A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98

Erik Cota-Robles
James P. Held
Intel Architecture Labs

Abstract

Windows† 98 and NT† share a common driver model known as WDM‡ (Windows Driver Model) and carefully designed drivers can be binary portable. We compare the performance of Windows 98 and Windows NT 4.0 under load from office, multimedia and engineering applications on a personal computer (PC) of modest power that is free of legacy hardware. We report our observations using a complementary pair of system performance measures, *interrupt* and *thread latency*, that capture the ability of the OS to support multimedia and real-time workloads in a way that traditional throughput-based performance measures miss. We use the measured latency distributions to evaluate the quality of service that a WDM‡ driver can expect to receive on both OSs, irrespective of whether the driver uses thread-based or interrupt-based processing. We conclude that for real-time applications a driver on Windows NT 4.0 that uses high, real-time priority threads receives an order of magnitude better service than a similar WDM‡ driver on Windows 98 that uses Deferred Procedure Calls, a form of interrupt processing. With the increase in multimedia and real-time processing on PCs the interrupt and thread latency metrics have become as important as the throughput metrics traditionally used to measure performance.

1. Introduction

Real-time computations in multimedia applications and device drivers are typically performed in response to interrupts or the completion of previous computations that were themselves performed in response to interrupts. Under the Windows Driver Model (WDM‡) [1][21] such computations are typically implemented as either Deferred Procedure Calls (DPCs)[1][21], a form of interrupt processing, or in kernel mode threads. The ability of applications and drivers to complete their computations before their respective deadlines is thus a function of the expected worst-case delay between the hardware interrupt and the start of the computation.

These delays, or *latencies*, are highly sensitive to the amount of OS overhead that is incurred to service any other applications that may be executing concurrently on the system. Traditional real-time systems cope with this problem by strictly limiting the amount of concurrent non-real-time computation and by using a real-time OS with tightly bounded service times. This minimizes the amount of overhead penalty to which any one computation is subjected. On personal computer and workstation platforms the execution environment is highly dynamic and may include a wide variety of concurrently executing applications whose nature can only be estimated in advance. It is therefore not practicable to either limit application concurrency or to use a real-time OS.

Application (low latency streaming**)	Buffer size in ms. (***t***) (typ. range)	Number of buffers (***n***) (typ. range)	Latency Tolerance (***n***-1)****t*** ms.
ADSL	2 to 4	2 to 6	4 to 10
Modem	4 to 16	2 to 6	12 to 20
RT audio**	8 to 24	2 to 8[1]	20 to 60[1]
RT video**	33 to 50	2 to 3	33 to 100

Table 1: Range of Latency Tolerances for Several Multimedia and Signal Processing Applications, tolerance range roughly (n_{max}-1)*t_{min} to (n_{min}-1)*t_{max} ms.

Applications and drivers vary widely in their tolerance for missed deadlines, and it is often the case that two drivers with similar throughput requirements must use very different kernel services (e.g., DPCs and kernel mode threads). Before an application or driver misses a deadline all buffered data must be consumed. If an application has *n* buffers each of length *t*, then we say that its *latency tolerance* is (*n*-1) * *t*. Table 1 gives latency tolerance data for several applications [5][11]. It is interesting to note that the two most processor-

† Third-party brands and names are the property of their respective owners.

‡ **N.B. In this paper "Windows Driver Model" and "WDM" refer to the subset of WDM 1.0 that is compatible with NT 4.0 kernel mode drivers.**

[1] 8 is the maximum number of buffers used by Microsoft's KMixer and is on the high side. 4 buffers, giving a latency tolerance of 20 to 40 ms., is more realisitic.

intensive applications, ADSL and video at 20 to 30 fps, are at opposite ends of the latency tolerance spectrum. Traditional methodologies for system performance measurement focus on throughput and average case behavior and thus do not adequately capture the ability of a computing system to perform real-time processing.

