
Tracing and Characterization of Windows NT-based System Workloads

Jason P. Casmira
David P. Hunter
David R. Kaeli

To optimize the design of pipelines, branch predictors, and cache memories, computer architects study the characteristics of benchmark programs by examining traces, i.e., samples of program execution. Since commercial desktop applications are increasingly dependent on services and application programming interfaces provided by the host operating system, the authors argue that traces from benchmark execution must capture operating system execution in addition to native application execution. Common benchmark-based workloads, however, lack operating system execution. This paper discusses the ongoing joint efforts of the Northeastern University Computer Architecture Research Laboratory and Compaq Computer Corporation's Advanced and Emerging Technologies Advanced Development Group to capture operating system-rich traces on Alpha-based machines running the Windows NT operating system. The authors describe the latest PatchWrx software toolset and demonstrate its trace-generating capabilities by characterizing numerous applications. Included is a discussion of the fundamental differences between using traces captured from common benchmark programs and using those captured on commercial desktop applications. The data presented demonstrates that operating system execution can dominate the overall execution time of desktop applications such as Microsoft Word, Microsoft Visual C/C++, and Microsoft Internet Explorer and that the characteristics of the operating system instruction stream can be quite different from those typically found in benchmarking workloads.

The computer architecture research community commonly uses trace-driven simulation in pursuing answers to a variety of design issues. Architects spend a significant amount of time studying the characteristics of benchmark programs by examining traces, i.e., samples taken from program execution. Popular benchmark programs include the SPEC¹ and the BYTEmark² benchmark test suites. Since the underlying assumption is that these programs generate workloads that represent user applications, today's computer designs have been optimized based on the characteristics of these benchmark programs.

Although the authors of popular benchmarks are well intentioned, the resulting workloads lack operating system execution and consequently do not represent some of the most prevalent desktop applications, e.g., Microsoft Word, Microsoft Visual C/C++, and Microsoft Internet Explorer. Such applications make heavy use of application programming interfaces (APIs), which in turn execute many instructions in the operating system. As a result, the overall performance of many desktop applications depends on efficient operating system interaction. Clearly operating system overhead can greatly reduce the benefits of a new computer design feature. Past architectural studies, however, have generally ignored operating system interaction because few tools can generate operating system-rich traces.

This paper discusses the ongoing joint efforts of Northeastern University and Compaq Computer Corporation to capture operating system-rich traces on DIGITAL Alpha-based machines running the Microsoft Windows NT 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 software tools such as the PatchWrx dynamic execution-tracing toolset, which we describe in this paper.

The PatchWrx toolset was originally developed by Sites and Perl at Digital Equipment Corporation's Systems Research Center. They described PatchWrx, as developed for Windows NT version 3.5, in "Studies of

Windows NT Performance Using Dynamic Execution Traces.”³ The Northeastern University Computer Architecture Research Laboratory and Compaq’s Advanced and Emerging Technologies Advanced Development Group continue to develop the toolset. We have updated the framework to operate under Windows NT version 4.0, added the ability to trace programs that have code sections larger than 4 megabytes (MB), added multiple trace buffer sizes, and developed additional postprocessing tools.

After briefly discussing related tracing tools, we describe the PatchWrx toolset and specify the new features we have added. We then analyze PatchWrx traces captured on Windows NT version 4.0, demonstrating the capabilities of the tool while illustrating the importance of capturing operating system-rich traces. In the final section, we summarize the paper, discuss the current limitations of the toolset, and suggest new directions for development and study.

Trace Generation Tools

Trace-driven simulation has been the method of choice for evaluating the merits of various architectural trade-offs.^{4,5} Traces captured from the system under test are recorded and replayed through a model of the proposed design. Computer architecture researchers have proposed methodologies that capture both application and operating system references. These tools include hardware-based⁶⁻¹⁰ and software-based¹¹⁻¹⁵ methods. Some of the issues involved in capturing operating system-rich traces are

1. Tracing overhead (system slowdown)
2. Accuracy (perturbation of the memory address space)
3. Completeness (capturing all desired information, e.g., the operating system reference stream)

Table 1 contains a list of 10 tracing tools that have been developed over the past 10 to 15 years. Although

far from complete, this list provides a sample of the tools that have been used to generate input to a variety of trace-driven simulation studies. We have characterized each tool in terms of the three issues (criteria) previously mentioned. Table 1 lists the target platform(s) for each tracing tool.

Note that many of these tools cannot 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 yet maintains the integrity of the address space. The next section discusses the PatchWrx toolset.

PatchWrx

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

PatchWrx and the Alpha PAL Routines

The PatchWrx software tool is made possible through the PAL used by DIGITAL 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. We modified and extended the shrink-wrapped Alpha PALcode on a DIGITAL Alpha 21064-based system to support the PatchWrx operations. The mod-

Table 1
Sample of Tracing Tools

Name	Average Slowdown	Address Perturbation	Operating System Activity	Platform
ATOM ¹³	10X to 100X	No	Yes	DIGITAL Alpha UNIX
ATUM ¹⁶	20X	No	Yes	DIGITAL VAX OpenVMS
EEL ¹⁷	10X to 100X	Yes	No	SPARC Solaris
Etch ¹⁸	35X	Yes	No	Intel x86 Microsoft Windows NT V4.0
NT-Atom ¹⁹	10X to 100X	No	No	DIGITAL Alpha Microsoft Windows NT V4.0
PatchWrx ³	4X	No	Yes	DIGITAL Alpha Microsoft Windows NT V4.0
Pixie ²⁰	10X to 100X	Yes	No	DIGITAL MIPS ULTRIX
QPT ¹²	10X to 100X	Yes	No	SPARC Solaris, DIGITAL ULTRIX
Shade ²¹	6X	No	No	SPARC Solaris
SimOS ¹⁴	10X to 50,000X	No	Yes	DIGITAL Alpha UNIX, SGI IRIX, SPARC Solaris

ified PatchWrx PAL routines serve two major purposes: (1) to reserve the trace buffer at system boot time and (2) to log 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 can be varied depending on the amount of physical memory installed on the system. Since we use PAL routines to reserve this memory, the operating system is not aware that the memory exists because the PAL code performs all low-level system initialization before the operating system is started.

PatchWrx logs all trace entries in this buffer. Writing trace entries directly to physical memory has several advantages. First, writing to memory is much faster than writing to disk or to 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 regardless of which address space is currently running.

To enable PatchWrx to operate under Windows NT versions 3.51 and 4.0, we started with the PAL routines modified by Sites and Perl³ and made additional modifications as required by the operating system versions. These modifications were concentrated in the process data structures. The PatchWrx-specific PAL routines are listed in Table 2. The first three routines are used for reading the trace entries from the buffer and for turning tracing on and off. The remaining five routines are used to log trace entries based on the type of instruction instrumented.

PatchWrx Image Instrumentation

Next we describe how we use PatchWrx to instrument Microsoft Windows NT images. Patching the operating system involves the instrumentation of all the binary images, including applications, operating system executables, libraries, and kernel. Once patching is complete, trace entries are logged by means of PAL routines as images execute.

