Execution Characteristics of Desktop Applications on Windows NT

Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer,
Thomas E. Anderson, and Brian N. Bershad

Abstract

This paper examines the performance of desktop applications running on the Microsoft Windows NT operating system on Intel x86 processors, and contrasts these applications to the programs in the integer SPEC95 benchmark suite. We present measurements of basic instruction set and program characteristics, and detailed simulation results of the way these programs use the memory system and processor branch architecture. We show that the desktop applications have similar characteristics to the integer SPEC95 benchmarks for many of these metrics. However, compared to the integer SPEC95 applications, desktop applications have larger instruction working sets, execute instructions in a greater number of unique functions, cross DLL boundaries frequently, and execute a greater number of indirect calls.

1 Introduction

Progress in microarchitecture, memory hierarchy, and instruction-level parallelism quickly outpaces the benchmarks that are designed to assess the performance of modern computer systems. It is clear that SPEC92 is already obsolete. Newer benchmark suites such as SPEC95 [SPEC 95] and Instruction Benchmark Suite [Uhlig et al. 95] for scientific and engineering applications, server and commercial applications [Maynard et al. 94], and parallel applications [Woo et al. 95] have emerged to provide a rich set of insights about application performance characteristics from a UNIX-based batch application environment. However, most of the world's personal computers and workstations run some flavor of the Microsoft Windows operating system on Intel x86 processors. Moreover, most of these computers run personal productivity and entertainment applications rather than engineering or server workloads.

Recently, there has been some progress in understanding applications running under the Windows NT operating system. [Chen et al. 95, Endo et al. 96] have looked at the characteristics of different operating systems running on top of the Intel x86 processors using performance counters. [Perl & Sites 96] have measured the memory performance of three Windows NT applications (a SPEC application, a compiler back-end and a database application) running on the DEC Alpha platform. However, no clear picture of the execution characteristics of desktop applications running on Windows NT on the x86-platform has been published.

This paper makes two contributions. First, it characterizes a set of instruction and address traces from the runs of five common desktop applications running on Windows NT. Second, using these traces, it compares these desktop applications to the SPEC95 applications from the perspective of the processor architecture and memory system. Our traces were generated using Etch, a binary instrumentation engine developed for Windows NT on the x86 platform [Romer et al. 97].

There are some obvious qualitative differences between desktop applications and traditional benchmarks that intuitively suggest differences in hardware utilization and program behavior:

- Desktop applications are interactive. The execution path of interactive programs may be less predictable than the paths of batch applications since the user provides input that directs execution during the lifetime of the application.
- Desktop applications are graphical. While traditional benchmarks output their results in a text file consisting of a few pages of output, many desktop applications draw tens or hundreds of graphical widgets and pictures to the screen in addition to generating more conventional output.
- Desktop applications are feature rich. While traditional benchmark applications often perform a single

The authors can be reached through the Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195-2350, or through email at {dlee,pcrowley,baer,tom,bershad}@cs.washington.edu. This work was supported in part by NSF Grant MIP-9700970 and by Intel Corporation. Thomas Anderson was supported by an NFS Presidential Faculty Fellowship. Dennis Lee was supported by a National Semiconductor Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the University of Washington, Intel Corporation, or National Semiconductor Corporation.

task, most desktop applications support many functions in addition to their main task (e.g., drawing graphs in Powerpoint). These features lead to bigger binaries and make procedure calls across the address space more likely.

- Desktop applications tend to be multithreaded both as a structuring mechanism and as a way of hiding long latency operations from the user.
- Desktop applications tend to more heavily use shared system libraries (Dynamic Linked Libraries or DLLs) to capture commonly used functionality, both as a way of saving memory and disk space, and of presenting a uniform "look and feel" to the user.

Whether these differences in qualitative properties translate into significant quantitative dissimilarities in program behavior characteristics and architectural performance is an open question. For example, one could expect interactive applications that use many different DLLs and an increased level of multithreading to exhibit worse locality than batch applications. Graphical output might also involve larger code and data working sets, thus influencing the memory hierarchy performance.

Our results show that desktop applications differ from SPEC95 integer applications in two important ways: First, desktop applications have a bigger instruction working set size due to the fact that they tend to call a greater number of distinct functions. Second, the desktop applications tend to execute many more indirect calls. However, in many other respects the desktop applications are similar to the SPEC95 benchmarks.

The rest of the paper is organized as follows: Section 2 describes our methodology. Section 3 describes the applications in our benchmark suite and considers their high level application characteristics. Section 4 presents measurements of these applications with traditional architecture metrics for cache and branch prediction behavior. Finally, Section 5 summarizes our findings.

2 Methodology

2.1 Etch

We use *Etch* [Romer et al. 97], a binary instrumentation engine for the x86-Windows NT platform, to gather user level traces. Etch allows a user-provided instrumentation module to insert arbitrary code at well-defined points in the binaries. For our study, we use Etch to insert calls to tracing routines at each instruction and at each load and store; Etch provides instruction addresses of the original unetched binary and load-store effective addresses when it calls our tracing routines. The techniques used by Etch are very similar to those employed by other static instrumentation engines such as QPT [Larus & Ball 92], EEL [Larus & Schnarr 95], and Atom [Srivastava & Eustace 94].

To ensure that we had consistent input to an application across all our metrics, we wrote a Visual Test [Rational Software Corporation 96] script for each desktop application. We then gathered traces on a single machine in order to eliminate the effects due to the differences between the states of different machines (e.g., the locations and versions of shared libraries).

Aside from containing instruction, data, and branch addresses, the traces also include additional information about the current state of the program: which module, procedure, and thread is currently executing. This extra information is meant to aid in associating high level application behavior with low-level architecture metrics.

2.2 Limitations

This subsection discusses the limitations of our methodology and caveats for the reader to be aware of when interpreting the numbers presented in this paper. These limitations fall into two broad categories: limitations of Etch and limitations due to the nature of software instrumentation.

Although Etch can instrument most NT binaries, it currently does not understand kernel drivers and occasionally fails to instrument some DLLs (These DLLs either have hand-coded assembly routines or communicate with other processes through well-known addresses). To quantify the extent of these limitations we used VTune [Intel Corporation 97], a PC sampling tool from Intel to determine if we were missing a significant fraction of the instructions from unetched modules. Using VTune, we determined that the kernel driver responsible for graphics and windowing functionality (*win32k.sys*) constitutes from 15 to 30% of the application total instruction count for a given application run. Given this, and the general rule of thumb that operating system code has less locality than application code, our numbers for cache miss rates and branch predictability should be interpreted as being optimistic. Running VTune on our benchmarks also show that we miss less than 1% of executed user-level instructions.

