The Measured Performance of Personal Computer Operating Systems

J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières
Antonio Dias, Margo Seltzer, and Michael D. Smith
Division of Applied Sciences
Harvard University

Abstract

This paper presents a comparative study of the performance of three operating systems that run on the personal computer architecture derived from the IBM-PC. The operating systems, Windows for Workgroups, Windows NT, and NetBSD (a freely available variant of the UNIX operating system), cover a broad range of system functionality and user requirements, from a single address space model to full protection with preemptive multi-tasking. Our measurements were enabled by hardware counters in Intel's Pentium processor that permit measurement of a broad range of processor events including instruction counts and on-chip cache miss counts. We used both microbenchmarks, which expose specific differences between the systems, and application workloads, which provide an indication of expected end-to-end performance. Our microbenchmark results show that accessing system functionality is often more expensive in Windows for Workgroups than in the other two systems due to frequent changes in machine mode and the use of system call hooks. When running native applications, Windows NT is more efficient than Windows, but it incurs overhead similar to that of a microkernel since its application interface (the Win32 API) is implemented as a user-level server. Overall, system functionality can be accessed most efficiently in NetBSD; we attribute this to its monolithic structure and to the absence of the complications created by hardware backwards compatibility requirements in the other systems. Measurements of application performance show that although the impact of these differences is significant in terms of instruction counts and other hardware events (often a factor of 2 to 7 difference between the systems), overall performance is sometimes determined by the functionality provided by specific subsystems, such as the graphics subsystem or the file system buffer cache.

This research was sponsored in part by grants from Digital Equipment Corporation, Intel Corporation, Sun Microsystems Laboratories, the Sloan Foundation, and the National Science Foundation. Chen and Seltzer were supported by NSF Career Awards. Smith was supported by an NSF Young Investigator Award.

1. Introduction

While most current operating systems research takes place using a variant of the UNIX operating system, the vast majority of mainstream computing occurs on personal computer (PC) systems, derived from the IBM-PC architecture, running Microsoft Windows. The differences between the OS platforms used in research and mainstream OS products makes some systems research irrelevant with respect to the current needs of the software industry. At the same time, researchers ignore important problems in non-UNIX systems.

The operating system most commonly used on PC platforms is Microsoft Windows. Windows lacks many features that the OS research community takes for granted, most notably preemptive multitasking and protected address spaces. New operating systems for the PC market such as Windows NT and OS/2 incorporate the multitasking support found in modern UNIX operating systems, while also supporting Windows applications. Though the operating systems community is familiar with the benefits of features such as separate protected address spaces, little attention has been given to systems without them. Our goal is to compare a typical research system with typical commodity systems by measuring primitive operation costs, and to use these measurements to understand how system differences affect application performance.

As a first step towards this goal, we present a quantitative comparison of three operating systems, all of which run on the same PC hardware. Microsoft Windows for Workgroups is a version of Windows with integrated networking support. NetBSD is a freely available version of the UNIX operating system. Microsoft Windows NT combines support for Windows applications with operating system features found in UNIX operating systems. There are important similarities in how the systems are used. All are used in personal (i.e. single-user) computing. All are used to load multiple applications into memory and switch between them. All are used with an interactive, window-based interface. These similarities give us a basis for comparison.

Nevertheless, the three systems provide different core functionality. The success of Windows suggests that the functionality it lacks as compared to the other systems is not of primary importance in a personal computer operating system. More specifically:

- Protected address spaces: People learn to work around bugs that cause systems to crash.
- Preemptive multi-tasking: People wait for printers and batch jobs and initiate context switches manually.

- High-level system abstractions: Abstractions like pipes and background jobs are not immediately useful for highly interactive applications.

A key distinction between NetBSD and the Microsoft operating system products is that the Microsoft products must support software originally written for the Intel 8086. This has influenced the structure of Windows NT. For example, in order to support multiple application interfaces (e.g. Win32, OS/2, POSIX), the interfaces are implemented as user-level servers, producing microkernel-like behavior and performance for NT.

This study explores how structural features affect performance. We use hardware counters in the Pentium microprocessor [19, 26] to gather a variety of system metrics and use these metrics to quantify performance. We present results from two sets of experiments, one using microbenchmarks and another using application workloads. The general strategy for our analysis is:

- Gather statistics for the microbenchmarks.
- Explain the measurements for the microbenchmarks in so far as possible from available documentation.
- Gather statistics for the applications.
- Explain application performance based on microbenchmark results and available documentation.

In our comparison we were obliged to make compromises on certain factors beyond our control. For example, we could not compile the three operating systems using the same compiler since we do not have access to source code for Windows and Windows NT. Instead we measured systems that were compiled in the same way as the systems in common use. There were also occasions where our lack of access to source code for Windows or Windows NT prevented us from answering detailed questions about their internals. Nonetheless, analysis of data from the measurements yields revealing and frequently surprising results. We found that:

- For 16-bit Windows system calls, frequent changes in machine mode, required for backward compatibility, increase system overhead by a factor of seven.
- The microkernel-like structure of Windows NT adds significantly to the cost of accessing system functionality.
- The integrated buffer cache and device-dependent graphics in Windows NT give it significant performance advantages over NetBSD.

The next section provides background on the three operating systems and the hardware on which they run. In Section 3 we discuss issues specific to our experiments, including configuration details for the machine used, information on the hardware counters and how we accessed them, and a description of the microbenchmarks and application workloads. In Section 4 and Section 5 we describe our experiments and explain the results we obtained. In Section 6 we discuss the ramifications of our work and note some issues that warrant further investigation.

2. Background

This section provides background on the systems that we measured and machines on which they run. Table 1 presents some general data on the three operating systems. Due to space constraints, we cannot give all relevant details about the three systems. References are provided for readers requiring more complete technical documentation on the systems.

system	principal components	executable size (K bytes)
Windows for Workgroups 3.11	MS-DOS	77
	KRNL386.EXE - control for segmented memory	74
	WIN386.EXE - the Virtual Machine Manager (VMM) and other VxDs. The VMM handles task switching, paging, and other low-level services	564
	USER.EXE - the windowing system	257
	GDI.EXE - graphics engine module	215
Windows NT Version 3.5, Build 807	Executive - interrupt handling, VM, IPC, file system, network	735
	Win32 subsystem - Win32 user interface, windowing system	4096
NetBSD 1.0	/netbsd — monolithic kernel	592
	XF86_S3 - X11 Windowing System Server 3.1.1	2344

Table 1. Components of the three systems. Sizes for Windows NT are from dynamic estimates using the *PView* utility.

2.1 Windows

Windows is the *de facto* standard operating environment for PCs. Windows is sometimes referred to as an *MS-DOS extender* and must be used in conjunction with MS-DOS, which provides a layer of basic system functionality. MS-DOS also defines a system interface that Windows must preserve. As Windows and MS-DOS are usually used together, we will use the term “Windows” to refer to the composition of Windows running on top of MS-DOS.

Current Windows applications typically execute in 16-bit protected or “Win16” mode, ignoring the top 16 bits of 32-bit registers on the Intel 80386. Win16 applications operate in a segmented address space, with a maximum segment size of 64K bytes imposed by the 16 bit registers used for segment offsets. Windows and all Win16 applications share this segmented address space. Context switches between applications occur when an application voluntarily yields control. Windows does not support protected address spaces or involuntary context switches. We estimate that about half of Windows is written in hand-coded assembler.

The Windows system software is composed of four components that are loaded separately: KRNL386.EXE, USER.EXE, WIN386.EXE, and GDI.EXE. Third-party developers who wish to extend Windows can do so by creating dynamically loaded libraries (DLLs) and loadable device drivers (VxDs), which are loaded into the Windows shared address space. DLLs and VxDs in Windows make it possible to incorporate new system functionality into the system and are used extensively in Windows software.

We selected Windows for Workgroups over Windows because it was the most recent Windows release available and because its network support is similar to that provided by Windows NT and NetBSD. Only the network microbenchmark and the Web server use the network. All Windows experiments used the FAT filesystem. We used the default 16-bit MS-DOS implementation of FAT for all experiments except for the filesystem benchmarks, where we also tested the 32-bit driver option. We used Smartdrv (with delayed write enabled) in conjunction with the MS-DOS FAT implementation to provide caching for disk reads [31]. The 32-bit FAT driver implements caching internally, eliminating the need for Smartdrv.

Although we did not have source code for Windows, we did have a Windows debugger that made it possible to single-step through interrupt handlers and other parts of the system,[1] helping us answer questions about Windows control structure. There are many books which provide a more comprehensive technical description of Windows [6, 29, 35, 38]

2.2 Windows NT

Unlike Windows, Windows NT supports protected address spaces, preemptive multi-tasking and multiple APIs, including the Win32 API [41] as well as the MS-DOS and Windows APIs. Windows NT implements system APIs using user-level server processes called *protected subsystems*. As such, many aspects of Windows NT performance and behavior are consistent with earlier results for microkernels [7].

1. We used the SoftIce debugger from NuMega Technologies Inc.

The kernel in Windows NT is called the NT Executive. It provides interrupt and exception handling, virtual memory, interprocess communication (IPC), file systems, and network access. The NT Executive implements a set of native services which are used by the protected subsystems. Services exported by the NT Executive are intended for use by protected subsystems, and are not meant to be accessed directly by applications [11].

The Win32 API is the preferred API for Windows NT applications. It provides a flat 4G byte virtual address space. The upper half is reserved for the NT Executive, and the lower half is the user address space. Unlike Windows, the user address space is private and is not shared between Win32 processes. The Win32 protected subsystem implements the display interface, the console input device, the window manager, and also maintains input queues for each Win32 client. Other subsystems (such as the POSIX subsystem) act as Win32 clients when accessing the display or console input device.

Windows NT includes a Local Procedure Call (LPC) facility. LPC supports message delivery via shared memory that avoids copies when large amounts of data must be transferred [11]. To reduce communications overhead, the NT Executive can cache information in the client DLL and within the executive. Additionally, multiple system call requests can sometimes be batched in a single message.

The I/O system is composed of an I/O manager and device drivers that can be loaded and unloaded dynamically as required. NT device drivers can be composed hierarchically to implement an I/O abstraction. For example, file systems such as FAT, HPFS, and NTFS are implemented as device drivers. The multiple file system implementations do not have direct access to the hardware. Instead, they use an intermediate driver that accesses the hardware directly. Intermediate drivers can also implement additional functionality (such as mirroring, encryption, compression) when installed at an appropriate level in the stack of drivers.

With the exception of the NTFS tests in Section 4.5 all Windows NT experiments used the FAT file system. Unlike Windows, disk caching is an integral part of Windows NT, so no user-level caching program was required. There are many books providing a more comprehensive technical description of Windows NT [11, 12, 30].