We define a patched instruction as an instruction within an image's code section that is overwritten with an unconditional branch (BR) to a patch. The target of the BR contains the *patch section*. The patch section includes the trap (CALL_PAL) to the appropriate PAL routine that 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; instead, it generates new images that contain patches. This operation preserves the original images on the system in case they need to be restored. Instrumentation involves replacing all branching instructions of type unconditional branch, conditional branch (e.g., branch if equal to zero [BEQ]), branch to subroutine (BSR), function return (RET), jump (JMP), and jump to subroutine (JSR) within an image's code section with unconditional branches to a patch section. If loads and stores are also traced, PatchWrx replaces these instructions (e.g., load sign-extended longword [LDL]) with unconditional branches to the patch section, where the original load or store instruction is copied. A return branch is also needed to return control flow to the instruction subsequent to the original load. When PatchWrx encounters this patch, the tool records the register value of the original load or store instruction in the trace log. The patch section contains all the patches for the image and is added to the rewritten image. Figure 1 shows examples of patched instructions. PatchWrx replaces only branch instructions within an image to reduce the type and number of entries logged in the trace buffer. Using these traced branches, the tool can later reconstruct the basic blocks they represent.

As shown in Figure 1, PatchWrx replaces BR and JMP instructions with BR instructions that transfer control to the patch section. The original BR or JMP instruction is repeated in the patch section for the purpose 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 for reconstructing the traced instruction stream. PatchWrx

Table 2
PatchWrx-specific PAL Routines

PAL Routines	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

	ORIGINAL CODE	PATCHED CODE
EXAMPLE 1	<pre> : : JMP ZERO, (R19) : : </pre>	<pre> : : JMP ZERO, (R19) BR PATCH.001 : : ----- PATCH.001: CALL_PAL PWJSR JMP ZERO, (R19) </pre>
EXAMPLE 2	<pre> : : JSR R26, (R19) : : </pre>	<pre> : : JSR R26, (R19) BSR R26, PATCH.002 : : ----- PATCH.002: CALL_PAL PWJSR JMP ZERO, (R19) </pre>
EXAMPLE 3	<pre> : : BEQ R3, TARGET.003 : : </pre>	<pre> : : BEQ R3, TARGET.002 BR PATCH.003 BACK.003 : : ----- PATCH.003: BEQ R2, PATCH.003T CALL_PAL PWBRF BR BACK.003 PATCH.003T: CALL_PAL PWBRT BR TARGET.003 </pre>
EXAMPLE 4	<pre> : : LDL R20, 4(R16) : : </pre>	<pre> : : LDL R20, 4(R16) BR PATCH.004 BACK.004 : : ----- PATCH.004: CALL_PAL PWLDST LDL R20, 4(R16) BR BACK.004 </pre>

Figure 1
Instruction Patch Examples

replaces JSR and BSR instructions with BSR patches. This replacement preserves the return address (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.

Conditional branches have a larger and more complex patch than the other branch types 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 or fall-through trace entry. The return branch in the patch section is a replica of the original conditional branch.

As explained earlier, for all patches, PatchWrx replaces the original branch with a patch unconditional branch. Since Alpha instructions are equal in size, this replacement process allows patching without increasing the code size within the image. Although the code size remains unchanged, the image size will increase in proportion to the number of patches added. This

image size change becomes an issue for dynamically linked library (DLL) images.

Patching Dynamic Link Libraries

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

DLLs are loaded into memory and shared in this manner. When patches are added to a DLL, the size of the image increases. When this image is mapped to

physical memory (as per its preferred base load address), the larger image may overlap with another image having a base address within the new range. This image overlap can prevent the operating system from booting properly: some environment DLLs will conflict in memory because they 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, which modifies the base load addresses of each patched DLL to eliminate conflicts. Rebasing affects the address accuracy of the patched system, though we are able to readjust the addresses during reconstruction. An increase in the paging activity may also be observed since the additional code may cross page boundaries.

The original version of the PatchWrx toolset was developed on Microsoft Windows NT version 3.5. When versions 3.51 and 4.0 were released, several modifications were made to the image format. In completing the 3.51- and 4.0-compatible versions of PatchWrx, we had to address this issue. One change that affected 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 look up the addresses of DLL procedures used (i.e., imported) by the executable binary. In developing the current generation of PatchWrx, we had to make modifications to use image header fields that had previously remained unused or reserved, indicating the executable code sections that contained data areas.

Another issue that we addressed in the recent modifications to PatchWrx was long branches. The original version of PatchWrx replaces all branch, jump, call, and return instructions with either BR or BSR instructions 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., JMP or JSR instructions) to provide this branch-to-patch transition. Register and register-indirect branching instructions would require perturbing the machine state. Therefore, the developers could use only program counter (PC)-based offset branching instructions.

As discussed previously, in replacing a control flow instruction with a patch branch, PatchWrx uses a BR or BSR instruction in which the offset field is set to branch to the corresponding patch within the image's patch section. The Alpha architecture branching instructions use the format shown in Figure 2.



Figure 2
Alpha Branch Instruction Format

The branch target virtual address computation for this format is $\text{newPC} = (\text{oldPC} + 4) + (4 * \text{sign-extended}(21\text{-bit branch displacement}))$. The register field holds the return address for BSRs. With this branch format and target virtual address computation, the Alpha architecture provides a branch target range of 4 MB from an instruction's current PC.

Several applications that run today on Microsoft Windows NT version 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 4-MB 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, we developed two new branch instructions for use with PatchWrx. These new branches were not implemented in the instruction set architecture of the Alpha architecture. Instead, we used PALcode to implement them. The two new branches are designated long branch (LBR) and long branch subroutine (LBSR). Figure 3 illustrates the format of these two instructions.

The computation of the target virtual address is $\text{newPC} = (\text{oldPC} + 4) + (4 * \text{sign-extended}(25\text{-bit branch displacement}))$ for LBR branches and $\text{newPC} = (\text{oldPC} + 4) + (32 * \text{zero-extended}(20\text{-bit branch displacement}))$ for LBSR branches. PatchWrx uses LBRs when patching any control flow instruction that has a displacement greater than 4 MB. PatchWrx uses LBSRs similarly for control flow instructions that must preserve the register field value.

When an LBR or LBSR instruction is executed within the image code section, a trap to PALcode occurs. Normally, CALL_PAL instructions have one of several defined function fields that cause a corresponding PAL routine to be executed. The two long branch instructions have function fields that do not belong to any of the defined CALL_PAL instructions 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.

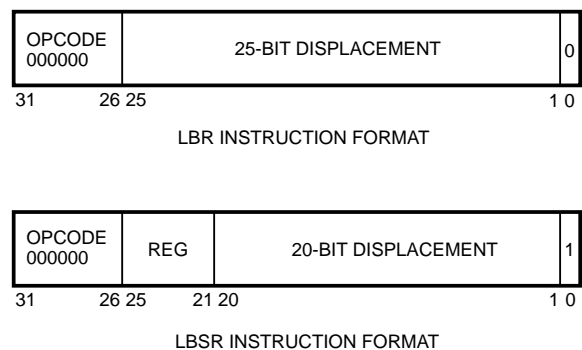


Figure 3
PALcode Long Branch Instruction Formats

As shown in Figure 3, both long branch types have the same PALcode operation code (opcode) value of 000000. To distinguish between the two types, the least significant bit in the instruction word is set to 0 for LBRs and to 1 for LBSRs. This bit is not included as a usable bit for the displacement fields of either branch type. Consequently, each LBR has a 25-bit displacement field and each LBSR has a 20-bit field. With a 25-bit usable displacement field, the PALcode performs the LBR target address computation, allowing a ± 64 -MB range.