Using software instrumentation to gather traces also presents a series of issues. First, software instrumentation tools insert code into the binary and thus change the addresses seen by the CPU [Srivastava & Eustace 94] (for our tracing tool, Etch expands the size of the code by a factor of 32). Etch compensates for this by sending the instruction addresses that came from the original unetched program to the tracer rather than the address of the etched binary. A similar mechanism is used to fix references to data addresses in the module itself. Heap addresses, though, are offset due to the increased size of the binaries. The addresses should just be a fixed offset off of the original heap so cache results should still be valid since the indices depend on the low order bits.

Second, tracing slows down the application significantly.

Application	Description	Instructions Executed (millions)
acrord32	Adobe Acrobat Reader 3.0: Reader for portable document format (PDF) files. The benchmark loads acrobat.pdf (a 277 KB file) from the standard acrobat reader distribution, and navigates through the document three different ways: through the hyperlinks in the document itself, through the forward and back button provided by acrobat reader, and through a view of the document outline provided by acrobat reader. Finally, the benchmark searches for the word "buy" in the document before closing the program.	408
netscape	Netscape Navigator 3.1 web browser. The benchmark opens four web pages: `www.cs.washington.edu`, `www.cnn.com`, `www.mtv.com`, and `www.washington.edu`. These pages were viewed on March 18, 1998. The java module for netscape was turned off because Etch does not handle the dynamically generated code generated by the java just-in-time compiler.	92
photoshp	Adobe Photoshop 4.0 image editing package. The benchmark loads fruit.jpg (a 591 KB still-life photograph of fruit) from the standard distribution and applies the *color pencil*, *accented edges*, *diffuse glow*, and *add noise* photo filters to the image.	1,511
powerpnt	Microsoft PowerPoint 7.0b slide preparation package. The benchmark loads in a 311 KB 18-page presentation (the presentation included five pages of graphs and six pages of figures in addition to text) in slide mode, scrolls through 3 pages, edits a figure, and continues scrolling through until the end of the document. The benchmark then goes into the outline mode and creates a new page and goes back into the slide mode to move text around. Finally, the benchmark goes into slide sorter mode and moves some slides around.	209
winword	Microsoft Word 7.0 word processor. The benchmark simulates a user typing in seven paragraphs in an eight page document (document size is 29K). The benchmark then performs four search and replace commands on the document before saving a text version of the file. The interactive spell checker was turned on.	351
compress	SPEC95 version of the UNIX compress utility. The benchmark generates a file of a given size using a random number generator then compresses and decompresses the file. The measurements were for a file of size 136K.	403
gcc	SPEC95 version of the GNU C compiler (version 2.5.3) which takes a preprocessed source file and converts these into Sparc assembly. The input to gcc was cccp.i from the test directory in the SPEC95 distribution.	1,158
go	SPEC95 version of the game of go. The benchmark was run with a play level of 40 and a 10x10 board size.	315
perl	SPEC95 version of the popular Perl interpreter (version 4.0.1.8). The benchmark was run with jumble.pl and jumble.in from the test directory in the SPEC95 distribution.	2,013
vortex	SPEC95 version of a single-user object-oriented database transaction benchmark. The input to vortex was the database and schema provided in the test directory in the SPEC95 distribution.	2,147

Table 1. *Benchmarks used for this study. The SPEC95 applications were compiled with Microsoft Visual C++ (MSVC) 4.2 using the makefiles from the distribution. The traces of these applications were produced on a dual Pentium Pro 200 system running Windows NT Workstation 4.0 service pack 3.*

For a small application, the traced application runs about 85-100 times slower than the original. For most applications, running slower means that external events like user input, disk I/O, and screen refreshes appear to complete faster [Eggers et al. 90]. We cope with this time dilation effect in two ways. First, our scripts feed commands to the application based on the response from the application, so the effect on the rate of user input should be minimal. Second, since most of the applications are not disk intensive and would wait for screen refreshes anyway, the effect due to "faster" disk and video subsystems is minimal. The only exception here is thread preemptions. Applications running slower means that thread preemptions happen more frequently. However, 4 of the 5 desktop applications spent greater than 97% of their time in a single thread, suggesting that preemptions were rare.

3 Application Characteristics

This section first describes the five desktop and the five SPEC95 integer benchmarks used in this study. We contrast their overall characteristics, namely the size of their executables and the use of DLLs. We then look at the implications of the execution environment provided by Windows NT (i.e., the use of multithreading and the presence of shared DLLs). In the third part of this section, we look at instruction set and program structure characteristics at a level closer to the architecture. We look at measures such as proportion of load/store instructions, basic block lengths, control flow instructions, and function characteristics. The numbers here do not directly measure the performance of the programs on any specific architecture, rather they broadly characterize the applications and point out similarities and differences between the desktop and the SPEC95 applications.

3.1 The Benchmark Suite

We measured five Windows NT applications that represent a variety of different tasks commonly performed on desktop computers. Table 1 gives the description of these applications as well as that of the SPEC95 programs that we used for comparison purposes. Table 2 presents the characteristics of the object files that compose the different applications. Windows NT applications are composed of the application binary, shared system DLLs, and application specific DLLs.

The first striking difference between the desktop and the SPEC95 applications is that the desktop applications have much larger executable file sizes: from 3 to 10 times larger than the SPEC95 applications. This is to be expected as the desktop applications tend to have more features which result in larger code sizes. The second major difference is that desktop applications make extensive use of DLLs, many of these shared with other applications, while the SPEC95 applications do not. The desktop applications tend to use on average 30 DLLs, increasing their memory footprint by a factor of three. Each application used between 21 and 25 shared system DLLs, the remaining ones being application specific. These shared system DLLs contain functionality commonly used in different applications, including access to operating system services, user interface functions, a graphics library, networking, a user interface library, and a C runtime library. On average the shared system DLLs are about 212 KB in size, contributing approximately 4 MB to the memory footprint of the application. In contrast, the SPEC95 applications incorporate only two system DLLs contributing 727 KB to their memory footprint. The larger binary sizes of the desktop applications may lead to more function calls that cross page boundaries and conflict in the instruction cache.

