used system executables and DLLs is in order. Table 4 provides a listing of the names of the commonly used images and a brief description of each.

Name	Description
ntoskrnl.exe	Windows NT 4.0 operating system kernel core
hal.dll	Hardware Abstraction Library (HAL); responsible for underlying hardware interface
kernel32.dll	main kernel library
win32k.sys	central system server services provider
gdi32.dll	graphics display interface library
ntdll.dll	library routines provided to each client process on NT
MSVCRT.dll	MS Visual C/C++ runtime library
s3.dll	graphics adapter library for test platform
qv.dll	graphics adapter library for test platform

Table 4: Common System Images

We present image usage of the nine traces. This characterization includes all images (i.e., executables, DLLs, services, drivers, etc.), in Table 5. This data helps to demonstrate several points. First, commercial desktop workloads spend a lot more time in DLLs than benchmarks. From this we can project that the number of procedure calls in desktop applications will likely be higher than in benchmarks. Second, real applications depend not only on system DLLs, but upon their local DLLs as well. We see this explicitly with MS Word.

4.4 Instruction Mix

Although understanding the domain mix and image usage helps us identify differences between benchmarks and desktop applications, we would like to look deeper within each domain to see inherent differences that will affect design decisions. Figure 7 shows the application-only (i.e., instruction mix for only the application and application specific DLLs) instruction mix for each workload. Each entry in the legend represents a class of instructions found within the application

Workload	Image				
	Usage Count				
fourier	bytecpu.exe	winsrv.dll	win32k.sys	ntoskrnl.exe	user32.dll
	34024620	65946	42608	39921	6995
neural	bytecpu.exe	winsrv.dll	ntoskrnl.exe	win32k.sys	ntdll.dll
	34499680	52710	9256	8658	5982
go	go.exe	win32k.sys	ntoskrnl.exe	hal.dll	qv.dll
	36058596	768188	363772	155460	150679
li	li.exe	win32k.sys	ntoskrnl.exe	user32.dll	qv.dll
	32774148	327755	206164	50267	48734
cdplay	ntoskrnl.exe	hal.dll	win32k.sys	tcpip.sys	winsrv.dll
	16524577	2972393	218772	83648	53882
fx32	hal.dll	s3.dll	OPENGL32.DLL	MSVCRT.dll	GLU32.dll
	9477363	5480773	2721733	2604883	598461
ie	iexplore.exe	win32k.sys	ntoskrnl.exe	Fastfat.sys	ntdll.dll
	36992080	19235216	17407503	6037121	6006248
vc50	c1.exe	ntoskrnl.exe	MSVCRT.dll	Ntfs.sys	win32k.sys
	25397991	3206047	842583	367466	330367
word	MSSP232.DLL	MSGREN32.DLL	ntoskrnl.exe	win32k.sys	hal.dll
	14563460	13608164	4082975	3078074	1587210