Since each LBSR instruction has a 20-bit displacement field, whereas the original Alpha architecture branch displacement field is 21 bits, the target instruction address computation for LBSR instructions is performed differently than for standard branches within the PALcode. As shown in the address computation equation, the 20-bit displacement is multiplied by 32 rather than by 4 (as for the LBR branch). Notice that the 20-bit displacement is always zero extended. The computation provides the LBSR instruction with a displacement of +32 MB.

This computation procedure has two implications. First, LBSR instructions can only be used to branch from an image code section to an image's patch section. Second, branches into the patch section are either BR or BSR instructions (or their long displacement counterparts). PatchWrx uses only BR or LBR instructions to return from the patch section to the original branch target within a code section; BSR and LBSR instructions are never used. Therefore, restricting LBSR instructions to use positive displacements does not present a problem.

The LBSR displacement multiplier value of 32 does present some restrictions, however. The multiplier value of 4 used in the original Alpha instruction set architecture represents the instruction word length of 4 bytes. Thus, 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., eight-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, PatchWrx can insert no operation (NOP) instruction words and advance the next available patch target address until the necessary alignment is achieved. PatchWrx never executes the NOPs; they are inserted for alignment purposes only. Although inserting these NOP instructions increases the image size, we have implemented several optimizations into the instrumentation algorithm to minimize this increase. For example, a queue is used to hold LBSRs that do not align. As LBR patches are committed, PatchWrx probes the queue to determine if any LBSRs align from their origin to the newly available patch target offset.

Trace Capture

The PatchWrx toolset allows the user to turn tracing on and off and thus capture any portion of workload execution. The tracing tool is also responsible for copying trace entries from the physical memory buffer to disk. Copying the trace buffer to disk is performed after tracing has stopped so that the time required to perform the copy does not introduce any overhead during trace capture.

PatchWrx logs a trace entry for each patch encountered during program execution. As it executes instructions within the code section, PatchWrx encounters an unconditional PatchWrx branch. Instead of branching to the original target, the patched branch transfers control to the image's patch section. Within the patch section, a PatchWrx PALcall traps to the PAL routine corresponding to the patch type and logs a trace entry to the trace buffer. The PAL routine then returns to the instruction following the CALL_PAL instruction. PatchWrx uses an unconditional branch to transfer control from the patch section back to the original target within an image code section. During the execution of the PatchWrx PAL routine, necessary machine state information is recorded and logged in the trace buffer. This allows for the capture of register contents, process ID information, etc., which are used later 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, the tracing tool must also capture a snapshot of the base load addresses of all active images on the system. This snapshot serves as the virtual address map used in reconstructing the trace. Each active process and its associated libraries is 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, the virtual address map would not be necessary to perform reconstruction. Instead, PatchWrx could map target addresses from the trace buffer using the base address values contained in the static image headers.

The type of trace record that PatchWrx logs into the trace buffer depends on the type of branch or low-level PAL function being traced. Figure 4 shows the trace record formats. The first three trace entry formats consist of an 8-bit opcode and a 24-bit time stamp. The time stamp is the low-order 24 bits of the CPU cycle counter. The 32-bit field of these three formats depends on 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.

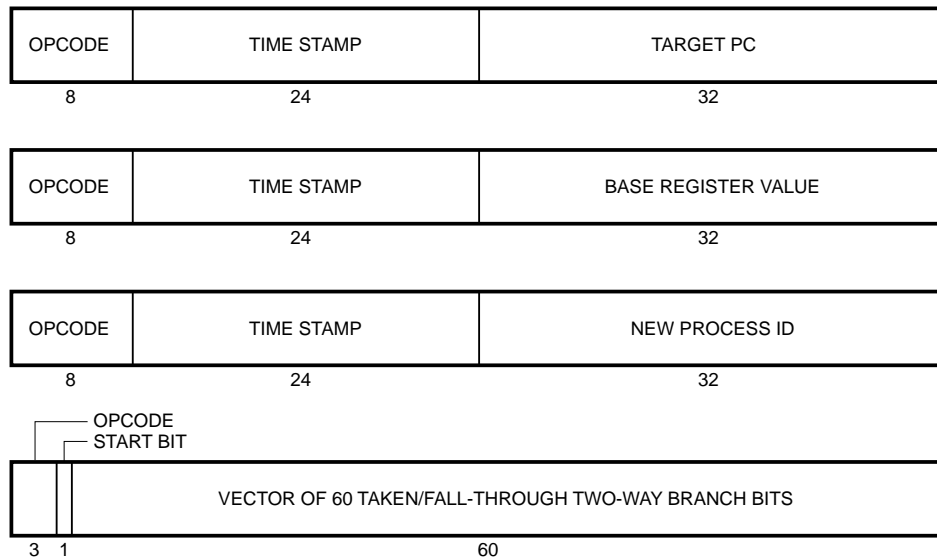


Figure 4
Trace Entry Formats

The fourth trace entry type is used for tracing conditional branches. It uses a 3-bit opcode and up to 60 taken/fall-through bits. A start bit is used to determine how many bits are active. The start bit is set to 1 if a conditional branch is taken and to 0 if the branch is not taken. This recording scheme allows a compact encoding of conditional branch trace entries. During trace reconstruction, PatchWrx uses conditional branch trace entries to reconstruct the correct instruction flow when conditional branches are encountered and to provide concise information about when to deliver interrupts in loops.

Trace Reconstruction

The reconstruction phase is the final step in generating a full instruction stream of traced system activity. As shown in Figure 5, trace reconstruction requires several resources in order to generate an accurate instruction stream of all traced system activity.

Trace reconstruction reads and initializes the heading of the captured trace, which includes a time stamp, the name of the user who captured the trace, and any important system configuration information, e.g., the operating system version number. Next, reconstruction reads the first four raw trace records, which are automatically entered whenever tracing is turned on. These records contain the first target virtual address, the active process ID, the value of the stack pointer, and the first taken/fall-through record to be used (such records always precede the branches they represent). PatchWrx uses this information to initialize the necessary data structures of the reconstruction process.

Using the first target virtual address and process ID pair from the captured trace, trace reconstruction consults the virtual address map to determine in which image the instruction falls (based on its dynamic base load address) and where that image is physically located on the system. The tool consults the patched image to determine the actual instruction at the target address, records this instruction, and then reads the next instruction from the patched image. This process continues until reconstruction encounters either a conditional branch or an unconditional branch. A conditional branch causes the tool to check the first active bit of the current taken/fall-through entry to determine subsequent control flow; the process then continues at that address. If an unconditional branch is encountered, reconstruction records the entry and checks it against the next captured trace entry. If the two entries match, the tool outputs the recorded instructions to an instruction stream file, consults the captured trace entry for the next target instruction virtual address, and repeats the procedure until the entire captured trace has been processed.