Application	Executable Size (MB)	Size with DLLs (MB)	# DLLs used (shared)
acrord32	2.26	9.73	34 (24)
netscape	3.17	9.95	28 (24)
photoshp	3.65	13.5	44 (25)
powerpnt	4.36	12.5	26 (21)
winword	3.78	11.2	26 (21)
compress	0.122	0.849	2 (2)
gcc	1.15	1.88	2 (2)
go	0.295	1.02	2 (2)
perl	0.323	1.05	2 (2)
vortex	0.570	1.30	2 (2)

Table 2. *Application object file characteristics. Desktop applications have larger executable file sizes than the SPEC95 benchmarks. The total footprint (size with DLLs) of the desktop applications is even larger. The desktop applications use an average of 30 DLLs with over 20 of them being shared across all applications. The SPEC applications only used kernel32.dll and ntdll.dll. Even if the SPEC benchmarks were linked with the shared C library (msvcrt.dll), this would have only increased the total footprint of each application by 281 KB.*

Application	# of Trds	Prim Trd(%)	Thread 2 (%)	Thread 3(%)	Others (%)
acrord32	3	98.63	1.37	0.00	-
netscape	4	99.58	0.26	0.16	0.00
photoshp	5	97.16	2.84	0.00	0.00
powerpnt	8	78.93	18.38	2.56	0.14
winword	3	99.92	0.08	0.00	-

Table 3. *Thread instruction distribution. Nearly all instructions are executed in the primary thread.*

3.2 Impact of Execution Environment

The desktop applications make extensive use of the execution environment provided by Windows NT. Programming to the services provided by this environment makes the desktop applications different from the SPEC95 applications in two respects: desktop applications are multithreaded and they use DLLs extensively.

To determine the potential impact of threads on application performance, we counted the number of instructions executed by each thread in the program. Table 3 shows the distribution of instructions among different threads in each application. With the exception of *powerpnt*, very little is done outside of the primary thread, so we expect minimal impact on performance due to resource contention among the threads.

Table 4 shows the distribution of instructions across the application binary and DLLs. As shown, the desktop applications execute an average of 11% of their instructions in three system DLLs (*user32.dll*, *gdi32.dll*, and *kernel32.dll*). While the execution stream of the programs reside mostly in the applications and application specific DLLs, the effects of these three system DLLs cannot be ignored.

To determine how DLLs are used by applications, we looked at the length of time spent inside DLLs. DLL calls are more expensive than calling statically linked functions: i) Unlike regular function calls, DLL calls are implemented as indirect function calls; ii) DLLs are shared between applications, so improving instruction locality through traditional reordering algorithms [Pettis & Hansen 90] is more difficult; and iii) the caller and callee must live in different pages in the address space (as DLLs are aligned on page boundaries). Frequent crossing of DLL boundaries will require pages from both DLLs to be resident in the address space even though only a small portion of these pages are actually used by the application (i.e., there is internal fragmentation).

Table 5 shows the distribution of the number of instructions executed between the time an application performs a call to a DLL to the time the application executes again. For example, the table shows that when *acrord32* performs

Application	app	user32.dll	gdi32.dll	kernel32.dll	other(>1%)	other(<1%)
acrord32	94.0	1.6	0.8	3.0	0.0(0)	0.6(31)
netscape	56.1	3.9	3.7	4.2	30.6(3)	1.5(22)
photoshp	36.8	3.3	0.3	3.4	55.8(3)	0.5(38)
powerpnt	75.9	5.0	6.7	10.9	1.1(1)	0.5(22)
winword	83.3	7.6	1.7	3.1	3.5(2)	0.9(21)

Table 4. *Percentage of instructions executed in different program modules. The first "other" column shows the sum of instructions executed in the remaining DLLs that contribute more than 1% to the total execution, and the second "other" column sums the remainder. The number of DLLs contributing to these totals are in parentheses. The SPEC95 benchmarks only use kernel32.dll and ntdll.dll, and each of these DLLs contribute less than 1% of the instructions executed. Netscape relies heavily on mfc40.dll (8.0%): the Microsoft foundation class library, a C++ class library that encapsulates the programming interface to the Win32 API; and msvcrt.dll (20.0%) : the C run-time library. Photoshp uses three modules: accented.8bf (9.0%), coloredp.8bf (34.4%), and diffuseg.8bf (12.4%), to implement the image filters we used in our experiment. No other shared system DLL contributed more than 1% for any of the applications.*

Application	Length of DLL Call		DLL Res. Length	
	50th %-tile	95th %-tile	50th %-tile	95th %-tile
acrord32	35	157	7	26
netscape	43	411	10	45
photoshp	36	217	12	39
powerpnt	45	571	19	52
winword	62	198	17	50

Table 5. *Instructions spent in DLLs. The length of DLL call gives the number of instructions executed between an application call to a DLL and its return. The DLL residence length give the number of instruction an application stays in a single DLL before transferring to a different DLL. Most DLLs calls are short and involve calls to other DLLs.*

a DLL call, half of these calls return after executing fewer than 35 instructions. In fact, 95% of all DLL calls return to the application after executing fewer than 571 instructions. This implies that applications use DLLs mostly as function libraries. A small number of these calls (about 20%) actually end up calling into the operating system. If these operating system calls are long lived, then there may be phases of the program where the performance characteristics would be operating system call specific. However, for most DLL calls, this is not the case.

Table 5 also shows the distribution of instructions executed inside any particular DLL before control transfers to a different DLL. These instruction sequences are very short, most sequences execute less than 60 instructions inside any particular DLL. In other words, desktop applications cross DLL boundaries frequently, paying the price of the greater number of indirect function calls and crossing page boundaries frequently.

To see if there is an opportunity to optimize the application executable through better DLL code layout, we measured the distribution of instructions in the different modules that comprise the executable. The same 30% of all the functions in *user32*, 13% of all the functions in *gdi32*, and 28% of all the functions in *kernel32* contributes more than 95% of the total number of instructions executed in these DLLs in *all* the applications. This suggests that one may be able to optimize the memory footprint of these applications through techniques like Just-In-Time Code Layout [Chen & Leupen 97] which loads in DLLs one function at a time. However, further study is needed to validate this observation for most applications and most users.

3.3 Instruction Set and Function Call Characteristics

Instruction set measurements give a high level approximation of the usage pattern of a program. These usage patterns paint a broad picture of the architecture characteristics of a program and point out areas where effort needs to be concentrated [Patterson & Hennessy 96].