2.3 NetBSD

NetBSD is a descendent of Berkeley (BSD) UNIX, and was selected as a representative example of modern UNIX systems. It is a monolithic kernel, with the system call interface implemented directly by the operating system kernel. Each NetBSD process runs in its own 32-bit protected address space, with the lower half available to the user process, and the upper half reserved for the system. The X11 Windowing System runs outside the operating system kernel as a collection of user processes. This is in contrast to Windows and Windows NT, where the windowing system is an integral part of the operating system. There are many excellent technical references for UNIX systems [3, 22].

Our NetBSD system used the Berkeley Fast File System (FFS) [27]. For our experiments, all NetBSD device drivers were statically loaded. Executables were statically linked except as noted. Our NetBSD kernel was compiled from the 1.0 distribution at the default optimization level (-O6) with gcc version 2.4.5.[1]

The selection of our experimental UNIX platform was a difficult choice. We selected NetBSD because source code is publicly available, and because it shares a common heritage and some source code with many current commercial UNIX systems. Although a commercial UNIX system might have permitted a comparison using UNIX applications that more closely resembled typical Windows workloads, all potential candidates had serious shortcomings,[2] so we chose a freely available system that we knew would support the device drivers required for our study.

2.4 Related Work

Although prior work has considered the performance impact of structure on UNIX operating systems [7], very little attention has been given to the behavior of commodity operating systems. In the commodity computing world, performance measurement studies are typically based on the assumption that the Windows operating system will be used, and concentrate primarily on the performance for variations in computer hardware [43]. Popular computing magazines regularly publish articles that compare current operating system offerings for PCs, but these evaluations usually focus on functionality. They tend not to explore performance issues [15, 42], and when they do the analysis is superficial, presenting end-to-end benchmark results with little attempt to explain or understand performance differ-

1. Our NetBSD system was configured with clustering enabled, rotdelay = 0 and maxcontig = 8. Coalescing I/O operations was enabled. Aside from the addition of a device driver to access the Pentium counters, one change was made locally to the system: a trivial (3 line) change was required to the initialization code of the NCR PCI SCSI device driver to support the NCR 815 controller.

2. For example, Microsoft Word runs under SCO UNIX, but only using XENIX emulation mode.

ences [24]. In addition, these articles frequently anticipate future OS offerings, going to press before the systems that they feature are ready for performance evaluation [24, 42]. We did find one study that compared 32-bit operating systems including several commercial UNIX products; however, the article focused on qualitative differences in functionality with criteria such as “Graphics and Multimedia,” “Business Productivity,” and “DOS and Windows emulation” [23]. Another study compared “Network Operating Systems,” but the only performance results presented were for local and remote file transfers [17].

Some attention from the research community has gone to supporting MS-DOS interfaces under a UNIX operating system. Forin & Malan [16] compare the FAT and Berkeley FFS file systems implemented for a Mach 3.0 system. Another article described a server for the MS-DOS API on top of Mach 3.0 [37].

3. Methodology

In this section, we document the hardware and software details of our experiments. We start by describing the hardware platform on which this study is based. We next describe the particular Pentium counters that we use in our study, and the software that we used to access them. We go on to describe our microbenchmarks and application workloads, and then close with a discussion of the metrics that we use to compare the three systems.

3.1 PC Hardware

Our experimental machine was based on the Intel Premiere II motherboard, with a 90-MHz Intel Pentium processor, a 256K byte direct-mapped second-level cache, and 32M bytes of main memory. The Pentium processor includes split first-level instruction and data caches. The first-level caches are 8K bytes each and two-way set-associative, with 32 byte lines and a pseudo-LRU replacement algorithm. The Pentium fetches blocks of instructions from the instruction cache as required and places them into one of four large prefetch buffers. To deal with the Pentium’s variable length instructions, the instruction cache can return anywhere from 17 to 32 bytes, as many as are required to fill the prefetch buffer. Prefetch buffer logic handles instruction identification and alignment. The processor can issue two “simple” instructions during each clock cycle. Dual datapath pipelines enable both instructions to issue memory operations to the first-level data cache. More complex x86 instructions (e.g. a string copy instruction) issue and execute alone, but use both datapath pipelines. The Pentium on-chip data cache is write no-allocate; writes that miss in the cache do not affect cache contents.

The Pentium processor also includes separate instruction and data translation lookaside buffers (TLBs) and a hardware TLB fault handler. Both the instruction and data TLBs are 4-way set associative with pseudo-LRU replacement. The instruction TLB contains 32 entries. The data TLB contains 64+8 entries and is dual-ported to support translation for two data memory references per cycle. The 64-entry portion of the data TLB stores translations for the 4K byte pages as specified by the 80386 architecture [19]. The 8-entry portion maps 4M byte pages. The larger page size is used for graphics frame buffers and some operating system segments, to avoid flushing the 4K byte-page portion of the data TLB.

The Intel Premiere II motherboard has a PCI bus, an ISA bus, and a BIOS ROM (AMIBIOS 1.00.10.AX1). We used a PCI SCSI-II controller based on the NCR 815 chip to control two Seagate ST32550N Barracuda hard disks. One disk was allocated to NetBSD and the other was shared by Windows and Windows NT. Our machine was equipped with a SMC-8013W ISA Ethernet controller card and a Diamond Stealth 64D video adaptor card. The video card has 2M bytes of DRAM memory and a Vision864 graphics chip.

3.2 The Pentium Counters

We made our measurements using event counters implemented in the Intel Pentium processor. Although the counters were not documented in the initial public specification for the Pentium chip, they were reported in popular magazines [26], and were included in the public documentation from Intel shortly before the publication of this work [19]. We relied primarily on unofficial report when preparing this study, as unencumbered documentation was not available from Intel. We have verified that our usage of the counters is consistent with the documentation Intel has recently published.

The Intel Pentium processor includes one 64-bit cycle counter and two 64-bit, software-configurable event counters. The Pentium provides several privileged instructions for configuring these counters. Each counter can be configured to count one of a number of different hardware events. In addition, the counters can be configured to count events in ring 0 (kernel mode), events in rings 1 through 3 (user mode), or events in all rings. This feature permits some separation of user and system activity; however such counts cannot be compared directly across the three operating systems because each of the systems implements a different amount of functionality in kernel mode. In Windows, the Virtual Machine Manager (VMM) and virtual device drivers run in kernel mode, but the rest of the system (including MS-DOS and the BIOS) runs in user mode. In Windows NT, the NT Executive runs in kernel mode, with applications and the Win32 subsystem running in user mode. In NetBSD, only the kernel runs in kernel mode. This partitioning must be considered when comparing kernel and user mode events across the different systems. Because of the barriers to comparison across systems, all counts reported in this paper are the total for all protection rings.

Table 2 lists the subset of the Pentium event counts that we report for our experiments. Although the cycle counter continues to increment when the machine is halted (e.g. during idle time for Windows NT), no other event counts are incremented.

Index	Name	Comments
0x00	Data Reads	Count is independent of data size. Misaligned reads count twice.
0x01	Data Writes	Misaligned writes count once.
0x02	Data TLB misses	
0x03	Data read cache misses	
0x04	Data write cache misses	
0x0d	Code TLB misses	
0x0e	Code cache misses	
0x0f	Segment register loads	
0x16	Instructions executed	Instructions using the REP prefix count as a single instruction.
0x19	Write stall cycles	Counts pipeline stall cycles due to write backup.
0x27	Hardware interrupts	

Table 2. Pentium Event Counters. Only counters used in this study are listed.

In addition to using the Pentium cycle counter to measure the run times of our benchmarks and applications, we also used them to monitor our experiments. We recorded the cycle count for every experiment, and used it to ensure that our experimental runs were repeatable and that our measurements were not unexpectedly affected by background activity.

3.3 Device Drivers

The Pentium counters are accessed and controlled using special instructions that are available in kernel mode (ring 0) only. We augmented each of the three operating systems with mechanisms to support user level counter access and control. To accomplish this under Windows, we used a virtual device driver, commonly known as a VxD [36]. A VxD is a dynamically-loadable module that runs in protection ring 0, in 32-bit mode. VxDs are commonly used to multiplex physical devices for virtual machines. Windows uses a dozen or so VxDs; additional VxDs can be specified in the system configuration files that are processed during Windows initialization. VxDs are commonly activated by hardware interrupts or system calls, but they can also publish an entry point, thereby permitting user-level programs to call them directly. We used the latter method to access our Pentium counter VxD.

For NetBSD, we accessed the Pentium counters using device drivers and named files in /dev. We used four separate named devices: one each for system and user event counts, one for the cycle counter, and a fourth that implemented a software-based count of idle loop iterations. This gave us a measure of the amount of time a given workload spent waiting for disk requests. The event counters were configured using ioctl system calls. Further ioctl calls zero or freeze all counters in a single operation.

Windows NT provides an elaborate API to support dynamically-loadable device drivers. Using this API we implemented a device driver that is dynamically loaded into the NT Executive to access the Pentium counters. This device driver allows user programs to manipulate the counters by using reads and writes of a device special file, similar to the driver used with NetBSD.

3.4 Microbenchmarks

We used a suite of microbenchmarks to measure and compare specific aspects of system functionality. Most of our microbenchmarks are based on the *lmbench* portable system measurement suite [28], and all of them borrow from the lmbench methodology. They include:

- *Null* — counter access latency. Measures time to access our counter control device. A starting point for understanding more complex behavior. This benchmark is not from the lmbench suite.
- *Syscall* — minimum system call latency. Time to invoke functionality implemented in the operating system.
- *Exec* — latency to load and run a trivial program. We tested Exec with both static and dynamically loaded libraries.
- *Memory access time* — access time for references spanning arrays of various sizes. This measures the impact of segmented memory in Windows and of page-mapping policy in Windows NT and NetBSD.
- A suite of file system performance tests.
- *Bitblt* — a graphics benchmark to test bitblt performance. This benchmark is not from the lmbench suite.
- *Netbw* — a network throughput test.

The insights gained from understanding the microbenchmark results provide a basis for analyzing the behavior of application workloads. More detailed descriptions of the individual microbenchmarks are given with the experimental results in Section 4.

The benchmarks were compiled for each of the three systems with the highest available optimization level. Windows does not support static linking for all libraries. All benchmarks on NetBSD and Windows NT were linked statically, except for the static vs. dynamic linking comparison of the Exec benchmark. The impact of compiler optimization on the microbenchmarks is small — most of the measured execution time is system time and very little time is spent in user code for the microbenchmarks. The microbenchmarks were compiled for NetBSD with gcc version 2.6.3 using optimization level -O6. We used Visual C++ 1.5 for Windows and Visual C++ 2.0 for Windows NT, with optimization level /O2. No Pentium-specific optimizations were applied for any of the experimental workloads or systems.