Since PatchWrx captures interrupts and other low-level system activities (e.g., page faults) in the trace, these activities must also be reconstructed. When PatchWrx logs an interrupt 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. PatchWrx flushes the currently active taken/fall-through entry to the memory buffer and initializes a new taken/fall-through entry. This new entry will be responsible for

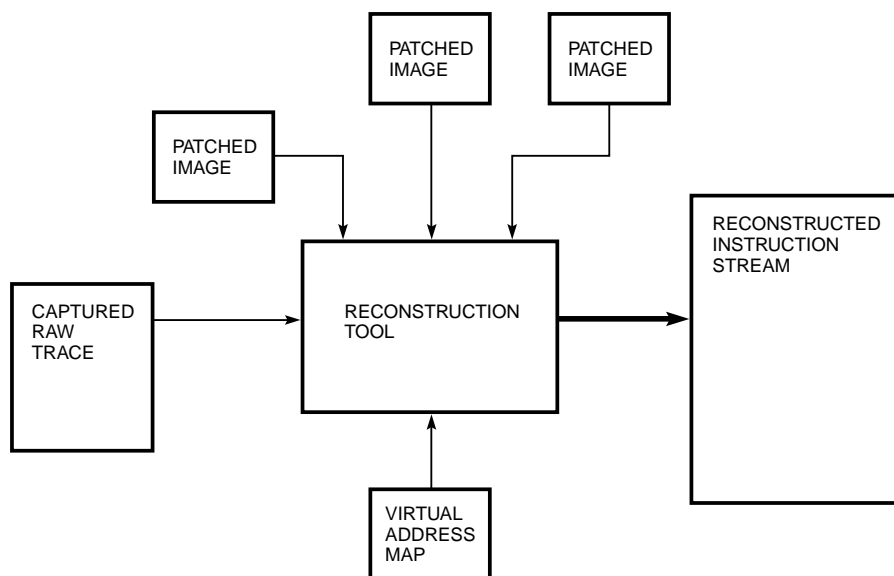


Figure 5
Instruction Stream Reconstruction Resources

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, the reconstruction tool looks for the interrupt's first unexecuted instruction address to know which instruction to stop at when reconstructing the instruction stream. The tool then begins reconstructing the instruction stream, including the interrupt handler stream. If the unexecuted instruction is within a loop, trace reconstruction utilizes the taken/fall-through entry convention. On taking the interrupt, the active taken/fall-through record is flushed and another record is started. This process allows the tool to continue to reconstruct iterations of the loop until all the taken/fall-through bits are exhausted.

Operating System–Rich Workload Characterization

As presented in the study by Lee et al.,¹⁸ desktop applications and benchmarks share some workload characteristics, but applications alone do not represent full system behavior. To investigate and address system design issues, computer architects should use operating system–rich traces.

To illustrate this point, we present a sample of the various workload characteristics that exist in a set of benchmark and desktop applications specially selected to study the differences in the use of the operating system and related services. The first characteristic we discuss is the amount of time each benchmark or desktop application spends within three domains:

1. Application-only domain (e.g., winword.exe and excel.exe)

2. DLL domain—Win32 user (e.g., kernel32.dll, user32.dll, and ntdll.dll)

3. Operating system domain—Win32 kernel, kernel, system processes, system idle process (e.g., Win32K.sys, ntoskrnl.exe, drivers, and the spooler)

Examining these times provides insight into a workload's use of each domain. We also examine DLL and system service usage on an image basis for each workload. This breakdown helps us more clearly identify the dependence between the workload and the system services provided by the Windows NT operating system.

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 further highlights the behavior lacking in application-only traces, such as increases in branch and memory instructions, when compared to application-only workloads. We present the average basic block lengths for each domain of execution (application-only, DLL, operating system) separately and then in combination. This metric reveals which workload domain dominates the branching behavior. Casmira's work provides a more complete description of these differences across a wider set of workload characteristics.²⁵

Workload Descriptions

We performed all the experiments reported on in this paper on a DIGITAL Alpha platform running the Microsoft Windows NT version 4.0 operating system. We captured the traces on a 150-megahertz Alpha 21064 processor. The system configuration included 80 MB of physical memory. Table 3 lists the workloads we examined.

Table 3
Workload Description

Workload	Description
fourier	BYTEmark benchmark; a numerical analysis routine for calculating series approximations of waveforms
neural	BYTEmark benchmark; a small, functional back-propagation network simulator
go	SPEC95 <i>Go!</i> game benchmark
li	SPEC95 Lisp interpreter benchmark
cdplay	Microsoft CD Player playing a music CD
fx!32	DIGITAL 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/C++ V5.0 compiling a 3,000-line C program
word	Microsoft Word97 V7.0, spell-checking a 15-page document

The fourier and neural workloads are from the BYTEmark benchmark test suite: the neural workload is a small array-based floating-point test; the fourier workload is designed to measure transcendental and trigonometric floating-point unit performance.

The go and li workloads are from the SPEC95 integer benchmark suite: the go workload is a simulation of the game *Go!*, with the computer playing against itself; the li workload is a Lisp interpreter. All the workloads use the standard inputs provided with the benchmarks and are compiled with the default optimization level using the native Alpha version of Microsoft C/C++ version 5.0.

The cdplay workload is the Microsoft CD Player application included in Microsoft Windows NT version 4.0. The device was traced while playing a music CD using default playing options (e.g., playing all the songs in order).

The fx!32 workload is the DIGITAL FX!32 version 1.1 emulator/translator provided by Compaq's DIGITAL Alpha Migration Tools Group.²⁶ We ran the robot arm OpenGL sample Intel-based application in the foreground during trace capture.

The ie workload is the standard Microsoft Internet Explorer version 2.0 workload included in Microsoft Windows NT version 4.0. The ie workload was traced while traversing four links through the Sony home web page, arriving finally at the Sony PlayStation Store web page. The trace was captured on May 4, 1998; pages may have changed since this date. The history cache and the web link cache were both empty when the trace was captured.

The vc50 workload is the Microsoft C/C++ version 5.0 compiler compiling a 3,000-line C source code file. We used the command line interface, and we used the default optimization levels and other parameters, which best represented the common usage of the compiler.

The word workload is Microsoft Word from the Microsoft Office97 desktop application suite for the Alpha processor used to capture a manual spell check of a 15-page Microsoft Word document. The standard Microsoft Word dictionary was employed.

To provide a clear and representative comparison of workload behavior, we captured several traces. For all scenarios, full traces of each workload captured approximately 5 to 10 seconds of execution, filling the 45-MB 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 ensure that the traces captured were representative of the overall workload behavior, we captured multiple traces. We chose different points during execution for tracing to allow comparison between different portions of the selected scenarios. To investigate the variability present in selected workloads, we traced additional scenarios. A second Microsoft Word trace was captured with the application performing an auto-format operation of the same document used in the first trace of the spell-check operation, and we captured a second Microsoft Internet Explorer trace, repeating the Sony links but with the links cached. We captured a second trace 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. To condense the number of memory pages occupied by an image, Microsoft designed the new linker to allow data to reside within the code regions. Hookway and Herdeg²⁶ provide an explanation of the DIGITAL FX!32 emulation and translation/optimization procedures. Casmira discusses these scenarios and others.²⁵

Domain Mix

To illustrate the inherent differences between benchmark and desktop application behavior, we break down the captured trace in terms of three mutually exclusive domains. These domains are (1) application, (2) DLL, 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 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. The x-axis lists the workloads, and the y-axis presents the percent composition of the captured trace. Note that the four benchmarks, i.e., fourier, neural, go, li, spend at least 95 percent of their execution within their application image. Both the fourier and the neural benchmarks spend about 99 percent of their execution within their application image. The go and li benchmarks do exhibit some operating system activity, but this activity 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 cdplay workload. The Microsoft CD Player application is I/O bound, relying heavily on the necessary services provided by the operating system and the 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).