Application	Loads/ Inst	Stores/ Inst	Ctrl Flow/ Inst	Avg BB Len(inst)
acrord32	0.34	0.24	0.15	6.6
netscape	0.35	0.19	0.21	4.7
photoshp	0.35	0.22	0.15	6.6
powerpnt	0.36	0.20	0.23	4.3
winword	0.52	0.37	0.19	5.3
compress	0.35	0.17	0.19	5.2
gcc	0.35	0.19	0.21	4.7
go	0.40	0.17	0.21	4.7
perl	0.32	0.22	0.21	4.7
vortex	0.41	0.30	0.19	5.2
Avg(Dtop)	0.38	0.24	0.19	5.5
StdDev	0.08	0.07	0.04	1.1
Avg(SPEC)	0.37	0.21	0.20	4.9
StdDev	0.04	0.05	0.01	0.3

Table 6. *General benchmark characteristics. For these metrics, the desktop applications look similar to the SPEC applications.*

The ratio of memory reference and control flow instructions to the total number of instructions reflects the relative importance of the memory system and the branch prediction

Application	% Ctrl Flow	% Calls	% Rets	% Brches	% Brks	Inst/ Brk
acrord32	15.1	1.1	1.1	12.8	10.2	9.8
netscape	21.3	3.0	2.0	16.3	12.9	7.7
photoshp	15.1	1.4	1.4	12.3	6.0	16.6
powerpnt	23.0	4.5	3.8	14.7	12.5	8.0
winword	18.9	2.2	2.1	14.6	12.0	8.4
compress	19.4	2.9	2.9	13.6	12.3	8.2
gcc	21.1	1.7	1.7	17.8	12.7	7.9
go	21.2	1.6	1.6	17.9	13.4	7.5
perl	21.1	2.2	1.9	17.1	14.5	6.9
vortex	19.1	2.5	2.5	14.1	9.8	10.2
Avg(DTop)	18.7	2.4	2.1	14.1	10.7	10.1
StdDev	3.6	1.4	1.0	1.6	2.8	3.7
Avg(SPEC)	20.4	2.2	2.1	16.1	12.5	8.1
StdDev	1.0	0.5	0.6	2.1	1.7	1.3

Table 7. *Control Flow Characteristics. The table shows a breakdown in the control flow instructions of the program. % Breaks shows the percentage of instructions which cause a break in the sequential flow of the instructions.*

Application	Branches				Calls		
	% CB	% (T)	% DB	% IB	% DC	% IC	% DLL
acrord32	89.3	(58)	9.9	0.7	82.3	10.7	7.0
netscape	91.2	(44)	7.6	1.2	50.2	16.0	33.9
photoshp	97.1	(24)	2.6	0.3	85.5	8.3	6.1
powerpnt	94.8	(24)	5.0	0.2	68.9	20.1	11.0
winword	89.5	(47)	10.1	0.4	87.2	5.8	7.0
compress	90.7	(42)	9.3	0.0	100.0	0.0	0.0
gcc	92.3	(49)	3.7	3.9	98.0	2.0	0.0
go	95.7	(55)	4.0	0.3	100.0	0.0	0.0
perl	89.5	(57)	6.0	4.5	87.2	12.7	0.0
vortex	93.1	(29)	6.6	0.3	100.0	0.0	0.0
Avg(DTop)	92.4	(39)	7.0	0.6	74.8	12.2	13.0
StdDev	3.4	15	3.2	0.4	15.5	5.8	11.8
Avg(SPEC)	92.3	(46)	5.9	1.8	97.0	2.9	0.0
StdDev	2.4	11	2.3	2.2	5.6	5.5	0.0

Table 8. *Branch and Call Breakdown. We show the breakdown of branches into conditional branches (CB), direct unconditional branches (DB), and indirect unconditional branches (IB). We also show the percentage of conditional branches that are Taken (T). Under the Call category, we show the breakdown of calls into direct calls (DC), indirect calls (IC), and DLL calls (DLL). We differentiate between indirect calls and DLL calls because although DLL calls are implemented as indirect calls, they call a single target for the entire duration of a program run.*

architecture to the performance of the program. Table 6 indicates that as far as the instruction set is concerned, the desktop applications and the SPEC95 applications are very similar. Both average about 0.38 loads and 0.24 stores per instruction. Both average close to the same number of control flow instructions per instruction (around 0.19).

To get a better sense of the demands that the desktop applications will have on the branch prediction architecture, we broke down the control flow instructions into different types. We also measured the number of breaks in sequential flow of instruction in the application. The number of instructions per break gives an indication of the extent to which instruction prefetching would be beneficial. Table 7 shows that with respect to control flow instructions, desktop applications and SPEC95 again are very similar.

Table 8 shows the break down of the types of calls and branches in the applications. This table shows that for control flow characteristics, desktop applications differ from the SPEC95 applications in only one important respect: desktop applications use a higher proportion of indirect calls: averaging about 25% (IC + DLL) versus none for SPEC95 with the exception of *perl*. Although these indirect calls still constitute a small fraction of the total number of instructions executed, it does imply that microarchitecture improvements that rely on being able to extract long sequences of instructions from the execution stream (e.g., trace caches) need to pay more attention to indirect calls than would be suggested by the SPEC95 benchmark suite.

The increased number of indirect calls in desktop applications may imply that most of the desktop applications are written in an object oriented style. [Calder et al. 94] observed that on average, C++ programs tend to have more indirect calls because C++ programs tend to use dynamic dispatch (i.e., virtual function calls) to take the place of conditional logic (i.e., if-then-else or switch statements). For our benchmark programs, *netscape*, *photoshp*, and most of *powerpnt* are written in C++, while *acrord32* is written in object-oriented style C. *Winword* is written mostly in C. However, [Calder et al. 94] also found that C++ programs tend to have smaller function sizes and longer basic block sizes. None of the desktop applications have particularly longer basic blocks and only *powerpnt* has significantly smaller functions.

To determine the impact of the larger executable size on the application performance, we measured the number of unique (static) functions that an application touches during its run. The application binary may be large but the application may only use a small fraction of it at a time. Table 9 shows that the desktop applications call at least one order of magnitude more unique functions than the SPEC95 programs; this is not surprising since the application executable is bigger and the function sizes are about equal: the desktop applications simply have more functions to perform. However, the number of unique functions is not correlated to trace length; program execution tends to be dispersed across more functions compared to the SPEC95 applications. This increases the probability of conflict and capacity misses in the instruction cache.

Application	# Insts Exec'ed (millions)	# Funcs Called (millions)	# Unique Funcs	Avg Insts/ Func
acrord32	408	4.49	7,882	90.9
netscape	92	2.76	5,766	33.3
photoshp	1,511	21.15	9,598	71.4
powerpnt	209	9.40	11,012	22.2
winword	351	7.72	6,909	45.5
compress	403	11.69	143	34.5
gcc	1,158	19.67	1,310	58.8
go	315	5.04	450	62.5
perl	2,013	44.29	334	45.5
vortex	2,147	53.67	707	40.0
Avg(Dtop)			8,233	52.7
StdDev			2,095	28.1
Avg(SPEC)			589	48.3
StdDev			452	12.0

Table 9. *Use of functions in desktop and SPEC95 applications. # Unique Functions refer to the number of unique static functions that are touched by the program.*

4 Architectural Characteristics

Two major sources of performance degradation in current microprocessor systems are stalls due to the memory system and incorrect prediction of the speculative path. In this section, we present measurements of the performance of desktop applications on several typical memory and branch prediction structures, and compare these measurements with similar ones for the SPEC95 benchmarks.