3.5 Application Workloads

For application workloads we restricted our study to software that ran on all three systems. Regrettably, this excludes the "shrink-wrapped" software that makes up the bulk of Windows computation. Our workloads may tend to favor NetBSD because all of them were originally developed for the UNIX API. Despite this bias, they do provide grounds for comparison, which Windows software would not.

Wish is a command interpreter for the Tcl language that provides windowing support using the Tk toolkit [34]. Our goal in using this workload was to model behavior that might be a part of a graphical user interface. Our Wish benchmark was based on the "widget" demo included in the standard Tk release. We used the "source" Tcl command to load the widget file and "invoke" commands to exercise various widgets on the screen. This workload was CPU intensive, making extensive use of the windowing system with little disk activity. Wish was compiled with standard optimizations (-O) on all three systems.

Ghostscript is a publicly available Postscript previewer [2]. For our experiment, we used Ghostscript to display a 13 page, 372K byte conference paper. This workload was compute intensive and made significant use of the windowing system and display. Relatively little time went to file access. Ghostscript was run with the same screen resolution, view size, magnification, and screen refresh rate on all three systems, and was compiled at the highest optimization level. Ghostscript was dynamically linked on all three systems.

Our third application, a *World Wide Web Server* [4], uses the network and the file system. The NetBSD HTTP server used was version 1.3R of the NCSA HTTP daemon, compiled with gcc 2.6.3 at optimization level -O2. The Windows NT server is distributed as an executable from the European Microsoft Windows NT Academic Centre at the University of Edinburgh [1], and is the server used by Microsoft (host www.microsoft.com). The Windows HTTP server we used is from Alisa Corporation and is based on the NCSA Mosaic code base.

We used a separate counter-control program to start, stop, and record counter values. In this way we avoided including server start-up and shutdown overhead in our measurements. The Web experiments ran long enough so that the activity generated by the counter control program was not significant. To improve the reproducibility of our results, we defragmented our FAT disk before loading the Web server tree. Because FFS has good aging properties [40], FFS defragmentation was not required.

3.6 Methodology and Metrics

When running our experiments we attempted to minimize background activity in the operating system. For NetBSD, this meant that we ran our experiments in single-user mode. Experiments in multi-user mode yielded similar results but with larger standard deviations. We disconnected the network during all tests except the Web server and network bandwidth experiments.

Idle time is implemented differently on each of the three operating systems, and we compensate for these differences in interpreting the counter values. Windows NT executes a HALT instruction when the system is idle, and although the cycle counter continues to increment, no instructions or other events occur. NetBSD, on the other hand, uses an idle loop. We adjust our NetBSD counts for idle loop activity so that they are comparable with the Windows NT counts. Windows also halts the machine during idle time, however the idle halt is not used during disk waits; I/O requests cause Windows to busy wait. This makes it harder to interpret counts for Windows when significant disk activity occurs.

We use several different metrics in our comparison of the different operating systems executing on the Pentium. The most important measure is the time required by the computer system to perform a specific task, and so we frequently use *cycle counts* to compare the total latency for comparable computations on the three different systems. We also report *instruction counts* for comparable computations. The instruction count gives a measure of the total work involved in a computation for RISC processors, but cannot always be readily interpreted for the Pentium. With its complex multi-cycle instructions, there are many opportunities for system programmers to implement equivalent computations that cause vastly different instruction counts. A simple example is a block copy operation, which can be implemented as a loop containing load, store, and branch operations, or as a single instruction (REP MOVSB). In the first case, the instruction count is proportional to the number of data accesses. In the second case the instruction count is one. We found that NetBSD and Windows NT use the repeat prefix in their respective block copy routines, and we found that Windows uses the repeat prefix extensively in hand-coded assembler. In short, instruction counts cannot be immediately compared when evaluating two different operating systems.

We also report the total number of data read references and data write references for each experiment. Again, these numbers can vary depending on how a given operation was implemented, although the variability is much lower than that of the instruction count. We also use the Pentium counters to compare cache and TLB performance for the three systems. These metrics are reported as *misses per reference*.

Ideally, we would have liked to present the percentage of total cycles that result from (as an example) data cache read stalls. The Pentium does provide stall cycle counts, but they cannot be interpreted in this way. The Pentium with its ability to issue multiple instructions per cycle and overlap the processing of multiple events made it impossible for us to assign cycles to specific events.

The interpretation of data cache miss rates is further complicated by the write no-allocate policy used by the Pentium data cache, because write misses do not cause a cache fill. A side effect of this policy is that consecutive writes to the same cache line can cause multiple write misses. This magnifies the impact of writes relative to total data activity. Although the misses per reference metric we use to report data cache activity gives a measure of the relative behavior of the three systems in the data cache, it should not be used to estimate data stalls cycles and should not be compared with instruction cache miss rates. For readers interested in the precise count of read and write miss events that occurred during our experiments, this data is available in a Harvard technical report [8].

The above metrics are used throughout the experiments. Several experiments also use data from additional counters which will be described as they are introduced.

4. Microbenchmark Results

4.1 The Null Benchmark

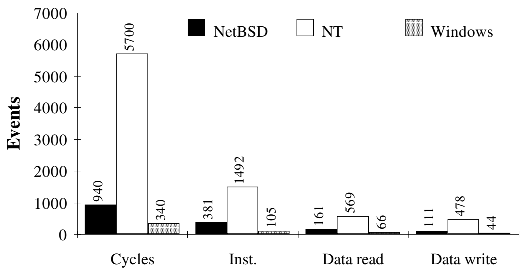
The Null benchmark measures the overhead for invoking our device that controls the Pentium counters. Pseudocode for this benchmark is as follows:

```
for (each counter)
    for (50 iterations) {
        start();
        stop();
    }
```

The function `start()` zeros the current counter, and `stop()` stores the counter value in an in-memory log. We used the results from this benchmark to determine correction factors for removing counter maintenance overhead from the other experiments.

Figure 1 shows baseline results for the Null benchmark on the three systems. Comparison of the instruction counts shows that Windows requires far fewer instructions to access the counters than do the other systems. This is because a user program can call the Windows protected mode driver directly, with no significant operating system activity. In NetBSD and Windows NT, the call to the counter driver must pass through a layer of file system functionality before reaching the actual device driver. Windows NT requires many more instructions to access the counters relative to the other systems. This overhead is due to the multiple protection domain crossings required to access the counter device in Windows NT. From the Windows NT documentation [11], we have inferred that six protection boundary crossings are required for this operation: from the application to the Executive; from the Executive to the Win32 subsystem; from the Win32 subsystem back into the executive; and then a return through each of these domains. In comparison, NetBSD requires two protection boundary crossings. Windows does not require protection boundary crossings, although it does require two changes of machine mode.

The results of the Null benchmark illustrate several other distinctions between the operating systems that occur throughout our experiments. The first is a higher cycles per instruction ratio (CPI) for Windows (3.2) than for NetBSD (2.5). The higher CPI in Windows is due in part to multi-cycle instructions used to implement calls between different Windows subsystems. Although the same instructions are used in NetBSD, they have lower latency because a change of machine mode is not needed. As an example, the INT and IRET instructions that are used to call and return



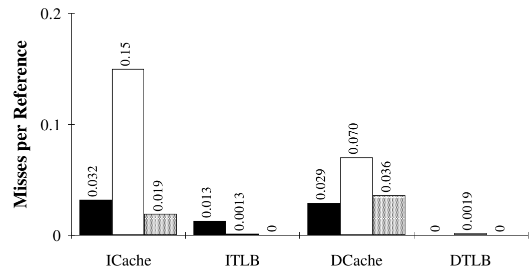


Figure 1. The Null Benchmark. These graphs show measures of system activity required to access third party device drivers. They illustrate the activity required to start and stop the Pentium counters.

from a VxD subroutine are accompanied by changes in the machine mode under Windows, such that the latency of the INT and IRET instructions are about 100 cycles [19]. In Windows NT and NetBSD, access to a protected device driver does not require a change in machine mode, and this decreases the latencies of the INT and IRET instructions to 48 and 27 cycles, respectively.

The counters reveal that the elevated cycle counts under Windows NT are also due in part to a high instruction cache miss rate relative to the other systems. Using 10 cycles as a lower bound for the cache miss penalty, about 40% of the execution time is due to instruction cache misses.

Overall, our Null benchmark accurately reflects the overhead for accessing third-party kernel mode functionality in the three systems. The results show that Windows applications that directly access hardware devices get a substantial performance advantage compared to the other two systems. However, the results are not indicative of the time required to access system functionality in general. The Syscall benchmark presented in the next section gives a better indication of the cost of accessing functionality implemented in the system.

4.2 The Syscall Benchmark

The Syscall benchmark measures the minimum time required to invoke a system call. Pseudocode for our microbenchmark is as follows:

```
for (each counter)
    for (50 iterations) {
        start();
        system call
        stop();
    }
```

To test each system, we selected a call that always went to the system (i.e. could not be batched or cached). For Windows NT and NetBSD we used dup(). The choice in Windows was less straightforward. In Windows, the concept

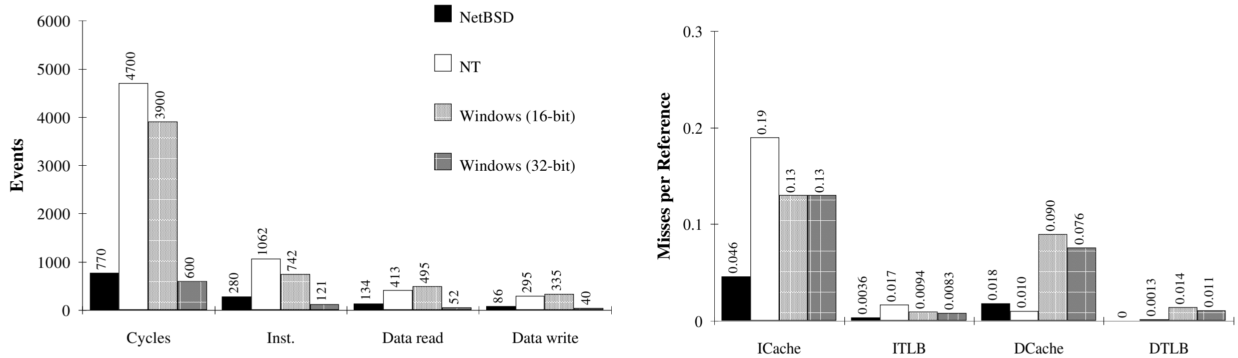


Figure 2. The Syscall Benchmark. These graphs show measures of system activity required to access functionality implemented in the operating system. Figures are corrected for the cost of accessing the Pentium counters.

of "system call" has been complicated by the evolution of the system. Although early Windows implementations ran entirely in 16-bit CPU modes, significant functionality in more recent releases runs in 32-bit mode [21, 39]. We present results for two Windows system calls, "get extended error information" (int 21 function 59), which runs in 16-bit mode, and "get interrupt vector" (int 21 function 35), which runs in 32-bit mode. The comparison of system services implemented in 16-bit vs. 32-bit mode allows us to evaluate the impact of the shift towards a 32-bit Windows implementation.