The fx!32 workload spends nearly all its execution time operating within DLLs. The robot arm Intel x86 OpenGL sample that the DIGITAL FX!32 application is interpreting heavily exercises the graphics display libraries and console display services.

The ie workload is more evenly distributed across the three domains. The moderate amount of operating system activity is due to the network and screen display I/O and also to the Microsoft Internet Explorer's caching of the pages it touches to local disk files. The DLL activity is generated by operating system services for screen and file I/O and by 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 coordination accounts for the high percentage of application domain execution exhibited by ie, as shown in Figure 6.

The vc50 workload spends nearly all its execution time within its application image. This phase of the compiler is responsible for performing the parsing and lexical analysis of the source code file. There is some use of DLLs through invoking library routines to load included header files. The operating system activity,

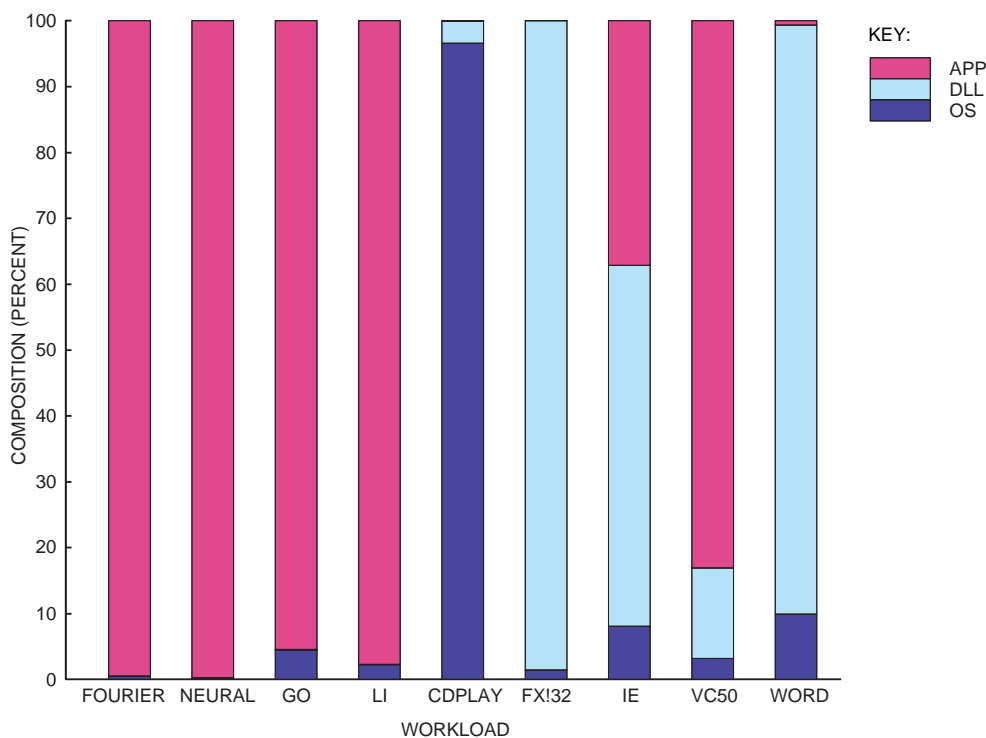


Figure 6
Domain Execution Mix

although small, is present; all I/O must be accessed by means of a system service.

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

Figure 6 shows the consistent pattern of instruction domains that the four benchmarks follow in contrast to the variability in the instruction mix domain of the desktop application workloads. Even though there is slight operating system activity for *go* and *li* (attributable to I/O services), the benchmarks spend practically all their execution within their application images; no DLL use is 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, the commercial desktop applications examined use DLLs more heavily than they do operating system services. This is especially true in the *fx!32* and *word* workloads, which carry out the tasks captured in the trace by means of DLL routines.

Characterization of Image Usage

To investigate the domains present in the trace at the image level, we identified the top five most heavily used images, based on the 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 lists the names of the commonly used images and a brief description of each.

We present the image usage of the nine traces. This characterization includes all the images (e.g., executables, DLLs, services, and drivers) listed in Table 5. The data helps demonstrate several points. First, commercial desktop workloads spend a lot more time in DLLs than benchmarks do. Consequently, we can project that the

number of procedure calls in desktop applications will be higher than the number of calls in benchmarks. Second, real applications depend not only on system DLLs but also on their local DLLs. We see this behavior explicitly with the Microsoft Word application.

Instruction Mix

Although understanding the domain mix and image usage helps identify differences between benchmarks and desktop applications, we would like to look deeper within each domain to see inherent differences that affect design decisions. Figure 7 shows the application-only instruction mix (i.e., the instruction mix for only the application and application-specific DLLs) for each workload. Each entry in the legend represents a class of instructions found within the application domain. The y-axis denotes the percent composition of the trace; the workloads are displayed on the x-axis.

Note that the instruction mix for the *fx!32* workload is zero. This value is a result of the lack of execution within the application image itself. Referring back to Table 5 and the domain instruction mix, note that nearly all the workload execution is within DLLs (some execution is within *ntoskrnl.exe*). The remaining workloads consist mainly of load, store, conditional branch, and arithmetic and logic unit (ALU) logic operations. No overriding characteristic differentiates benchmarks and desktop applications. Note the significant variability in the instruction mix among the different benchmarks and among the different desktop applications.

Figure 8 shows the instruction mix of the entire trace. The first and most noticeable 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 6 *CALL_PAL* instruction types (all use the same opcode, but invoke 6 different PAL routines) in the full traces. Since each executed *CALL_PAL* instruction causes a trap that takes on the order of tens of cycles to complete, we can conclude that this is a

Table 4
Common System Images

Name	Description
<i>ntoskrnl.exe</i>	Windows NT operating system kernel core
<i>hal.dll</i>	Hardware Abstraction Library (HAL), which is responsible for the underlying hardware interface
<i>kernel32.dll</i>	Main kernel library
<i>win32k.sys</i>	Kernel-mode device driver
<i>gdi32.dll</i>	Graphics display interface library
<i>ntdll.dll</i>	Library routines provided to each client process on the Windows NT system
<i>MSVCRT.dll</i>	Microsoft C/C++ run-time library
<i>s3.dll</i>	Graphics adapter library for the test platform
<i>qv.dll</i>	Graphics adapter library for the test platform