4.1 Cache Behavior

To measure the impact of the structure of desktop applications (i.e., large number of functions and frequent DLL calls) on the memory subsystem, we measured the cache miss rate of the applications for cache and associativities that are typical of today's machines. Figure 1 shows the results of these measurements.

For small direct mapped instruction caches, three of the desktop applications (*netscape*, *powerpnt*, and *winword*), and four of the SPEC applications (*gcc*, *go*, *perl*, and *vortex*) have poor instruction cache performance. Except for *winword* and *gcc*, many of these misses are conflict misses and disappear with a 4-way associative cache. *Winword*, *powerpnt*, and *gcc* appear to suffer from mostly capacity misses. As the cache size increases, the miss rates of these three applications all decrease by a substantial fraction. *Photoshp* sits in a tight loop while applying a photo filter to a large picture, hence the low miss rate.

Notice that as instruction cache sizes become bigger (around 32 KB), the desktop applications start to improve less rapidly than the SPEC95 applications. For example, *netscape* has better I-cache behavior than most of the SPEC95 applications for caches less than or equal to 16 KB. However, at 32 KB, *netscape* has about the same miss rate as *gcc* and *vortex*, and becomes worse at 64 KB. This higher miss rate with larger caches is probably due to the fact that the desktop applications reference many more unique functions than the SPEC95 applications (cf., Section 3.2). The working sets of neither the desktop applications nor of the SPEC95 applications fit in the small caches. However, slightly larger caches are able to capture most of the functions touched by the SPEC95 applications, but the larger number of unique static functions needed by desktop applications still do not fit in the cache.

For the data caches, our measurements suggest that the desktop applications and the SPEC95 applications have comparable performance. For example, except for *compress* and *winword*, all the applications have miss rates below 2% for a 16 KB 4-way associative cache. Both sets of applications suffer from conflict misses as shown by the big drop in miss rates going from a direct mapped to a 4-way set associative cache (e.g., comparing 16 KB caches, the miss rate of *photoshp* drops from 3.5% to 1.1%, similarly the miss rate of *go* drops from 8.2% to around 1.7%).

For combined caches, the verdict for direct mapped caches is more muddied: *acrord* and *photoshp* have lower miss rates than most of the SPEC95 applications while *winword* has a higher miss rate. Again our measurements suggest that both the desktop applications and the SPEC95 applications suffer from conflict misses. The miss rates improve significantly moving from a direct mapped to 4-way set-associative caches; this is especially telling with *go* (from 1% to almost 0% for 4 MB combined caches), and to a lesser extent *winword* (from 1% to 0.13% for 256 KB combined caches).

4.2 TLB Behavior

Frequent calls to many different small functions in a big executable and frequent switches between DLLs take a greater toll on the instruction TLB than on the instruction cache. With a big executable, it is likely that many of the functions live in different pages in the address space. This is definitely the case for functions in different DLLs. Figure 2 shows the TLB behavior of our benchmark applications. The instruction TLB behavior of the desktop applications tend to be worse than that of the SPEC95 benchmarks. For example, with a 32-entry instruction TLB, all the desktop applications have a TLB miss rate greater than 0.05% while only *gcc* and *vortex* have comparable miss rates for the SPEC95 applications. Even *photoshp* which had a very low instruction cache miss rate shows a TLB miss rate comparable to *gcc* for a 32-entry ITLB.

For the data TLB, our measurements indicate that the desktop applications have better locality than the SPEC95 applications. For example, at 32 entries, none of the desktop applications has greater than a 0.75% miss rate while only *gcc* among the SPEC95 applications has less than a 0.75% miss rate. Interestingly, while *photoshp* and *acrord32* had

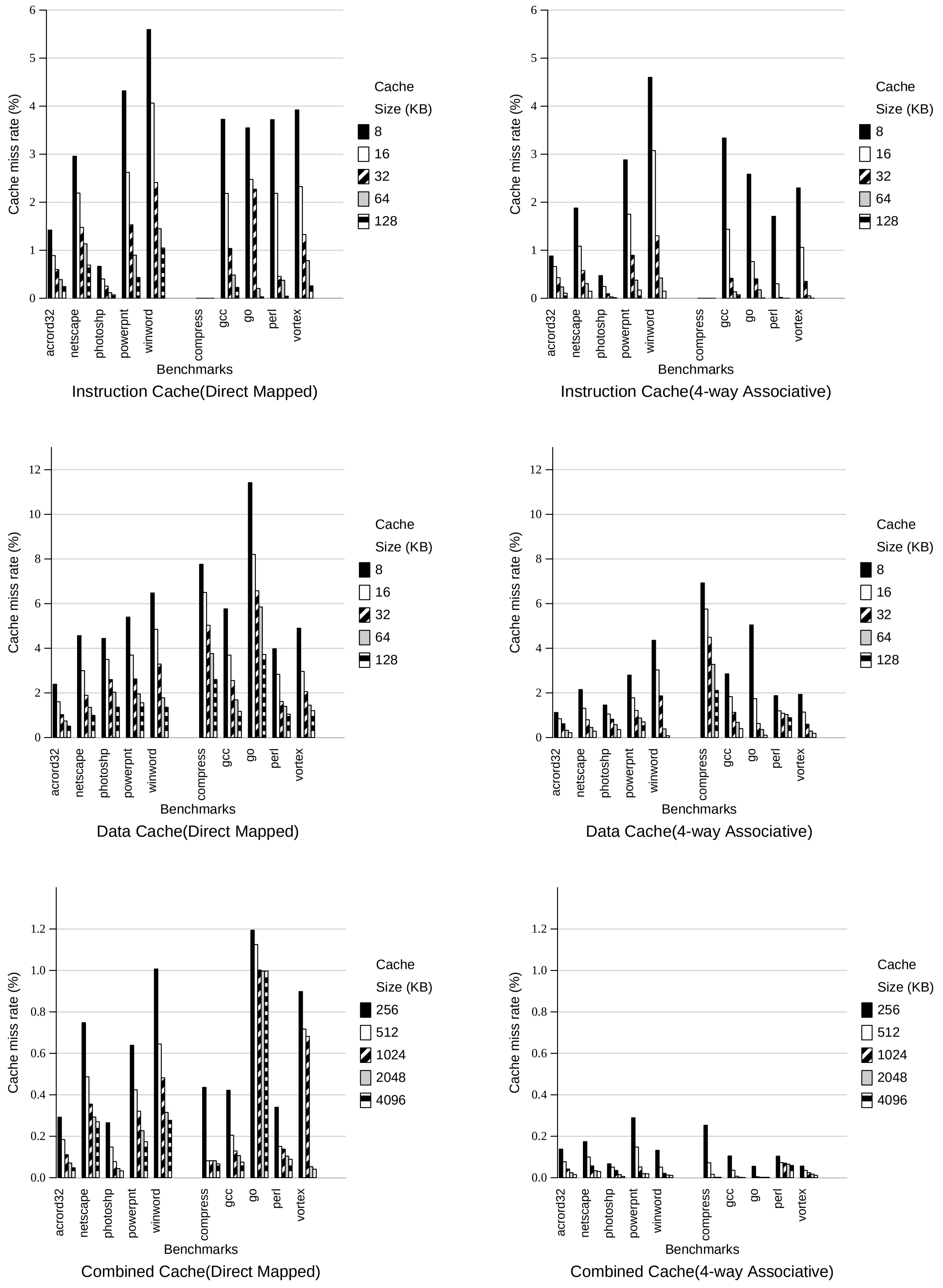
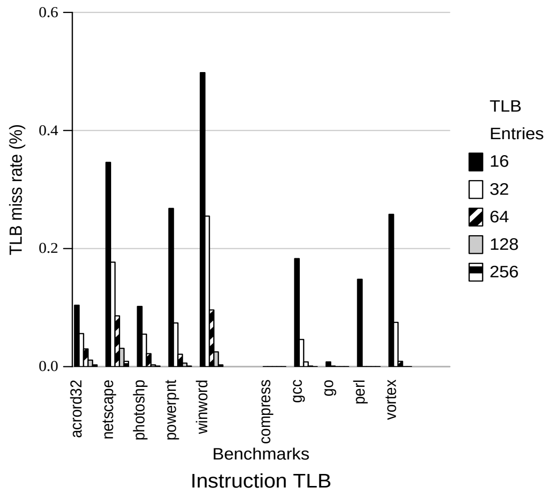