Figure 2 shows results for the Syscall benchmark, corrected for counter manipulation overhead. The overhead for the dup() system call in Windows NT and NetBSD is similar to the overhead required to start and stop the counters in the Null benchmark. Also, observations from the Null benchmark with respect to the instruction cache continue to hold (Figure 1 and Figure 2). The "Get interrupt vector" system call in Windows is the most efficient, with the overhead for dup() in NetBSD slightly higher.

Comparing the two Windows system calls, the overhead for the 16-bit call is a factor of six higher. The additional overhead can be attributed to two causes. The first is changes in machine mode. The 32-bit Windows call requires two changes of machine mode, from 16-bit protected mode (user code) to 32-bit protected mode (the kernel) and back again. In contrast, a 16-bit Windows call requires at least six changes between three CPU modes: 16-bit protected mode, 32-bit protected mode, and virtual 8086 mode [32]. Additional machine mode changes are required when the system call uses privileged instructions or executes from the BIOS ROM, as is the case with system calls for file access.[1] Each change of CPU mode requires a single instruction with latency of about 100 cycles to save and restore hardware state, plus additional overhead to maintain software state.

Another control structure that contributes to the cost of the 16-bit Windows call is the use of "hooks," a mechanism by which a program can extend MS-DOS functionality by intercepting system calls. A system call hook requires modifying the system call vector entry for int-21; in this way, all interested programs filter system calls as they arrive to see if the requested functionality is something they implement. In our system, three software systems used hooks: SMARTDRV — providing MS-DOS disk caching, MSCDEX — for CD-ROM support, and IFSHELP — which allows Windows to over-ride certain MS-DOS system services. By repeating our test after removing these programs, we determined that these hooks are responsible for about 1000 cycles in the 16-bit Windows system call.

Our Syscall benchmark shows that NetBSD provides access to all system functionality with low overhead, and the overhead is similar for Windows functionality implemented in 32-bit.mode. Both Windows NT and the 16-bit Windows calls have higher cost, due to switching overhead between the various components of the operating systems. As this test exercised very simple system calls, most of the activity corresponds to system call invocation overhead. In a realistic situation, system call invocation is a smaller percentage of the latency required to service a system request.

4.3 The Exec Benchmark

The Exec benchmark measures how quickly the system can load and run a trivial test program. Pseudo code for our benchmark is as follows:

```
for each counter
    for (50 iterations) {
        start();
        start program in new process
        wait for process to exit
        stop();
    }
```

Data reported in this section is for the cost of a single program execution averaged over 50 invocations. The activity required to execute a program is slightly different for each system. NetBSD uses the vfork() and exec() system calls. The vfork() avoids the overhead of duplicating the parent address space. In Windows NT, the CreateProcess() system

1. The variety of protection mode transitions required by various Windows calls is too complex to provide a comprehensive explanation in this paper. The technical documentation for Windows provides a more detailed description of this behavior.[32].

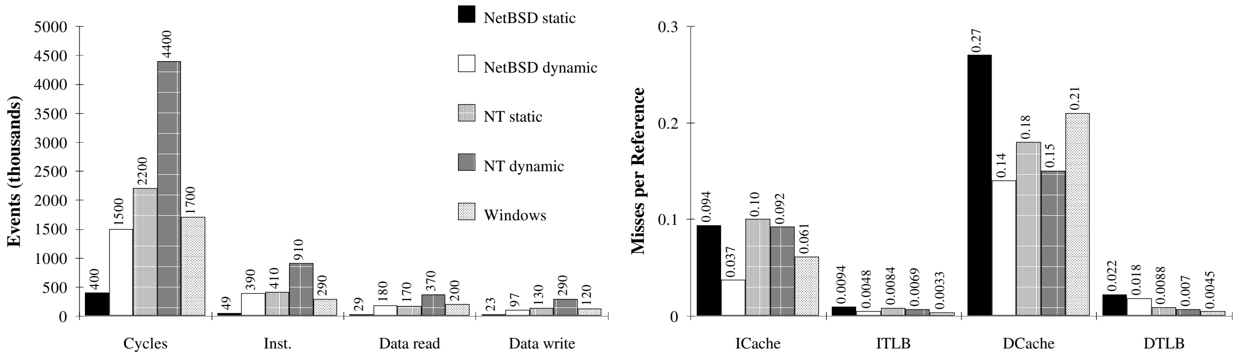


Figure 3. The Exec Benchmark. This workload measures the activity required to load and execute a trivial program. The horizontal bars for the DCache columns indicate the relative contribution of reads and writes to the DCache miss rate.

call takes an executable file as one of its parameters, and creates a new address space in which the executable is subsequently run. For Windows, Win16 applications run in a single shared address space so no address space creation occurs. WinExec() takes an executable file along with other parameters for running a program, loads the specified program into the shared address space, and begins execution at its entry point.

We tested both static and dynamic linking for NetBSD and Windows NT. Windows does not provide a comparable linking option. Windows executables are linked statically with libraries that provide functionality comparable to that of UNIX libraries. References to system APIs are resolved optimistically, under the assumption that DLLs (dynamically loaded libraries) will be available at a preferred location in the shared address space. If, when the program is run, a DLL cannot be loaded in its preferred location or has been loaded elsewhere, references to library functions are updated with the correct address. A similar mechanism is used with dynamically loaded Windows NT executables.

Our test program for Windows was loaded as a Windows executable using Visual C++ Version 1.5 default settings (medium memory model, includes prologue for protected mode application functions). On Windows NT we used a Win32 binary running in console mode.

Figure 3 shows that the NetBSD static case gives the best performance. Address space creation and program loading are relatively simple operations in NetBSD and have been optimized over many generations of UNIX systems. In contrast, the Windows NT static case requires a factor of five more cycles than NetBSD. Part of this overhead is due to how functionality is split between the NT Executive and the Win32 subsystem. The need to support multiple APIs through protected subsystems make it necessary for the NT Executive interface to be simple and general. In the case of CreateProcess(), the Win32 subsystem uses two distinct NT Executive calls, one to create an address space, and a second to create a thread to execute in that address space [11]. Our measurements for the static case are consistent with prior results for systems where the system API is implemented in a user process [7]. For Windows, exec overhead is higher than NetBSD in both static and dynamic cases.

Windows NT requires six times more data reads and five times more data writes than NetBSD. Table 3 shows that the sizes of the executables for the test programs are not responsible for this difference. We suspect that extra copies of the executable file between the NT Executive and the Win32 subsystem and the copies required for address space duplication contribute to this overhead. A further possibility is that mechanisms in NT are not optimized for the case of static linking, as dynamic linking is the most common case.

	NetBSD	Windows NT	Windows
static	9	10	3
dynamic	14	3	not supported

Table 3. Size of child program for the Exec Benchmark, in kilobytes. The anomaly in the sizes for NetBSD is due to extra code included to support dynamic linking. This extra code is insignificant relative to the size of a non-trivial program.

Dynamic linking doubles the exec cost for NetBSD and Windows NT. This is due to the increased cost of linking at execution time. The overhead for Windows NT is by far the highest of any of the systems. Although this may not be important for programs that run for a long time, it would contribute to the delay in starting applications.

The results for the Exec benchmark serve as an example of the cost of invoking significant system functionality. They also serve to demonstrate how the structural differences between Windows NT and NetBSD contribute to differences in performance. Although the Exec benchmark is interesting for understanding system structure, its relevance to performance is limited, particularly for Windows and Windows NT, where applications typically run for a long time once they are started, and where functionality is frequently implemented as a subroutine or library call rather than a separate program (for example, the dir command in the MS-DOS command interpreter).

4.4 Memory Access Time

We used the memory read microbenchmark from *lmbench* to measure average memory access time for repeated references to integer arrays of various sizes, using a stride of 128 bytes. Pseudocode for the benchmark is as follows:

```
void *a[SIZE], *p;
for (i = 0; i < SIZE; i++)
    a[i] = (void *)&a[(i+32) % SIZE]
p = a[0];
```

```
start();
for (1,000,000 iterations)
    p = *p;
stop();
```

Figure 4 shows experimental results for array sizes ranging from 512 bytes to 1M byte. The curves show that differences in memory access time between the three systems can be substantial. Comparing Windows NT and NetBSD, both systems get similar average access times below 8K bytes, where the test array fits in the on-chip cache. Above 8K bytes, access times are determined by how pages are mapped into the 256K byte board cache. Windows NT uses a deterministic page mapping policy [7] that gives similar performance for arrays up to the size of the second level cache and a smooth performance degradation for arrays from 256K bytes up to 512K bytes (twice the size of the second level cache). The deterministic page mapping policy guarantees that no two pages will conflict for arrays less than 256K bytes, and that pages will conflict in pairs for arrays from 256 - 512K bytes. At the 512K byte boundary, each page conflicts with at least one other page; thus the board cache is of no benefit as the array is traversed sequentially.

NetBSD uses the trivial page mapping policy of simply taking the next page off the free list. Since the free list is unordered, this results in a non-deterministic page mapping policy. The curve for NetBSD shows that the NetBSD policy can do a poor job of avoiding cache conflicts. The randomness in the NetBSD policy causes unnecessary conflicts for arrays between 8K bytes and 512K bytes, giving NetBSD worse performance relative to Windows NT up to a crossover point slightly before 512K bytes. Above 512K bytes, the NetBSD policy randomly avoids conflicts, sometimes providing slightly better performance than Windows NT.

Windows NT uses a bin hopping algorithm that tries to allocate physical pages so as to avoid conflicts in the cache [20]. Additionally, its file system buffer cache is integrated with the VM system, allowing pages to be used by whichever subsystem appears to need them the most. We found that this integration can sometimes interfere with the page allocation policy, as the page allocator will sometimes ignore a cache placement suggestion rather than stealing a page from the disk-cache manager. The NT measurements for Figure 4 were obtained from a recently booted system, so that few pages were in use by the cache manager. Other tests showed that an NT system with more pages allocated to the disk cache can exhibit worse behavior between 8 and 256K bytes.