Table 5
The Five Most Frequently Used Images in Each Application or Benchmark

Workload	Image Name (Percentage of Total Number of Instructions Executed within the Image)					
	fourier	bytecpu.exe (99.5%)	winsrv.dll (0.2%)	win32k.sys (0.1%)	ntoskrnl.exe (0.1%)	user32.dll (0.02%)
neural	bytecpu.exe (99.7%)	winsrv.dll (0.2%)	ntoskrnl.exe (0.03%)	win32k.sys (0.03%)	ntdll.dll (0.02%)	Other (0.02%)
go	go.exe (95.5%)	win32k.sys (2.0%)	ntoskrnl.exe (1.0%)	hal.dll (0.4%)	qv.dll (0.1%)	Other (1.0%)
li	li.exe (97.7%)	win32k.sys (1.0%)	ntoskrnl.exe (0.6%)	user32.dll (0.1%)	qv.dll (0.1%)	Other (0.5%)
cdplay	ntoskrnl.exe (81.8%)	hal.dll (14.7%)	win32k.sys (1.1%)	tcpip.sys (0.4%)	winsrv.dll (0.3%)	Other (1.7%)
fx!32	hal.dll (42.5%)	s3.dll (24.6%)	OPENGL32.DLL (12.2%)	MSVCRT.dll (11.7%)	GLU32.dll (2.7%)	Other (6.3%)
ie	iexplore.exe (37.2%)	win32k.sys (19.3%)	ntoskrnl.exe (17.5%)	Fastfat.sys (6.1%)	ntdll.dll (6.0%)	Other (13.9%)
vc50	c1.exe (83.1%)	ntoskrnl.exe (10.5%)	MSVCRT.dll (2.8%)	Ntfs.sys (1.2%)	win32k.sys (1.1%)	Other (1.3%)
word	MSSP232.DLL (36.4%)	MSGREN32.DLL (34.0%)	ntoskrnl.exe (10.2%)	win32k.sys (7.7%)	hal.dll (4.0%)	Other (7.7%)

significant insight into the system's inherent run-time latency, not visible with application-only workloads.

Next note the striking similarities in instruction mix for the four benchmarks in Figures 7 and 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 number of store instructions for the cdplay workload *decreases* from about 11 percent to approximately 1 percent. The number of BSR instructions *increases* from 1 percent to about 6 percent. Most interesting for this application is the decrease in the number of ALU operations from almost 30 percent to about 2 percent, while the number of CALL_PAL instructions increases from 0 to 21 percent. Referring to Figure 6, the domain execution mix plots clearly show why the differences for this workload are so large when the system activity is included—more than 95 percent of the workload trace is operating system execution.

Considering the latency incurred by executing CALL_PAL instructions, clearly an optimization that concentrates on improving ALU operations based on the application domain instruction mixes would have a much smaller impact on the true system performance. The measured difference in instruction mix underscores the importance not only of using real workloads for trace-driven simulations but also of including the operating system behavior in order to see the full picture.

The fx!32 complete trace instruction mix is, of course, completely different from the application instruction mix of Figure 7, in which no instructions

were executed within the fx!32 application image. Both the ie and the word workloads introduce CALL_PAL instructions when including the operating system. The ie instruction mix shows an increase in jumps, calls, and returns, which most likely reflects the increase in sub-routine calls for system services. The word instruction mix experiences a reduction in load instructions from approximately 52 percent to 35 percent. This decrease can be attributed to the increase in ALU operations present when operating system activity is included.

The results presented in Figures 7 and 8 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.

Average Basic Block Length

Including the operating system activity in our traces yields an overall increase in the percentage of control flow instructions present. Figure 9 shows a consequence of this fact. In this figure, we present the average basic block length for each workload, on a per-domain basis. The ALL bar is the average basic block length across all domains; OS denotes the operating system instructions only; DLL denotes the workload's DLL instructions only; APPDLL denotes the combined application and DLL instructions; and APP denotes the application instructions only.

Inspecting the four benchmarks, we notice little difference between the application-only basic block length and the overall basic block length. Referring to our domain instruction mix figure, recall that the benchmarks spend about 95 percent of their execution