Figure 1. *Cache performance. This figure shows the miss rates for different sized Instruction, Data, and Combined caches, for direct-mapped and 4-way associative caches. Line size is 32-bytes. Write policy is write-allocate/write-back. For the data and combined caches, we show the miss rates for data loads. We simulated the effects of stores on the cache state but do not report misses due to stores because, in general, processors do not block on stores [Patterson & Hennessy 96]. We did not simulate a virtual to physical mapping policy so all the addresses seen by the caches are virtual addresses.*



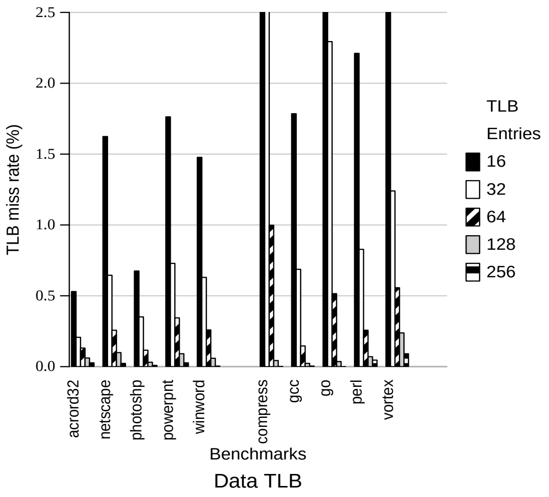


Figure 2. *TLB performance. The figure shows the TLB performance of Instruction and Data TLBs. The TLBs are 4-way set associative. (Miss rates for numbers that go beyond the top of the graph: for 16-entry DTLB, compress has a miss rate of 7.4%, and go has a miss rate of 7.5%; for 32-entry DTLB, compress has a miss rate of 2.7%)*

data cache miss rates comparable to *netscape* and *powerpnt* (cf., Figure 1), they have much better data TLB behavior than either of the two.

4.3 Predicting Conditional Branches

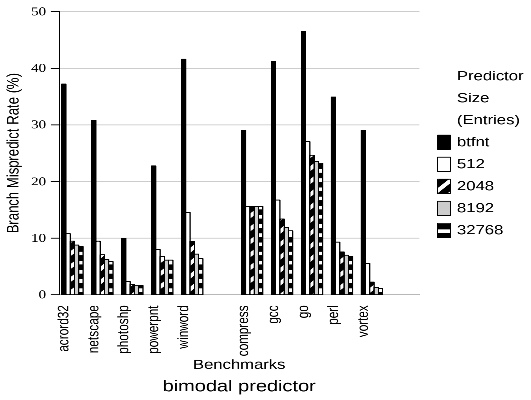
To see if the interactive desktop applications which have to respond to user actions would be less predictable than the SPEC95 batch applications, we measured the predictability of conditional branches for two common types of branch predictors and a static predictor.

The first predictor is a simple bimodal predictor that uses a two-bit saturating counter per branch. The second one is a gshare predictor that uses a global shared history of taken/not-taken decisions (patterns) xor' ed with the PC to obtain the index into a table of two-bit saturating counters [Yeh & Patt 92, McFarling 93]. Essentially, the gshare predictor is able to distinguish different paths through the execution stream of the program and make predictions based on the path rather than just relying on the branch PC. We also looked at the mispredict rate of a static Backward-Taken/Forward-Not-Taken predictor to serve as a point of comparison.

Figure 3 shows the performance of the branch predictors. Clearly, the static predictor does a really poor job of determining the direction of the branches except for *photoshp* which stays in tight loops. Adding a little bit of history improves the predictor performance quite significantly. For the desktop applications, with a 512-entry bimodal predictor, *winword* goes from over a 40% mispredict rate to just over 15%. *Acrord32* and *netscape* also see a drop in their mispredict rate of more than 20%. A similar trend can be seen for the SPEC95 applications.

The bimodal predictor exploits bigger branch predictor tables by capturing more unique branches for the program into distinct slots in the branch table. However, for all our applications (desktop and SPEC95), the bimodal predictor improves very little above a table size of 8192 entries. For example, *powerpnt* has a mispredict rate of about 6% for 8K and 32K entry bimodal predictors. The gshare predictor is able to exploit bigger tables as it distributes different paths to different entries in the table. At smaller sizes, however, the aliasing of multiple paths into common entries increases the misprediction rate of these predictors [Young et al. 95]. As expected, the gshare predictor works better as the tables get bigger. For example, at 512 entries, *winword* has a misprediction rate of 20% for the gshare predictor and only 15% for the bimodal predictor. At 32 K entries, the situation is reversed with the gshare predictor giving a misprediction rate of 4.6% and the bimodal predictor giving a mispredict rate of 6.4%.