Turning to Windows, the segment size limitation of the Win16 interface causes performance degradation in all the tests. The Win16 interface imposes a 64K byte upper limit for allocation of contiguous memory. Because of the

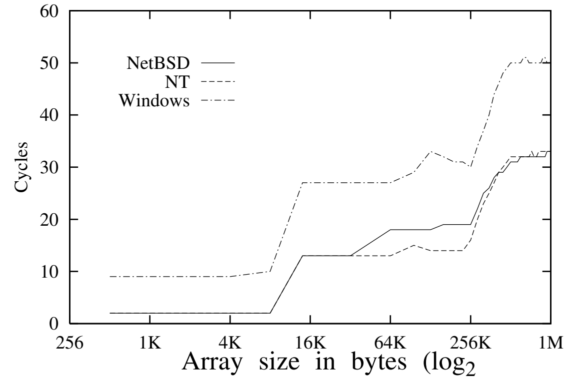


Figure 4. Memory Access Time. This graph illustrates the relationship between cycles per native integer memory reference and array size, for arrays ranging from 512 bytes to 1M byte. A stride of 128 bytes was used. The shape of the curves show how system policy, API, and memory hierarchy parameters combine to determine memory access time.

pointer-chasing loop used by our microbenchmark, it is impossible for the Win16 compiler to determine when a memory reference will cross a segment boundary. This forces the Win16 compiler to generate a four-instruction sequence to load the segment register, which is executed before every array reference. Segment register maintenance in Windows increases the instructions per memory reference from one to five, and minimum cycles per memory reference from two to nine.

As the size of the array increases, we see that the effects of the memory hierarchy on Windows reference times are similar to those of the other two systems, except that the Windows curve is shifted up due to the overhead of maintaining segment registers. The segmented address space used by the Win16 API causes significantly increased overhead both in terms of cycles and instructions for the Windows test, and is representative of behavior expected in Win16 programs that use significant amounts of data (more than 64K bytes) but have not been carefully tuned to accommodate segment boundaries. Readers should take note that this benchmark tests the worst possible case for Windows data reference patterns, and that the penalty for Win16 applications will often be smaller. In practice, skilled Windows programers can use special “near” and “far” declarations to control the compiler and avoid segment management overhead.

Our memory access test shows that the operating system can affect application performance in ways that have little to do with the latency of operating system activity during program execution. API and system policy have a large impact on the average memory access times for user code.

4.5 File System Performance

Our file system tests evaluate three kinds of activity:

- operations that hit in the disk cache,
- accesses to small files that go to disk, and
- file creation.

We designed our experiments to focus on each type of activity independently, and to draw out the differences between the systems. It is important to recognize that a real system will require a blend of operations that will not correspond to any single microbenchmark. For Windows NT we tested both the FAT and the NTFS filesystem. Windows uses the 16-bit MS-DOS implementation of the FAT filesystem by default, but a system configuration option can be set to use a 32-bit implementation. For our file system experiments we measured both implementations. All other Windows experiments reported in this paper used the default 16-bit FAT implementation.

A few details on the three file systems are useful for interpreting these results. A FAT file system has a single File Allocation Table (FAT) with a fixed size of 64K entries. Each entry holds information for one logical disk block. As the size of the FAT is fixed, the size of a logical disk block is determined by the size of the filesystem. Our tests were run using a 256M byte partition, which gives an allocation size of 4K bytes. Our NTFS filesystem used 4K byte blocks[1] (the maximum supported NTFS block size), and our NetBSD file system used 8K byte blocks. A crucial detail is that NTFS is a journaling file system [9]. This means that write operations are written to a log and then written again when the logged operation is committed. FAT is a very simple file system as compared to NTFS or FFS; this affects the amount of activity needed to perform basic file operations.

To investigate the cost of operations that hit in the disk cache we used a workload that repeatedly reads a set of files smaller than the cache. The relative behavior of NetBSD and the Windows NT/FAT system is similar to the comparison of NetBSD and Windows NT in the Syscall benchmark: NT/FAT consistently required more instruction overhead than NetBSD. Figure 5 shows that NT/FAT executes over twice as many instructions, and each instruction is more expensive due to higher I-Cache and ITLB miss rates as compared to NetBSD. For accessing small files in the buffer cache, NT/FAT runs 2.5 times slower. Using NTFS instead of FAT increases the overhead for Windows NT, with the instruction count 35% higher and the elapsed time in cycles increasing by 30%. The default Windows system

1. In NTFS terminology, our filesystem used 512 byte blocks and 4K byte clusters.

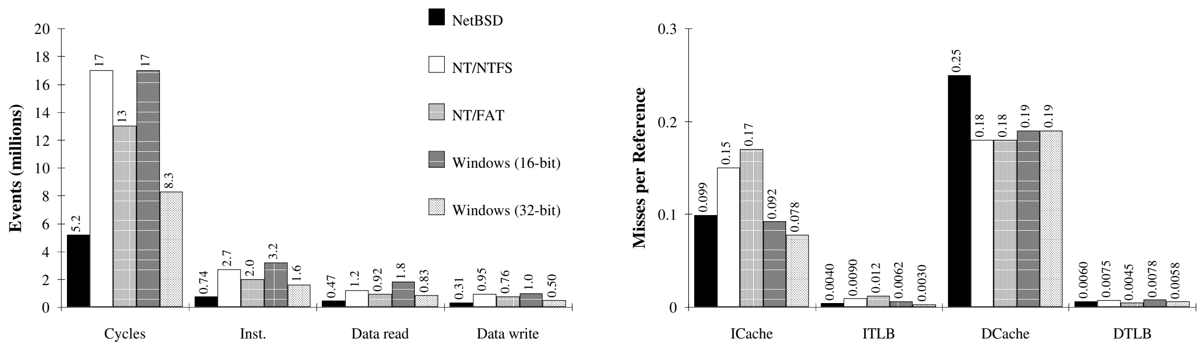


Figure 5. Buffer Cache Test. This test measured read performance from the file system cache by measuring the read time for 128 4K byte files that had been recently created. Hardware interrupts were monitored to ascertain that no disk accesses occurred.

shows significantly higher overheads than Windows with the 32-bit option enabled. Counts for cycles, data reads and data writes for the 32-bit system are about one-half of those measured for the 16-bit system. The 16-bit vs. 32-bit buffer cache implementation alone does not explain the performance difference, however, as the limiting resource in this experiment should be memory bandwidth and not instruction issue rate. We believe that an extra copy occurs within the user-level buffer cache used in the 16-bit case. The limitations of our methodology do not permit a more precise analysis.

For file system operations that require disk I/O, we found that Windows NT has much higher latency than the other systems. Figure 6 shows that, in both the NTFS and FAT cases, Windows NT performance lags behind the other two systems by almost a factor of two. One probable factor is the structure and complexity of Windows NT. Figure 6 shows that instruction counts for Windows NT are seven times higher than for NetBSD.

Another issue exposed by our experiments is disk layout. FFS places file data and meta-data in nearby cylinders to reduce seek times when accessing a single file, and places files in the same directory in nearby cylinders to reduce seeks between different files. In contrast, FAT places all file meta-data in a single table, requiring potentially long seeks when both file data and meta-data are accessed in succession. Comparing NTFS to FAT under Windows NT, instruction counts are higher for NTFS but cycle counts are lower. This suggests that NTFS provides better disk layout than FAT for this workload.

NetBSD	NT/NTFS	NT/FAT	Win (16)	Win (32)
8842	113276	59944	444	450

Table 4. Non-clock Interrupts during the Small File Disk Test.

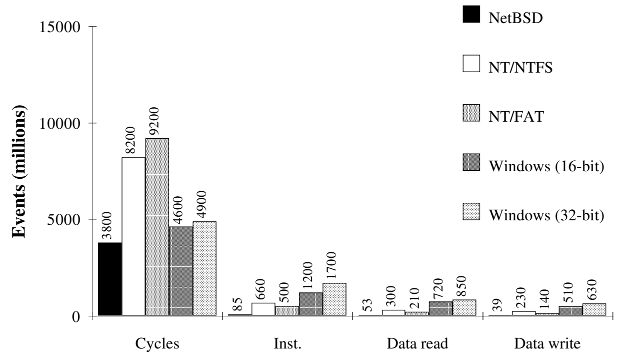


Figure 6. Small File Disk Test. Performance of file accesses to disk was measured by reading 8192 - 8K byte files. The files were split evenly among 32 directories.

Windows instruction behavior differs greatly from the other systems in that Windows busy waits during disk requests rather than using I/O interrupts.[1] This is a probable explanation for the high instruction counts and low cycle count for the Windows cases as compared to NT/FAT (Figure 6). The count of non-clock interrupts during the small file disk test (Table 4) shows that Windows is busy-waiting for disk events rather than using interrupts. Windows NT/FAT uses many more I/O interrupts per disk request than NetBSD. This is due to the absence of certain device-dependent optimizations in the Windows NT device driver implementation.[2] The large number of interrupts for NT/NTFS may be due to metadata logging overhead, although we could not determine this for certain.

For small file accesses, the 32-bit Windows option offers no advantage over the 16-bit case, increasing the latency by almost 10%. The similar performance is due to the fact that both implementations rely on the same 16-bit device driver for the SCSI disk.

Our last file system microbenchmark evaluates the cost of meta-data operations. FFS uses two synchronous writes for every file creation, to guarantee the integrity of file creation across system failures. In contrast, the disk caches used with FAT buffer meta-data operations in memory [12, 31], and NTFS writes them to a log, deferring the actual operation until a more convenient time. To quantify the performance impact of these policy differences, we used a benchmark that created 128 zero-length files (Figure 7). Meta-data operations are an order of magnitude faster

1. Windows performance is a factor of five worse (in cycles) when the default device driver is used to access the SCSI device. As MS-DOS runs in Virtual 8086 mode and Windows applications run in Protected 16-bit mode, the standard driver requires an expensive copy operation to transfer the I/O data from MS-DOS memory to memory the application can use. Better performance is possible with a SCSI device and device driver that use a technique called “VDMA” - DMA to virtual memory [33]. VDMA is supported by the NCR 815 SCSI interface we used.