Table 5: Overall Most Frequently Used Images

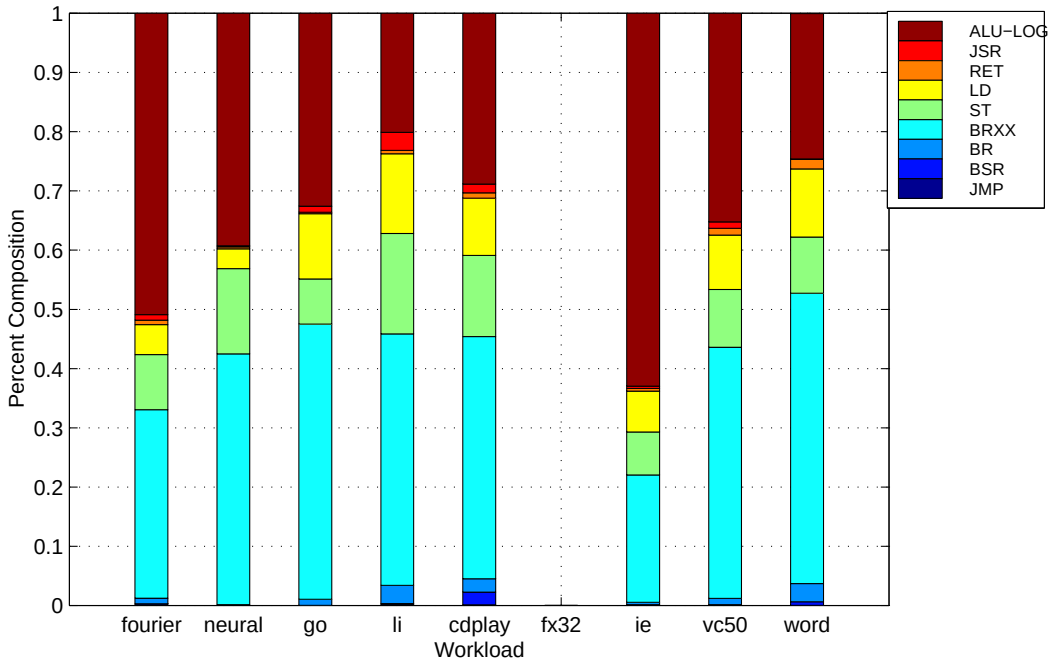

Figure 7: Application-Only Instruction Mix

domain. The y axis denotes the percent composition of the trace and the workloads are displayed on the x axis.

Note that the instruction mix for FX!32 is zero. This is due to the lack of execution within the application image itself. Referring back to Table 5 and the domain instruction mix, we see that nearly all of the workload execution is within DLLs (some is within ntoskrnl.exe).

We can see that all of the remaining workloads are composed mostly of loads, stores, conditional branches, and ALU-logic operations. There is no overriding characteristic to differentiate benchmarks and desktop applications. We can also see significant variability in the instruction mix among the different benchmarks and among the different desktop applications.

Looking next to Figure 8, we see the instruction mix of the entire trace. The first and most

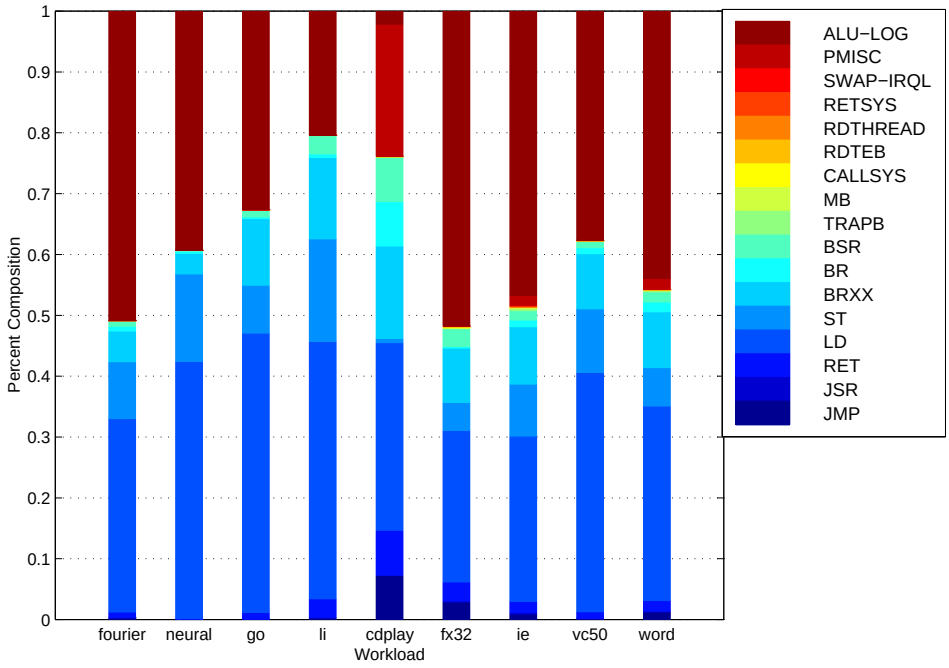

Figure 8: Complete Trace Instruction Mix

noticable difference between the application domain and full trace instruction mix figures is the increase in instruction types present in the trace. Nine instruction classes were present in the application domain instruction mixes, while 17 are present in the full system traces. Worth noting is the presence of six PALcall instruction types in the full traces. Since each PALcall causes a trap to PALcode and takes on the order of tens of cycles to complete, we can conclude that this is a significant insight into the system's inherent runtime latency, not visible with application-only workloads.