With respect to the branch predictors, all the desktop applications tend to perform better than *go*, *gcc*, or *compress*. This implies that the interactive nature of the desktop applications do not translate into unpredictability at the microarchitecture level. This is probably due to the fact that user interaction happens at a coarse level relative to the decisions that are made in the microarchitecture. For example, a single mouse click in *winword* takes a few thousand instructions (and hence a few hundred conditionals) to process. It is the path through these hundreds of conditionals that determine the performance of the branch predictor. Another way of putting it is that the user tells a program “what to do” (e.g., process mouse click at location (x,y)), rather than “how to do it” (e.g., the structure of the mouse click processing code).



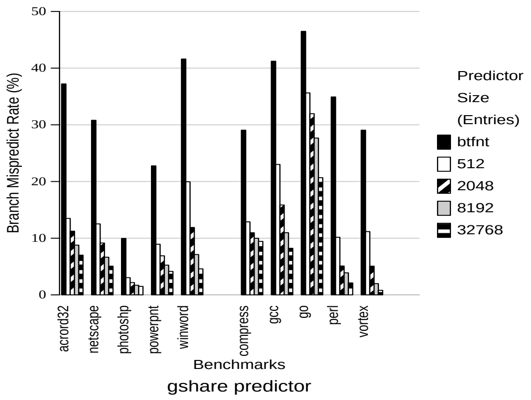


Figure 3. *Branch misprediction rates for bimodal and a global gshare predictors. BTFNT is a static predictor Backward-Taken-Forward-Not-Taken. This predictor serves as a baseline to compare the performance of the different predictors. For both desktop and SPEC95 applications, bimodal predictors work well if predictor sizes are small, while the gshare predictor does better as predictor sizes get bigger.*

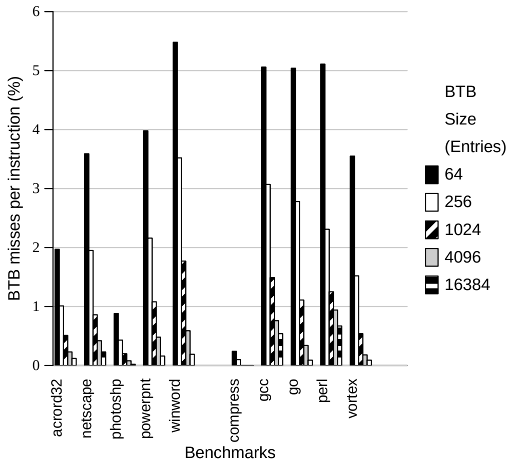


Figure 4. *Branch Target Buffer Miss Rates. The BTB' s are direct mapped. The BTB is used to predict the target of all control flow instructions. The numbers given are in misses per instruction and show all the times when the BTB fails to correctly give the address of the next instruction in the program. For most branches, this is just due to branches not being present in the BTB. However, for indirect branches, this may also be due to the entry in the BTB containing the wrong branch target.*

4.4 Predicting Branch and Call Targets

Pipelined processors typically compute the target address of control flow instructions during the decode phase of the pipeline. This inserts a bubble into the pipeline (a misfetch) since the fetch stage proceeds down the sequential path until the decode stage informs it of the branch target. To handle this situation, many processors employ a small cache called the Branch Target Buffer (BTB) which holds the last target address of a given control flow instruction. All branches and calls cause a misfetch when the branch is not present in the BTB. Indirect branches and calls present an additional challenge: even if the indirect branch or call is in the BTB, the cached branch target may be wrong and cause the fetch unit to proceed down a wrong path.

Figure 4 shows the number that the BTB mispredicted per instruction. For the desktop applications, *acrord32*, *netscape*, and *photoshp* perform well with even a small number of BTB entries while *powerpnt* and *winword* require larger BTB's. For the SPEC95 applications, all but *compress* require a larger number of entries.

For direct calls, direct branches, and conditional branches, BTB misses will recover by the decode stage. For indirect branches and calls, BTB misses do not recover until after the execute phase of the pipeline because the branch target has to be computed or fetched from memory. If the target is contained in a memory location then the memory reference could miss in the cache.

The desktop applications have more indirect calls than the SPEC95 applications (cf., Table 8). Figure 5 shows the misprediction rate per instruction for indirect branches and calls. For *netscape* and *powerpnt*, small BTB's cannot predict the indirect branches because there are not enough entries to hold the branches. This suggests an opportunity to use software techniques such as caching the target addresses of indirect branches in the instruction stream to improve their performance [Ungar et al. 84].

As BTB's get larger, the simple BTB prediction algorithm (predict the last target of the indirect branch or call) becomes the barrier to better BTB performance. For example, most of the mispredicts in *powerpnt* for a 64-entry BTB come from misses in the BTB when it is needed. However, with 16K entries, most of the mispredicts are due to the BTB containing the wrong data for the entry. For the desktop applications, especially *powerpnt*, much of the BTB mispredicts stem from indirect calls. However, for the SPEC95 applications where the misprediction matters

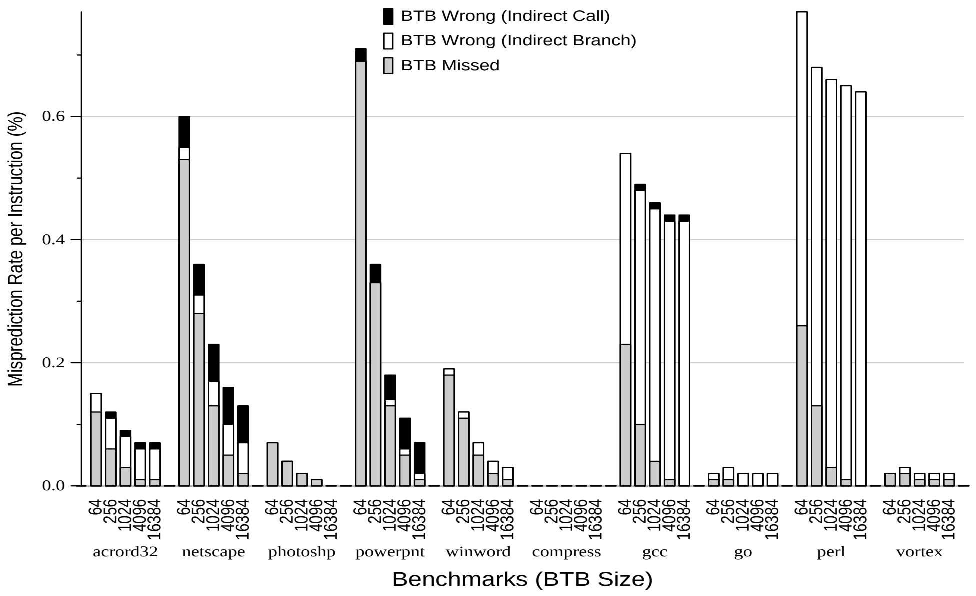


Figure 5. *Branch Target Buffer Mispredict Rates for Indirect Branches. The numbers given are in indirect branches mispredicted per instruction. The graph breaks down the cause of misprediction: either the BTB did not contain the branch address (BTB missed), or the BTB had the wrong entry (BTB wrong). We further classify whether the BTB was wrong for indirect branches or for indirect calls.*

(*gcc* and *perl*), most of the BTB mispredicts stem from indirect branches. For these applications, increasing the size of the branch target is insufficient: the branch prediction algorithm for the BTB needs to become more sophisticated [Driesen & Holzle 97].