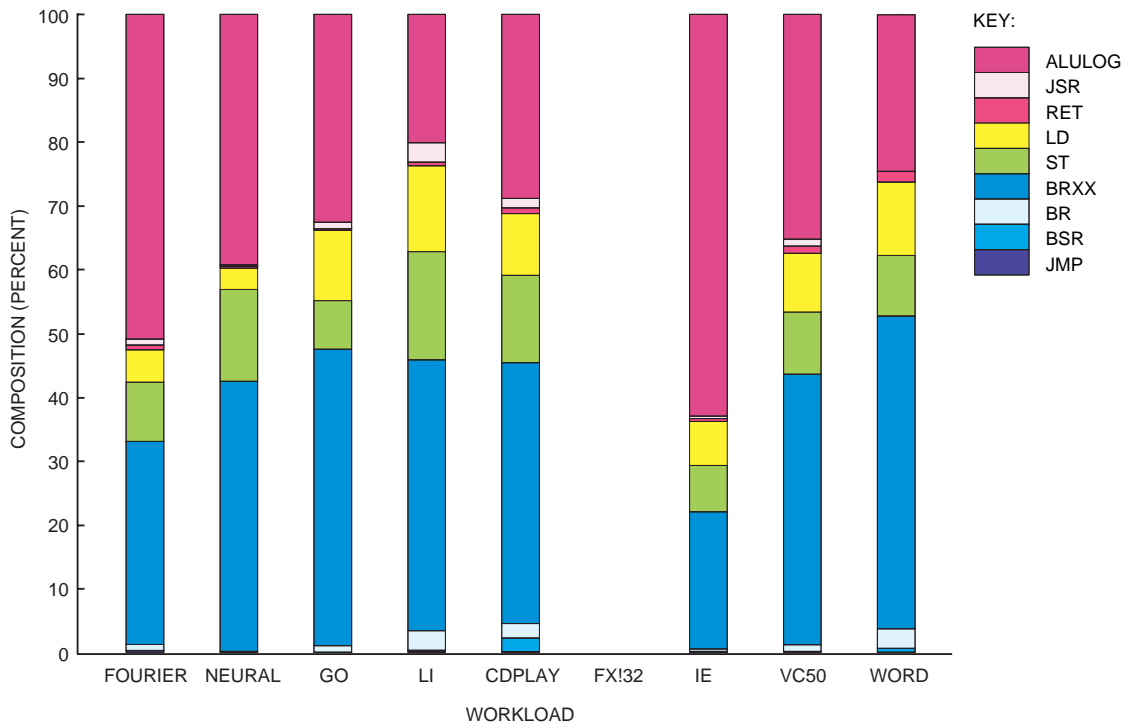


Figure 7
Application-only Instruction Mix

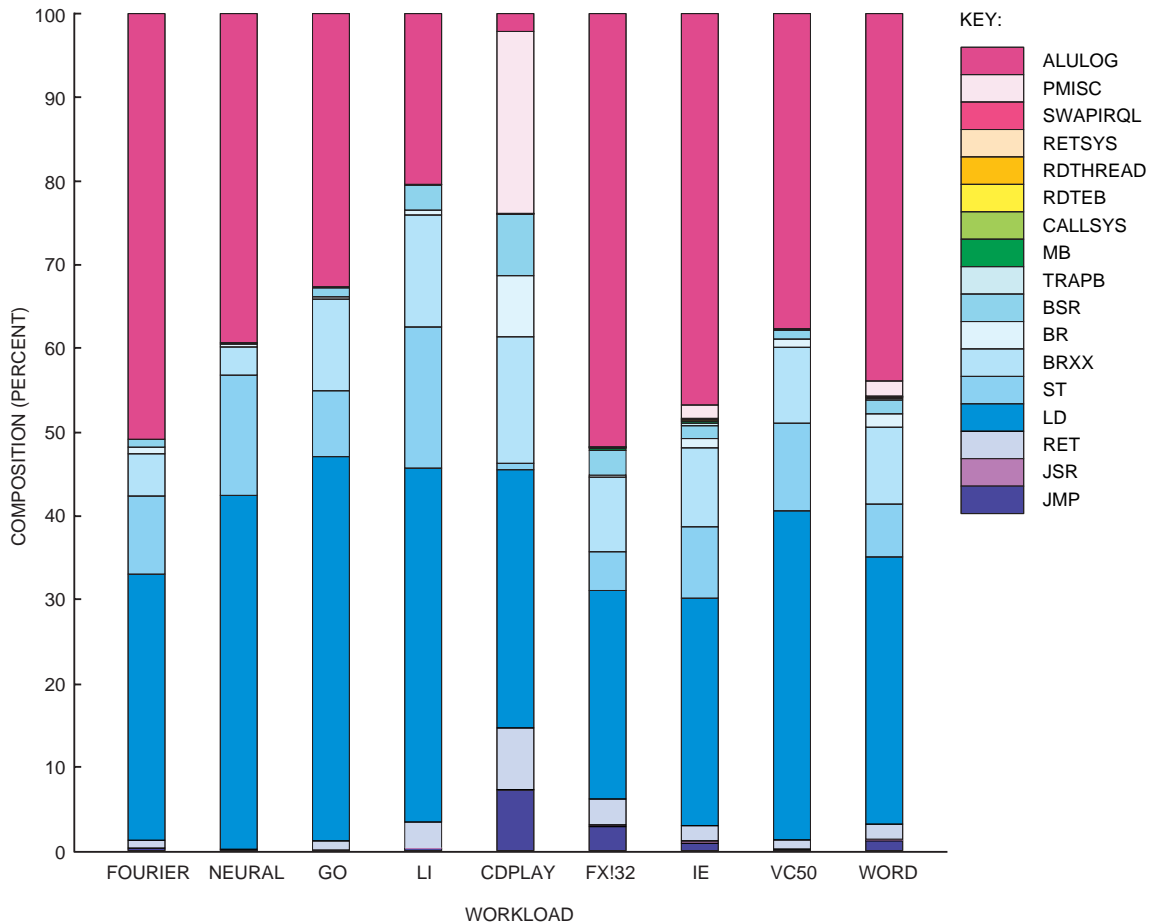


Figure 8
Complete Trace Instruction Mix

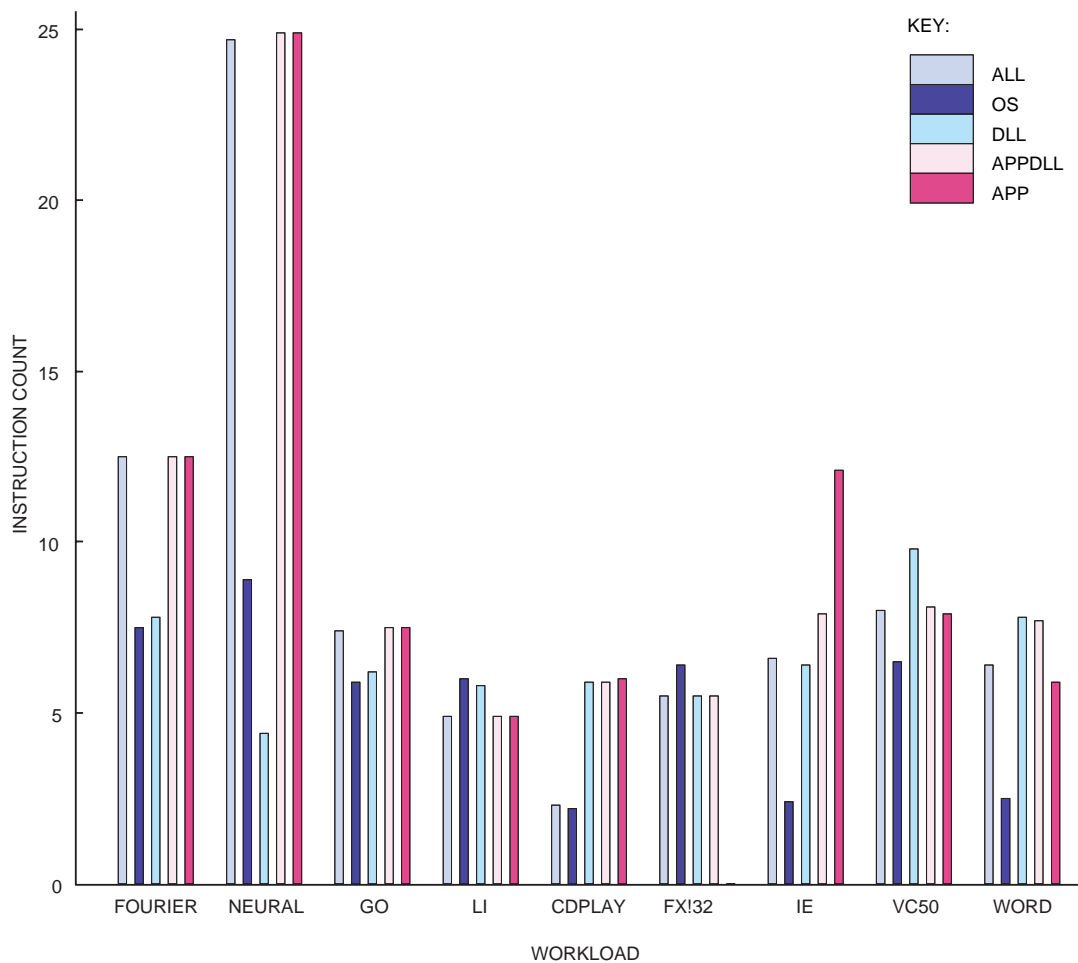


Figure 9
Average Basic Block Length

within their executable images. Therefore, including any operating system activity into a basic block length average has a minimal effect.

However, considering the large amount of operating system execution present in the cdplay trace, the overall basic block length is significantly less than the application-only length. The overall and operating system length values are almost the same. 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, the overall basic block length of the fx!32 trace tracks that of its DLLs. The length is directly proportional to the amount of time the workload spends in its DLL domain. The execution of the ie workload is more evenly distributed among the three domains, which affects the overall basic block length, producing a more evenly weighted average of all its domain basic block lengths (no one domain dominates).

The vc50 workload spends a significant amount of time within its own executable image, which leads to an overall average basic block length similar to the application-only value. The word workload is similar, but the DLL behavior dominates. The cdplay and ie workloads experience a 50 percent decrease in average basic block length. This decrease 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 four 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 exhibit significantly different behavior in the presence of the operating system.

Summary

In this paper we described the PatchWrx toolset. We 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 the DLLs. In addition, we showed that existing desktop benchmarks do not exercise the kernel and the DLL sufficiently to provide meaningful indicators of desktop performance.

These results have reinforced our argument that researchers need to use traces with both application and operating system information, especially as new applications spend more time executing within the operating system. The goal is for computer architects to use operating system–rich traces of applications that dominate the desktop market.