2. Alessandro Forin. Personal Communication.

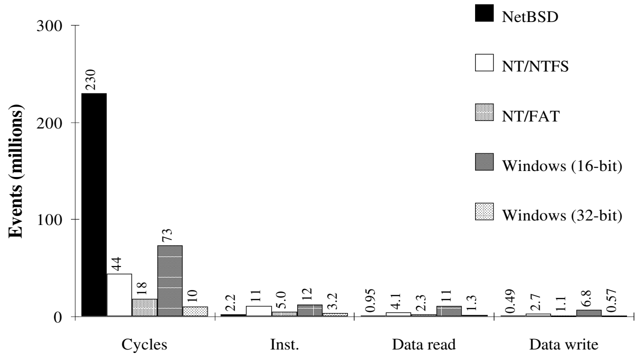


Figure 7. Meta-Data Test. This test measured the overhead of creating 128 zero-length files.

under Windows NT/FAT than under NetBSD/FFS. The low number of non-clock hardware interrupts shows that meta-data updates remain in memory for the FAT filesystems rather than being written synchronously to disk as with FFS (Table 5). As a result, this workload is disk-bound under NetBSD and CPU bound under Windows NT and Windows. The low number of I/O interrupts for NTFS is explained by the fact that it uses a lazy commit policy, batching log records rather than forcing them to disk immediately after a create [13].

The FAT tests for Windows NT and Windows show how weak FAT semantics influence the performance of meta-data updates. In both cases, FAT handles meta-data more quickly than FFS, although the high overhead of the 16-bit Windows implementation increases the overall instruction and cycle counts (Figure 2 and Figure 7). The meta-data test gives the best illustration of the impact of the compromise in NTFS between FAT and FFS semantics. NTFS provides low latency and recoverability, but does not guarantee that meta-data operations will survive a crash. FFS provides this guarantee but incurs higher latency. FAT provides neither recoverability nor guaranteed meta-data operations, but it does have better performance. Finally, the comparison of NT/FAT and the 32-bit Windows test shows the impact of the microkernel-like structure of Windows NT, with NT overhead almost a factor of two higher.

Overall, our file system measurements show that system structure has a significant impact on the cycles and instructions required for file system operations. For all our microbenchmarks, NT/NTFS requires four times more instructions than NetBSD for similar file operations. For operations that go to disk, the file allocation algorithms also affect latency and system overhead. While Windows NT performs worse than NetBSD for file operations to disk, it avoids overhead through relaxed commit semantics in the meta-data test. Additionally, by unifying the disk cache and VM free page pool, Windows NT can economically support a much larger disk cache than the statically allocated cache in NetBSD. For disk-intensive workloads, this can significantly reduce the number of disk operations required

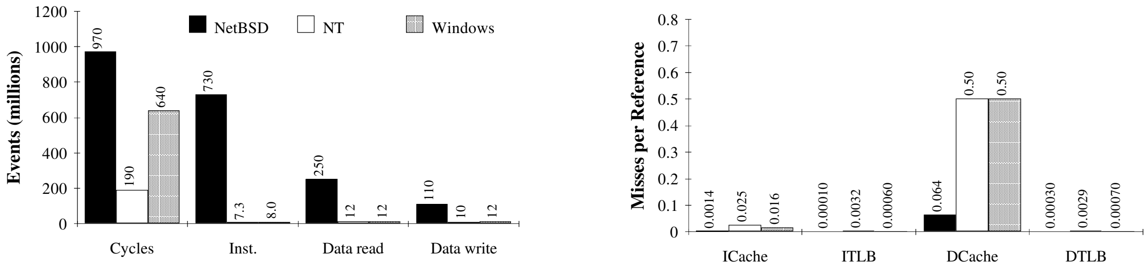


Figure 8. The Bitblt test. This workload repeatedly displayed 560x760 arrays of pixels to the screen.

by Windows NT as compared to NetBSD. This behavior is observed in our Web Server application benchmark (Section 5.3).

NetBSD	NT/NTFS	NT/FAT	Win (16)	Win (32)
261	46	0	0	0

Table 5. Non-clock interrupts during the Meta-data test.

4.6 Graphics Performance: Bitblt

The goals of our Bitblt experiment were to compare the three systems with respect to a specific graphics operation, and to gain insight on the impact of graphics performance on the Ghostscript workload. Although a comprehensive study of graphics performance is beyond the scope of this work, the performance of the bitblt operation exposes key differences between the three systems that give insight into other aspects of graphics performance.

Our Bitblt benchmark displayed a 570x1520 array of pixels in a window of size 570x760 (the maximum permitted by the resolution of our screen), repeating this operation 760 times. In the first iteration, rows 0-759 were displayed, then 1-760, and so on, increasing the maximum and minimum row indices by one with each iteration. Figure 8 shows results for an array of one-bit pixels. NetBSD requires a factor of ten more data references and a factor of 100 more instructions than Windows or Windows NT. To understand this behavior we consider the specific operations used by each system to move a pixel array from its source to the screen.

A 570x760 bitmap occupies 54K bytes. Copying the bitmap into graphics memory 760 times implies a transfer of approximately 40M bytes or 10 million 32-bit words. Windows and Windows NT copy each bitmap only once, from application memory directly into graphics memory as a 1-bit per pixel array. This explains the approximately 10 million data writes for Windows and Windows NT. X11 appears to copy each pixel map multiple times, with the most expensive copy transforming the 1-bit per pixel array into an 8-bit per pixel array. This increases the required number of writes by a factor of eight over Windows and Windows NT, accounting for 80 million writes. Our experiments indicate that an additional 26 million writes occur in the X11 server, although the complexity of the X11 source and the limitations of our tools made it difficult to identify the source of the writes with any greater precision. We measured about 1.5M kernel-mode writes during the test, indicating that no copy of the image data occurred in the kernel.

Although many counts for Windows and Windows NT are similar, the latency is significantly higher for Windows. Pipeline stalls due to write backups (counter 0x19) account for about 60% of the execution time for Windows (Table 6). We do not have an explanation for these high write buffer penalties.

NetBSD	Windows NT	Windows
1,700,000	8,600,000	407,000,000

Table 6. Pipeline stalls due to write backup for the Bitblt test.

The number of instructions executed in Windows and Windows NT is significantly less than the number of data references. This indicates that the REP prefix is being used to copy many words with a single instruction. In X11 pixels are copied using machine independent code written in C. It appears that the code generated by the compiler does not use the REP prefix, and each write instruction issued copies at most a single word. X11 uses twice as many read operations as write operations. We believe this is because each word is read twice during the conversion from one-bit to eight-bit pixels. Overall we see that protection and portability in X11 have a significant cost for the bitblt operation.

4.7 Network Throughput

Our network throughput benchmark was designed to determine if network throughput was limiting performance in our Web server experiments. We measured the time to send a 256K byte buffer to a remote host on a dedicated Ethernet, using TCP/IP and a 24K byte receive buffer. Figure 9 shows comparable networking behavior for NetBSD and Windows NT, with similar instruction, data references, and data cache miss counts. The only significant difference between the two systems was worse code locality for Windows NT, with both instruction cache and TLB misses significantly higher. This is consistent with results for other benchmarks, suggesting that Windows NT has worse overall instruction locality than the other systems.

Although Windows shows higher event counts than the other systems, throughput is comparable with the other two systems at about 1M byte per second. All three systems are limited by the capacity of the raw Ethernet, and not by internal software bottlenecks.

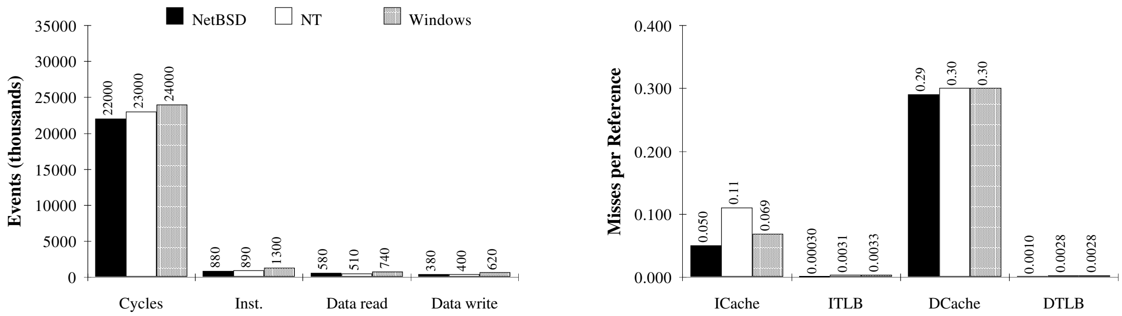


Figure 9. Network Throughput test. For this benchmark a 256K byte buffer was sent over a dedicated 10Mb Ethernet using a 24K byte receive buffer.

4.8 Microbenchmark Summary

Our experiments with microbenchmarks have identified several key performance differences between the three systems. In Windows, frequent CPU mode changes require expensive multicycle instructions, and the 64K byte segment size limitation imposes extra overhead in application code. The Null and Syscall benchmarks revealed higher instruction counts and instruction cache penalties in Windows NT, due in part to the user-level implementation of the Win32 API. At the same time, a more efficient graphics implementation and relaxed file system semantics give the Microsoft systems a performance advantage over NetBSD. Although they reveal important structural differences between the systems, the relevance of microbenchmarks to the performance of realistic workloads is limited. In the next section we examine the effect of the differences between the systems for more realistic workloads.

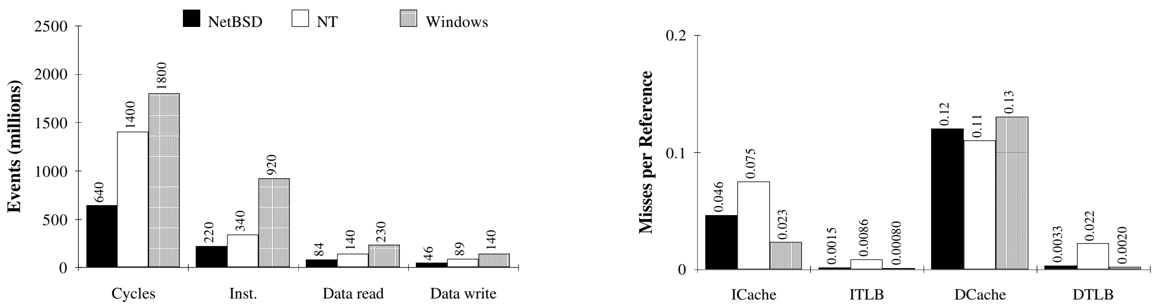


Figure 10. Wish. This benchmark was based on the “widget” demo included in the standard Tk release.

5. Application Workloads

Table 7 gives baseline information for our application workloads. Because all of the workloads have a working set size of under 32M bytes, no significant paging activity occurred during the experiments. Extensive use of dynamically loaded libraries makes it difficult to interpret executable sizes for these programs. For this reason we have provided both executable size and resident size. Although these measurements are crude, they do give an indication of the real memory requirements of the application.