The next thing to notice is the striking similarities in instruction mix for the benchmarks bars in Figure 7 and Figure 8. Benchmarks do not interact with the operating system in any significant manner. The desktop application workloads, however, show significant differences between the application domain and the complete trace instruction mixes.

The CD player decreases from approximately 11% store instructions to roughly 1%. BSR instructions *increase* from 1% to about 6%. Most interesting for this application is the decrease

in the number of ALU operations from almost 30% to about 2%, while the number of PALcalls increases from zero to 21%. Referring back to the domain domain execution plots in Figure 6, we can understand why the differences are so great for this workload when including the system activity (over 95% of the workload trace is operating system execution).

Considering the latency incurred by PALcalls, we can see how an optimization which concentrates on improving ALU operations based upon the application domain instruction mixes would have a much small impact on the true system performance. This difference underscores the importance of not only using real workloads for trace-driven simulations, but also that it is essential to include the operating system behavior in order to see the full picture.

The FX!32 complete trace instruction mix is, of course, completely different than the application instruction mix of Figure 7 (no instructions were executed within the FX!32 application image). Both Internet Explorer and Word introduce PALcalls when including the operating system. IE shows an increase in jumps, calls and returns. This most likely reflects the increase in subroutine calls for system services.

Microsoft Word experiences a reduction in load instructions from about 52% to 35%. This can be accounted for by the increase in ALU operations present when OS activity is included.

These results reinforce the points that benchmarks do not represent true desktop workloads and that the desktop workloads display significantly different characteristics when viewed in the presence of system activity.

4.5 Average Basic Block Length

With the inclusion of the operating system activity in our traces, there is an overall increase in the percentage of control flow instructions present. Figure 9 shows a consequence of this fact. In this

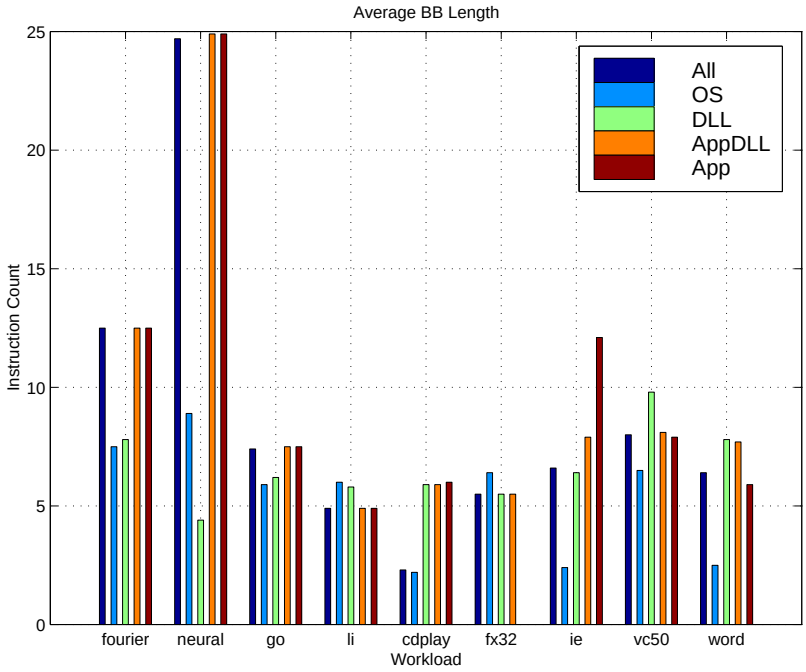

Figure 9: Average Basic Block Length

figure, we present the average basic block length for each workload, on a per domain basis. The *All* bar is the average BB length across all domains, *OS* is for the operating system instructions only, *DLL* is for the workload's DLL instructions only, *AppDLL* is for the combined application and DLL instructions, and *App* is for the application instructions only.