We have recently finished modifications to the PAL to enable PatchWrx to run on the Alpha 21164 platform. We plan to study a wider range of desktop applications, including database and server applications. Future plans also include migrating the toolset to the Windows 2000 operating system.

Acknowledgments

We would like to acknowledge the help and advice of the following people: Richard Sites of Adobe Systems; Sharon Smith, Geoff Lowney, Joel Emer, Steve Thierauf, Tom Wenners, Paul Delvy, and Dan Lambalot, all from Compaq Computer Corporation; and Robert Davidson from Microsoft Research. Jason Casmira and David Kaeli have been supported by a National Science Foundation CAREER grant.

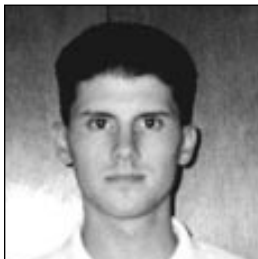
References and Notes

1. *SPEC Newsletter* (September 1995).
2. Information about the BYTEmark benchmark suite is available from *BYTE Magazine* at <http://www.byte.com/bmark/bmark.htm>.
3. S. Perl and R. Sites, “Studies of Windows NT Performance Using Dynamic Execution Traces,” *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation* (October 1996): 169–183.
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): 224–244.
5. R. Uhlig and T. Mudge, “Trace-Driven Memory Simulation: A Survey,” *ACM Computing Surveys*, vol. 29, no. 2 (June 1997): 128–170.

6. J. Emer and D. Clark, “A Characterization of Processor Performance in the VAX 11-780,” *Proceedings of the Eleventh Symposium on Computer Architecture* (June 1994): 126–135.
7. K. Flanagan, J. Archibald, B. Nelson, and K. Grimrud, “BACH: BYU Address Collection Hardware; The Collection of Complete Traces,” *Proceedings of the Sixth International Conference on Modeling Techniques and Tools for Computer Evaluation* (1992): 51–65.
8. D. Kaeli, O. LaMaire, W. White, P. Hennes, and W. Starke, “Real-Time Trace Generation,” *International Journal on Computer Simulation*, vol. 6, no. 1 (1996): 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*, vol. 41, no. 3 (May 1997): 205–214.
10. D. Nagle, R. Uhlig, and T. Mudge, “Monster: A Tool for Analyzing the Interaction Between Operating Systems and Computer Architectures,” Technical Report, CSE-TR-147-92, University of Michigan, 1992.
11. B. Chen and B. Bershad, “The Impact of Operating System Structure on Memory System Performance,” *Operating Systems Review*, vol. 27, no. 5 (December 1993): 120–133.
12. J. Larus, “Abstract Execution: A Technique for Efficiently Tracing Programs,” Technical Report, CS-TR-90-912, University of Wisconsin-Madison, 1990.
13. A. Srivastava and A. Eustace, “ATOM: A System for Building Customized Program Analysis Tools,” *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation, Orlando, Fla.* (June 1994): 196–205.
14. M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, “Complete Computer System Simulation: The SimOS Approach,” *IEEE Journal of Parallel and Distributed Technology*, 1998, forthcoming.
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): 78–103.
16. A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming* (Kluwer Academic Publisher, 1989).
17. J. Larus and E. Schnarr, “EEL: Rewriting Executable Files to Measure Program Behavior,” *Proceedings of the ACM SIGPLAN’95 Conference on Programming Language Design and Implementation, La Jolla, Calif.* (June 1995): 291–300.

18. D. Lee, P. Crowley, J.-L. Baer, T. Anderson, and B. Bershad, "Execution Characteristics of Desktop Applications on Windows NT," *Proceedings of the Twenty-fifth International Symposium on Computer Architecture*, Barcelona, Spain (June 1998).
19. E. Betts, D. Hunter, and S. Smith, "Moving ATOM to Windows NT for Alpha," *Digital Technical Journal*, vol. 10, no. 2, accepted for publication.
20. M. Smith, "Tracing with Pixie," Technical Report, CSL-TR-91-497, Stanford University, November 1991.
21. R. Cmelik and D. Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling," *Proceedings of ACM Sigmetrics* (May 1994): 128-137.
22. *Alpha AXP Architecture Handbook*, Order No. EC-QD2KA-TE (Maynard, Mass.: Digital Equipment Corporation, October 1994).
23. H. Custer, *Inside Windows NT* (Redmond, Wash.: Microsoft Press, 1993).
24. Microsoft Software Developer's Toolkit. This toolkit is available at <http://msdn.microsoft.com/developer/sdk/platform.htm>.
25. J. Casmira, "Operating System Rich Workload Characterization," Master's thesis, ECE-CEG-98-018, Northeastern University, May 1998.
26. R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, vol. 9, no. 1 (1997): 3-12.

Biographies



Jason P. Casmira

Jason Casmira received B.S. and M.S. degrees in electrical engineering from Northeastern University in 1996 and 1998, respectively, and is pursuing a Ph.D. degree in computer science at the University of Colorado, Boulder. For the past two years, Jason was a member of the Northeastern University Computer Architecture Research Laboratory (NUCAR), where he focused on developing the current version of the PatchWrX tracing toolset. He also investigated issues related to studying operating system-rich traces. While at NUCAR, Jason was supported by a grant from the National Science Foundation. He has published seven papers and is a member of the IEEE and the Eta Kappa Nu honor society.



David P. Hunter

David Hunter is the engineering manager of Compaq Computer Corporation's Advanced and Emerging Technologies Group. Prior to that he was the manager of DIGITAL's Software Partner Engineering Advanced Development Group, where he was involved in performance investigations of databases and their interactions with the UNIX and Windows NT operating systems. He has held positions in the Alpha Migration Organization, the ISV Porting Group, and the Government Group's Technical Program Management Office. David joined DIGITAL's Laboratory Data Products Group in 1983, where he developed the VAXlab User Management System. He was the project leader of the advanced development project, ITS, an executive information system, for which he designed hardware and software components. David has two patent applications pending in the area of software engineering. He holds a degree in electrical and computer engineering from Northeastern University in Boston, Massachusetts, and a diploma in National Security and Strategic Studies from the United States Naval War College in Newport, Rhode Island.



David R. Kaeli

David Kaeli received Ph.D. (1992) and B.S. (1981) degrees in electrical engineering from Rutgers University and an M.S. degree in computer engineering from Syracuse University in 1985. He joined the electrical and computer engineering faculty at Northeastern University in 1993 after spending 12 years at IBM, the last 7 of which were at the IBM T. J. Watson Research Center in Yorktown Heights, New York. David is the director of the Northeastern University Computer Architecture Research Laboratory (NUCAR), where he investigates the performance and design of high-performance computer systems and software. His current research topics include I/O workload characterization, branch prediction studies, memory hierarchy design, object-oriented code execution performance, 3-D microelectronics, and back-end compiler design. He frequently gives tutorials on the subject of trace-driven characterization and simulation. In 1995, David received the prestigious National Science Foundation CAREER Award. His research has been supported by the Office of Naval Research, Kopin Corporation, Digital Equipment Corporation, EMC, Data General, Microsoft Research, I-Tech Corporation, IEEE DAC, and IBM Research. David is a member of the ACM, IEEE, and the Eta Kappa Nu and Sigma Xi honor societies.