5 Summary

We traced and measured several desktop applications running under Windows NT on the x86 processor, and compared these to several SPEC95 applications on the same platform. We found that the desktop applications have larger instruction working set sizes because they access a larger number of DLLs, execute instructions in a greater number of unique functions, and cross DLL boundaries frequently. This leads to worse instruction cache behavior for these applications for moderate sized (32 KB) caches and poor instruction TLB behavior. The data cache behavior of the desktop applications is comparable to that of the SPEC95 applications while the data TLB behavior is slightly better than that of the SPEC95 applications.

As far as instruction set characteristics are concerned, we found that the desktop applications are similar to the SPEC95 applications with one exception: desktop applications execute a greater number of indirect calls. These indirect calls imply the need for larger BTB's or the use of software to improve the performance of indirect branches.

In the introduction, we presented some “obvious” qualitative differences between desktop applications and traditional benchmarks. We now revisit these differences and try to note their impact on performance.

- Desktop applications are interactive and hence may have less predictable behavior. We looked at the branch prediction performance of the desktop applications and they were not worse than the SPEC95 applications. This indicates that the user inserts unpredictability at a level coarser than that which can affect the microarchitecture.

- Desktop applications are graphical. Aside from perhaps making the applications run more code this also does not seem to affect performance. The data cache and data TLB behavior of the desktop applications is no worse than that of the SPEC95 applications.

- Desktop applications are feature rich. On this count, there are measurable differences in application performance. Since the desktop applications run more instructions that tend to be farther apart, the instruction cache and instruction TLB behavior of these applications is worse.

- Desktop applications are multithreaded. For the metrics we measured, this does not result in much of a performance impact. 4 of the 5 desktop applications spent most of their time running in the primary thread. For *powerpnt*, which executed 20% of its instructions in other threads, the cache, TLB, and branch prediction performance were not different from other applications.

- Fifth, desktop applications use DLLs. We found that DLL calls tend to be short, and furthermore that DLL boundaries tend to be crossed often. This crossing probably contributes to the worse TLB behavior of the desktop applications relative to SPEC95.

We believe that the traces gathered for this study will provide a means for the research community to evaltuate architectural ideas against the workloads that run on most people's desktops. The details for obtaining these traces is on our web page: `http://memsys.cs.washington.edu`.

Acknowledgments

This work would not have been possible without the help of Ted Romer, Alec Wolman, Geoff Voelker, Wayne Wong, and Ville Aikas who developed, extended, and fixed bugs on Etch. Dave Patterson, Ori Gershoni, and the anonymous referees provided helpful feedback on earlier drafts of the paper. Finally, Mike Ernst helped us with magic incantations that made LaTeXwork as we wanted.

References

[Calder et al. 94] Calder, B., Grunwald, D., and Zorn, B. Quantifying behavioural differences between C and C++ programs. *Journal of Programming Languages*, 2(4):313–351, December 1994.

[Chen & Leupen 97] Chen, J. B. and Leupen, B. D. D. Improving instruction locality with just-in-time code layout. In *Proceedings of the USENIX Windows NT Workshop*, pages 25–32, August 1997.

[Chen et al. 95] Chen, J. B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., and Smith, M. The measured performance of personal computer operating systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 299–313, 1995.

[Driesen & Holzle 97] Driesen, K. and Holzle, U. Limits of indirect branch prediction. Technical Report TRCS97-10, University of California, Santa Barbara, June 1997.

[Eggers et al. 90] Eggers, S., Keppel, D., Koldinger, E., and Levy, H. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proceedings of ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pages 37–47, 1990.

[Endo et al. 96] Endo, Y., Wang, Z., Chen, J. B., and Seltzer, M. I. Using latency to evaluate interactive system performance. In *Second USENIX Symposium on Operating Systems Design and Implementation*, pages 185–199, October 1996.

[Intel Corporation 97] Intel Corporation. VTune, performance tuning advisor for Intel architecture. software, 1997. version 2.4.

[Larus & Ball 92] Larus, J. R. and Ball, T. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconson-Madison, Madison, WI, March 1992.

[Larus & Schnarr 95] Larus, J. and Schnarr, E. EEL: machine-independent executable editing. In *SIGPLAN Notices*, pages 291–300, June 1995.

[Maynard et al. 94] Maynard, A. M., Donnelly, C., and Olszewski, B. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–155, 1994.

[McFarling 93] McFarling, S. Combining branch predictors. Technical Report TN 36, DEC-WRL, 1993.

[Patterson & Hennessy 96] Patterson, D. A. and Hennessy, J. L. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1996.

[Perl & Sites 96] Perl, S. and Sites, R. Studies of Windows NT performance using dynamic execution traces. In *Second USENIX Symposium on Operating Systems Design and Implementation*, pages 169–183, 1996.

[Pettis & Hansen 90] Pettis, K. and Hansen, R. C. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 16–27, June 1990.

[Rational Software Corporation 96] Rational Software Corporation. Rational Visual Test. software, 1996. version 4.0r, `http://www.rational.com`.

[Romer et al. 97] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., and Bershad, B. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–8, August 1997.

[SPEC 95] SPEC newsletter, September 1995. Information about the SPEC95 benchmarks used in this study is available at `http://www.spec.org/osg/cpu95/CINT95`.

[Srivastava & Eustace 94] Srivastava, A. and Eustace, A. ATOM: a system for building customized program analysis tools. In *Proceedings of the 1994 ACM Symposium on Programming Language Design and Implementation*, pages 196–205, 1994.

[Uhlig et al. 95] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S., and Emer, J. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 345–356. ACM Press, June 22–24 1995.

[Ungar et al. 84] Ungar, D., Blau, R., Foley, P., Samples, D., and Patterson, D. Architecture of SOAR: Smalltalk on a risc. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 188–197, June 1984.

[Woo et al. 95] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[Yeh & Patt 92] Yeh, T.-H. and Patt, Y. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, 1992.

[Young et al. 95] Young, C., Gloy, N., and Smith, M. D. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, June 22–24 1995.