Inspecting the four benchmarks, we notice that there is little difference between the application-only BB length and the overall BB length. Referring to our domain instruction mix figure, we recall that the benchmarks spend about 95% of their execution within their executable images. Therefore as we would expect, including any OS activity into a BB length average would have a minimal effect.

However, considering the amount of OS execution present in the CD player trace, we see how the overall BB length is significantly less than the application-only length. It is almost the same value as that of the OS length value. We see then that not only does including the system activity in the trace influence the overall basic block length, but the *amount* of system activity determines to what degree the length is affected.

In a similar fashion, we observe how FX!32's overall BB length tracks that of its DLLs. This is in line with the amount of time this workload spends in its DLL domain. Internet Explorer has a more evenly distributed amount of execution within the three domains (OS, DLL, Application). This affects the overall BB length, producing a more evenly weighted average of all of its domain BB lengths (no one domain dominates).

VC++ spends a significant amount of time within its own executable image, leading to an overall average BB length similar to the application-only value. MS Word is similar, but in this case it is the DLL behavior which dominates. CD player and Internet Explorer experience a 50% decrease in average basic block length. This can be attributed to an increase in the number of branches in the presence of operating system activity. With this increase in control flow instructions, we expect increased pressure to be placed upon the branch prediction hardware.

As observed in other characteristic categories, the benchmarks do not exhibit noticeable deviations from application-only behavior when the operating system activity is introduced. Again this explains why simulation results using benchmark traces usually track the actual performance when the benchmarks are run on the real system. In contrast, four of the five desktop applications would exhibit significantly different behavior in the presence of the operating system.

5 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 benchmarks do not exercise the kernel and DLL enough to provide meaningful indicators of desktop performance.

Current limitations of the tool include: 1) image bloat due to patching, 2) incorrectly patching data areas due to undocumented image formats, and 3) incomplete reconstruction of the data stream. While issue 1 seems to be unavoidable, issues 2 and 3 are currently under study. 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, Joel Emer, Steve Thierauf, Tom Wenners, Paul Delvy and Dan Lambalot all from Digital Equipment Corporation and Robert Davidson from Microsoft Research. Jason Casmira and David Kaeli have been supported by an NSF CAREER grant.

References

[1] SPEC Newsletter, September 1995.

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

[3] 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.

[4] 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.

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

[6] J.S. Emer and D.W. Clark, "A Characterization of Processor Performance in the VAX 11-780," *Proc. of the 11th Symposium on Computer Architecture*, June 1994, pp. 126-135.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] A. Srivastava and A, Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. of ACM Symposium on Programming Languages, Design and Implementation*, June 1994, pp. 196-205.

[14] 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.

[15] 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.

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

[17] J. Larus and E. Schnarr, "EEL: Rewriting Executable Files to Measure Program Behavior," *Proc. of Programming Languages, Design and Implementation*, June 1995, pp. 291-300.

[18] 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.

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

[20] E. Betts, D. Hunter and S. Smith, "Moving ATOM to Windows NT for Alpha," to appear in this issue.

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

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

[23] H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA, 1993.

[24] *Microsoft Software Developer's Toolkit*, URL: http://www.microsoft.com/msdn/sdk/platform.html.

[25] J. Casmira, *Operating System Rich Workload Characterization,* MS Thesis, Northeastern University, Dept. of Electrical and Computer Engineering, May 1998.

[26] R.J. Hookway and M.A. Herdeg., "DIGITAL FX!32: Combining Emulation and Binary Translation," Digital Technical Journal, Vol. 9, No. 1, 1997, pp. 3-12.