Workload	Execution time (seconds)			Executable size (kilobytes)			Resident Size (kilobytes)		
	NetBSD	NT	Windows	NetBSD	NT	Windows	NetBSD	NT	Windows
Wish	7.05	14.94	18.95	433	1080	836	1700	1000	800
Ghostscript	16.92	9.30	20.32	397	472	555	2400	3000	1500
Web Server	45.56	83.33	52.22	243	225	243	3700	4700	1000

Table 7. Application Workloads. Executable size for Wish includes executables and Wish-specific DLLs. The resident sizes were measured using the ps command under NetBSD, the Performance Monitor under Windows NT, and Stretch utility from Microsoft Visual C++ 1.5 under Windows. The Web Server under NetBSD forks approximately 22 processes, each of which has an average resident size of 399K bytes. These resident sizes are not additive because all text pages are shared and data pages are shared copy-on-write. The NetBSD Web resident size estimate assumes 156K bytes of unique data per process.

5.1 Wish

Our Wish benchmark allows us to compare the overhead required by each of the three systems to support a graphical user interface. The system activity includes context switches and communication between the application and graphics server. Because the benchmark does not use large data files, little disk activity occurs during the run. Figure 10 shows that the cost of accessing system functionality in Windows results in higher overhead both in terms of cycle counts and instructions executed. Wish requires frequent switching between the graphics device VxD (protected 32-bit mode), the window manager (16 bit mode), and the application (16 bit mode). These frequent changes of CPU mode, along with 16 bit data transfers, contribute to higher cycle counts in Windows. The differences between Windows NT and NetBSD can be attributed to frequent context switches and stall cycles due to worse memory locality in Windows NT. Windows has relatively good instruction cache behavior, consistent with results for the microbenchmarks. However, the high instruction count dominates Windows performance.

Counts of segment register loads (Table 8) provide a useful indication of the sources of system overhead in Wish. In Windows NT and NetBSD, segment register loads are required for changes of protection domain. In Windows they are required for changes of machine mode and to support the segmented address space. Apart from being an indicator of system activity, segment register loads can be expensive instructions (Section 4.4). Windows NT required eight times as many segment register loads as NetBSD. This shows that hardware protection boundary crossings are much more frequent in Windows NT. In NetBSD context switches are avoided through more aggressive batching of graphics requests in X11. Windows requires a factor of 28 more segment register loads than Windows NT, demonstrating how context switching and segmented memory in Windows has substantial overhead for applications using graphical user interfaces.

NetBSD	Windows NT	Windows
170,827	1,400,228	37,715,255

Table 8. Segment register loads for Wish.

Elevated TLB miss rates for Windows NT (Figure 10) also suggest frequent context switches, and give evidence of memory locality problems in Windows NT as compared to the other two systems.

5.2 Ghostscript

To understand the results of our Ghostscript experiment, it is important to understand the differences in how graphics are rendered in the three systems. The NetBSD version of Ghostscript uses Xlib functions such as XDrawString() to render graphics. This distinguishes it from the Windows and Windows NT versions of Ghostscript, which render directly into a pixel map in application memory. When a page has been completed, control is transferred to the windowing system and the pixel map is displayed on the screen. This rendering operation was explored in isolation in Section 4.6 with our Bitblt microbenchmark. Figure 11 shows that graphics performance differences between the three systems have a significant impact on application performance.

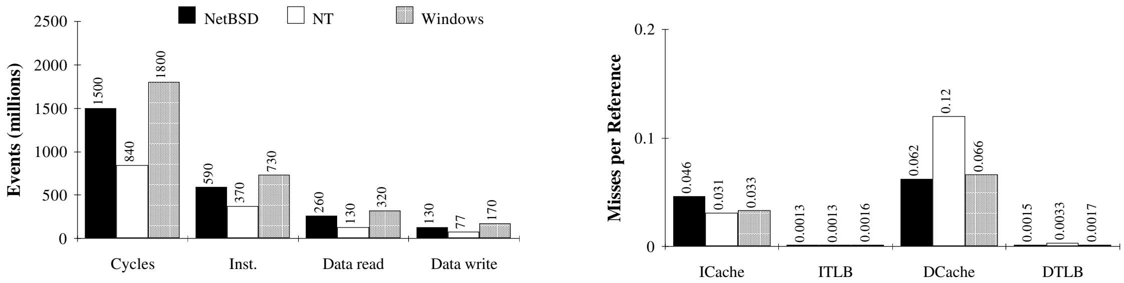


Figure 11. Ghostscript. For this benchmark, Ghostscript was used to preview a 13 page, 372K byte conference paper.

The performance of Windows NT was almost a factor of two better than the other systems, both in terms of instruction counts and elapsed time in cycles. Compared to Windows NT, NetBSD suffers from its inefficient graphics implementation. Part of the cycle differences between Windows and Windows NT is performance of graphics primitives; however, this does not explain the difference in instruction counts. We suspect that some part of the difference is due to a combination of the effects observed in the Memory Read microbenchmark and the Syscall benchmark. The difference in data reference counts between Windows NT and Windows is striking, although the limitations of our tools do not permit us to explain it fully. Sixteen bit data accesses under Windows are likely to be contributing to the problem.

We ran Ghostscript using publicly available distributions on all three systems, and although the three implementations are substantially similar, only the UNIX version displays in color. Color has minimal impact on the bandwidth required by X11 protocol requests, but it could have a large impact for Windows and Windows NT versions of Ghostscript if they used color rather than bitmaps. We investigated this impact of color pixel maps by running a version of the Bitblt test which used 8 bits-per-pixel rather than one. Not surprisingly, the number of data reference operations and the latency of the benchmark increased by approximately a factor of eight. This suggests that a color version of Ghostscript for Windows NT might have to change the way it interacts with the graphics system in order to avoid a large performance hit, either by using high-level graphics operations, more like X11 protocol requests, or by rendering directly into screen memory.

5.3 Web Server

Baseline results for the Web server appear in Figure 12. The throughput for the NetBSD and Windows servers is similar at about 20 requests per second, while the NT server achieves only about 12 requests per second. Unlike the network throughput benchmark, the limiting resource in this case was not the Ethernet, which was transferring data at less than 11% of its capacity (117K bytes/second for Windows NT), but rather the combined requirements of network activity and filesystem access. Additionally we believe that the Windows NT system was affected by problems in its networking implementation.

For each operating system, we set the Web client parameters to maximize server throughput. The accept queues of both Windows and Windows NT were easily overrun by parallel connection attempts, resulting in a significant performance penalty. Windows throughput dropped significantly under a load of nine or more concurrent requests, so we limited the number of concurrent Windows requests to eight. The Windows server generated approximately 900 connection-refused messages for 1024 successful HTTP retrievals. For Windows NT, even a small number of back-to-back connection requests caused long periods of idle time that we could not explain. By inserting a 40ms pause in the client between requests we were able to achieve reasonable throughput with up to 15 concurrent requests. We did observe higher throughputs (up to 16 requests per second) when the number of concurrent requests was increased over 15, but at this load the NT system would sometimes enter a state where throughput dropped to about four requests per second. This degenerate behavior occurred consistently with 20 or more concurrent requests. We ran our Windows NT experiments with a maximum of 15 concurrent requests to avoid this degenerate behavior. The throughput of NetBSD increased in a stable way with increased client parallelism. We drove the NetBSD Web server with 32 concurrent requests. No connections were refused under NetBSD.

All three systems suffer from very high memory system penalties. Figure 12 shows that Web server cache miss penalties are higher for the NetBSD and Windows NT Web servers than for our other application workloads. Although there is some locality in our Web trace, the cache memory is not large enough to take advantage of it. This suggests that machines intended to provide high performance Web service could benefit from optimizations to avoid cache latency.

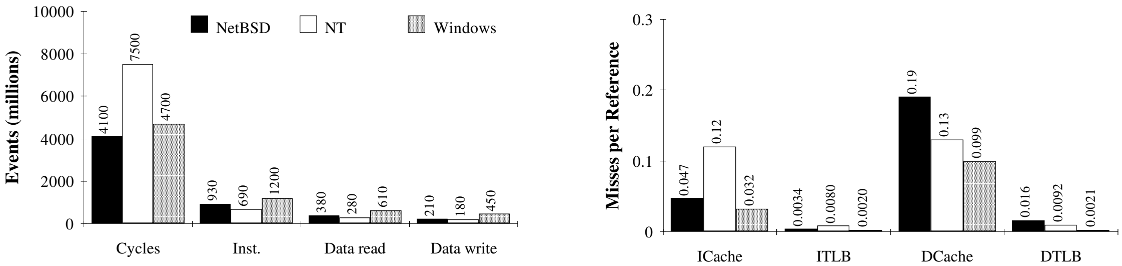


Figure 12. Web Server. This workload used a trace of 1024 Web server requests to the host *www.nsca.uiuc.edu* for a total of 9.7M bytes of data. The test was run using a dedicated network. The server load was generated by a client using non-blocking socket I/O to issue parallel HTTP requests. The servers were configured to keep an access log and to ignore access controls.

Instruction cache miss rates are significantly higher for Windows NT than for the other two systems, and this supports our overall conclusion that Windows NT has worse instruction locality than the other two systems.

Windows NT requires fewer instructions than the other two systems. Its integrated buffer cache/free memory pool means that more file system requests can be serviced without going to disk, hence there are fewer disk requests (Table 9). The other two systems have smaller buffer caches, resulting in more disk activity during the trace. The NetBSD system benefits from higher parallelism to maintain throughput in spite of increased disk activity.

NetBSD	Windows NT	Windows
13226	13516	43615

Table 9. Non-clock interrupts for the Web server. The interrupt counts for the Web Server benchmark show that NT benefits from its integrated buffer cache, doing many fewer disk requests than NetBSD. (Recall from Section 4.5 that NT uses four interrupts per disk request as compared to one for NetBSD). Our observation of the LED activity indicator on the disk drive confirms this conclusion.

5.4 Discussion

Our experiments have revealed a number of structural issues that affect the performance of the three operating systems we studied. System designers can benefit from an understanding of these issues.

Prior work has documented the overhead required to support a microkernel system [7]. Our experiments confirm these prior results and show that they apply to Windows NT and more generally to systems that implement system APIs as user processes.

Prior to our investigation of Windows we imagined we might discover advantages to the unified address space model that would provide support for proposed system structures such as Exokernels [14] and software virtual memory [25]. We did not. The Windows example provides no convincing evidence for the advantages of eliminating protection boundaries.

A number of shortcomings of the Windows system model have become apparent in our study of the three systems. System call hooks provide a degree of flexibility in Windows, but there is a performance penalty associated with that flexibility. Because Windows must support prior generations of software, machine mode changes are unavoidable. The penalties associated with this backwards compatibility requirement are apparent from our measurements, and the penalties are likely to increase in future generations of 80386 compatible processors [10]. The segmented memory architecture used by Windows not only complicates programming but also has serious performance implications. As with machine mode changes, these penalties are expected to increase in future PC processors. The performance issues that we discuss for Windows for Workgroups 3.11 have been recognized by Microsoft. By discouraging use of segmented memory and reducing machine-mode changes, Windows 95 [21] addresses these problems. Our results for 32-bit Windows functionality in Sections 4.2 and 4.5 give an indication of the performance that can be anticipated in Windows 95.

Our Web server experiment demonstrates the potential benefit of integrating the buffer cache and free page pool for Web servers and other systems where disk caching has a significant performance impact. However, the memory test provides an example of how such integration of resources also creates new problems in resource management. System implementors need to be aware of these problems and implement systems with the flexibility to accommodate potential tradeoffs.

Hardware manufacturers who want to provide improved performance for the current generation of Windows software will probably need to control the latency of multi-cycle operations such as segment register loads and machine mode changes. If Mach 3.0 and Windows NT are indications of the direction of future operating systems, hardware designers need to anticipate an increase in the absolute amount of system activity needed to support a typical user computation. They may need to consider more aggressive techniques for accommodating poor instruction locality [5, 44].

6. Conclusions

The structure of personal computer operating systems has a significant impact on the cost of accessing system functionality. Our experiments have served to quantify performance differences in three personal computer operating system, and to identify some of their sources.

The Windows backward compatibility requirement is detrimental to performance. Using our microbenchmarks we identified machine mode changes, system call hooks, and segmented memory as aspects of Windows structure that have a detrimental impact on performance.

A significant part of the cost of system functionality in Windows is due to the structure of the system rather than the API required by Windows applications. The superior performance of Windows NT relative to Windows suggests that PC-style applications can be supported without the large negative performance impact that occurs in the current Windows implementation. Even so, the microkernel system structure used by Windows NT to support protected address spaces and multiple APIs leads to a significant increase in system overhead for microbenchmarks relative to NetBSD. Furthermore, our measurements consistently showed that Windows NT has worse instruction locality than the other two systems. Windows NT presents a higher load to TLBs and caches. This may be due to its microkernel-like structure, or the way objects are used for resource management, although without better tools it is difficult to pinpoint the difference with any precision. Finally, an efficient graphics implementation helped Windows NT achieve the best overall performance for Ghostscript, in spite of the negative performance impact of Windows NT structure that was exposed by the microbenchmarks.

Our experiments with NetBSD demonstrate the performance advantages of its monolithic structure. Although in most cases the performance impact of protection and portability is not significant, it can lead to extra copies and poor use of hardware features, as shown by the graphics microbenchmark. The poor use of graphics hardware in NetBSD is due to issues of portability, adherence to open standards, and the overhead required to support a network windowing system. It would be possible to improve graphics performance for NetBSD by supporting more device-dependent graphics, but it is not clear that the trade-offs, in terms of functionality, make this a desirable option.

A major shortcoming of our workloads is that they are UNIX-centric. They fail to demonstrate the strong point of Windows — the abundance of popular application software available for that platform. Even so, they have permitted us to understand performance issues and identify crucial structural differences between personal computer operating systems.

Microbenchmarks are useful because they are easy to analyze. With realistic applications it is much more difficult to get a complete understanding of software activity during a computation. Our experiments with application workloads give evidence that the behavior we isolated with our microbenchmarks has significant impact in realistic situations. Performance counters provide clues about behavior, but often they cannot give conclusive evidence. A means of acquiring complete and detailed measurements of the activity within the system is needed. This is the subject of continuing work at Harvard.

7. Acknowledgments

We are grateful to Andrew Black, Alan Eustace, Sando Forin, Mike Jones, Gene Munce, and the SOSP program committee for useful comments on preliminary drafts of this paper. We would also like to acknowledge Larry McVoy, who made the *lmbench* suite available for our work. Special thanks to Terje Mathisen for his work documenting the Pentium counters.

8. Bibliography

[1]. Chris Adie. *HTTP Server Manual, Version 0.96*. European Microsoft Windows NT Academic Centre, University of Edinburgh, Edinburgh, UK, November 1994.

[2]. Adobe Systems Incorporated. *Postscript Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1985.

[3]. Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[4]. T. Berners-Lee, R. Cailliau, J-F. Groff, and B. Pollermann. “World-wide web: The Information Universe.” *Electronic Networking Research, Applications and Policy*, Volume 2, Number 1, 1992, pages 52–58.

[5]. Brian N. Bershad, J. Bradley Chen, Dennis Lee, and Theodore H. Romer. “Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches.” *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating System*, October 1994, pages 158–170.

[6]. Geoff Chappell. *DOS Internals*. Addison-Wesley, Reading, Massachusetts, 1994.

[7]. J. Bradley Chen and Brian N. Bershad. “The Impact of Operating System Structure on Memory System Performance.” *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993, pages 120–133.

[8]. J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Margo Seltzer, and Mike Smith. *The Measured Performance of Personal Computer Operating Systems*. Technical Report TR-09-95, Center for Research in Computing Technology, Harvard University.

[9]. S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, and R. Sidebotham. “The Episode File System.” *Proceedings of the 1992 Winter Usenix Conference*, San Francisco, CA, January 1992.

[10]. John Clyman and Nick Stam. “Intel Positions the P6 for 32-bit Apps, Eschewing Windows 3.1.” PC Magazine Trends Online (6/30/95), http://www.zdnet.com/~pcmag/trends/tr0630b.htm

[11]. Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 1993.

[12]. Helen Custer. *Inside the Windows NT File System*. Microsoft Press, Redmond, Washington, 1994.

[13]. Helen Custer. “Detail and Implementation Details of the Windows NT Virtual Block Cache Manager.” *Microsoft Systems Journal*, Volume 10, Number 7, July 1995, pages 67-79.

[14]. Dawson R. Engler, M. Frans Kaashoek and James O’Toole Jr. “Exokernel: An Operating System Architecture for Application-level Resource Management.” *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[15]. Ben Ezzell. “The Power Under the Hood; Preview: Windows NT.” *PC Magazine*, Volume 12, Number 11, June 15 1993, pages 219-263.

[16]. Alessandro Forin and Gerald R. Malan. “An MS-DOS File System for UNIX.” *Proceedings of the 1994 Winter USENIX Conference*, January 1994, pages 337–354.

[17]. Gary Gunnerson. “Network Operating Systems: Playing The Odds.” *PC Magazine*, Volume 12, Number 18, October 26, 1993, pages 285–336.

[18]. Linley Gwennap. “P6 Underscores Intel’s Lead.” *Microprocessor Report*, Volume 9, no. 2, February 1995.

[19]. Intel Corporation. *Pentium Family User’s Manual, Volume 3: Architecture and Programming Manual*. Intel Corporation, 1995.

[20]. R.E. Kessler and Mark D. Hill. “Page Placement Algorithms for Large Real-Indexed Caches.” *ACM Transactions on Computer Systems*, November 1992.

[21]. Adrian King. “Windows, the Next Generation: An Advance Look at the Architecture of Chicago.” Microsoft Systems Journal, Volume 9, Number 1, pages 15-24, January 1994.

[22]. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1989.

[23]. David S. Linthicum. “Life After DOS.” *PC Magazine*, May 31, 1994, pages 203–237.

[24]. David S. Linthicum and Steven J. Vaughan-Nichols. “The Beast Turns Beauty; UNIX on Intel.” *PC Magazine*, Volume 12, Number 11, June 15, 1993, page 219–263.

[25]. Steve Lucco. *Software Fault Isolation*. Invited talk, Harvard Computer Science Colloquium Series, Harvard University, 28 November 1994.

[26]. Terje Mathisen. “Pentium Secrets.” *Byte*, July 1994, pages 191–192.

[27]. M. McKusick, W. Joy. S. Leffler, R. Fabry. “A Fast File System for UNIX.” *ACM Transactions on Computing Systems*, Volume 2, Number 3, August 1984, pages 181-197.

[28]. Larry McVoy, *lmbench: Portable Tools for Performance Analysis*, To appear in *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[29]. Microsoft Corporation. *Microsoft Windows Guide to Programming*. Microsoft Press, Redmond, Washington, 1990.

[30]. Microsoft Corporation. *Microsoft Windows NT Resource Kit*. Microsoft Press, Redmond, Washington, 1993.

[31]. Microsoft Corporation. *MS-DOS SMARTDRV help page*. Redmond Washington, 1994.

[32]. Microsoft Corporation. *SDKs: Windows for Workgroups 3.11 Addendum; Chapter 1 - Windows for Workgroups 3.11 Architecture*. Microsoft Developer Network Development Library, Microsoft Developer Network, Redmond, Washington, January 1995.

[33]. Intel Corporation. *PCI SCSI Adapters, PCISCSI Product Family Installation Guide (with SDMS 3.0 User’s Guide)*. Intel Corporation, Order number 619896-002, 1994.

[34]. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[35]. Matt Pietrek. *Windows Internals*. Addison-Wesley, Reading Massachusetts, 1993.

[36]. Jeff Prosis. “Adventures in Flatland with VxDs.” *PC Magazine*, May 31 1994.

[37]. Richard Rashid, Gerald Malan, David Golub, and Robert Baron. “DOS as a Mach 3.0 Application.” *Proceedings of the USENIX Mach Symposium*, pages 27-40, November 1991.

[38]. Andrew Schulman, David Maxey, and Matt Pietrek. *Undocumented Windows*. Addison-Wesley, Reading Massachusetts, 1992.

[39]. Andrew Schulman. *Unauthorized Windows 95*. IDG Books Worldwide, Inc., San Mateo CA, 1994.

[40]. Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan. “File System Logging versus Clustering: A Performance Comparison.” *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, LA, January 1995, pages 249-264.

[41]. Richard Hale Shaw. “An Introduction to The Win32 API.” *PC Magazine*, pp 291–297, April 1994.

[42]. Jon Udell et al. “The Great OS Debate, Special Report on Advanced Operating Systems.” *Byte*, January 1994, pp. 117–168.

[43]. Mark L. Van Name and Bill Catchings. “Reaching New Heights in Benchmark Testing.” *PC Magazine*, December 1994, pages 327–332.

[44]. Richard Uhlig, David Nagle, Trevor Mudge, Stuart Sechrest, and Joel Emer. “Instruction Fetching: Coping with Code Bloat.” *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa-Margherita Ligure, Italy, May 1995, pages 345-356.