In this paper we propose a new methodology for OS performance analysis that captures the ability of a non-real-time computing system to meet the latency tolerances of multimedia applications and low latency drivers for host-based signal processing. The methodology is based on a complementary pair of microbenchmark measures of system performance, *interrupt latency* and *thread latency*, which are defined in section 2.1. Unlike previous microbenchmark methodologies, we make our assessments of OS overhead based on the distribution of individual OS service times, or latencies, on a loaded system. We present extremely low cost, non-invasive techniques for instrumenting the OS in such a way that individual service times can be accurately measured. These techniques do *not* require OS source code, but rely instead on hardware services, in particular the Pentium® and Pentium II processors' time stamp counters[9][10], and can thus be adapted to any OS.

We use these techniques to compare the behavior of the Windows Driver Model (WDM‡) on Windows 98 and Windows NT under load from a variety of consumer and business applications. We show that for real-time applications a driver on Windows NT 4.0 that uses either Deferred Procedure Calls (DPCs), a form of interrupt processing, or real-time priority kernel mode threads will receive service at least one order of magnitude better than that received by an identical WDM‡ driver on Windows 98. In fact, a driver on Windows NT 4.0 that uses high, real-time priority threads will receive service one order of magnitude better than a WDM‡ driver on Windows 98 which uses DPCs. In contrast, traditional throughput metrics predict a WDM‡ driver will have essentially identical performance irrespective of OS or mode of processing.

The remainder of this section provides background on prior work on OS benchmarking and the performance analysis of real-time systems. Section 2 presents our methodology for OS performance analysis, including definitions of the various latencies in which we are interested and a description of our tools and measurement procedures. Section 3 presents our application stress loads and test system configuration. Section 4 presents and discusses our results. In section 5 we explore the implications of our results for hard real-time drivers, such as soft modems, on Windows 98 and Windows NT. Section 6 concludes.

1.1 Macrobenchmarks

Traditional methodologies for system performance measurement focus on overall throughput. Batch macrobenchmarks, whether compute-oriented such as SpecInt† or application-oriented such as Winstone† [22], drive the system as quickly as possible and produce one or a few numbers representing the time that the benchmark took to execute. While appropriate for batch and time-shared applications this type of benchmarking totally ignores significant aspects of system performance. As Endo, et. al. note [7], throughput metrics do not adequately characterize the ability of a computing system to provide timely response to user inputs and thus batch benchmarks do not provide the information necessary to evaluate a system's interactive performance.

Current macrobenchmarks essentially ignore latency with the result that the resulting throughput analysis is an unreliable indication of multimedia or real-time performance. For multimedia and other real-time applications, increased throughput is only one of several prerequisites for increased application performance. Because macrobenchmarks do not provide any information about the distribution of OS overhead, they do not provide sufficient information to judge a computing system's ability to support real-time applications such as low latency audio and video.

1.2 Microbenchmarks

Microbenchmarks, in contrast, measure the cost of low-level primitive OS services, such as thread context switch time, by measuring the average cost over thousands of invocations of the OS service on an otherwise unloaded system. The principal motivations for these choices appear to have been the limited resolution of traditional hardware timers and a desire to distinguish OS overhead from hardware overhead by using warm caches, etc. The result, as Bershad *et. al.* note [2], is that microbenchmarks have not been very useful in assessing the OS and hardware overhead that an application or driver will actually receive in practice.

Most previous efforts to quantify the performance of personal computer and desktop workstation OSs have focused on average case values using measurements conducted on otherwise unloaded systems. Ousterhout evaluates OS performance using a collection of microbenchmarks, including time to enter/exit the kernel, process context-switch time, and several file I/O benchmarks [19]. McVoy and Staelin extend this work to create a portable suite, *lmbench*, for measuring OS as well as hardware performance primitives [17]. Brown and Seltzer extend *lmbench* to create a more robust,

flexible and accurate suite, *hbench:OS*, which utilizes the performance counters on the Pentium and Pentium Pro processors to instrument the OS [3].

For the purposes of characterizing real-time performance, all of these benchmarks share a common problem in that they measure a subset of the OS overhead that an actual application would experience during normal operation. For example, Brown and Seltzer revise the *lmbench* measurement of context switch time so as to exclude from the measurement any effects from cache conflict overhead due to faulting of the working set of a new process. The motivation given is that by redefining context switch time in this manner the *hbench:OS* can obtain measurements with a standard deviation an order of magnitude less than those produced by *lmbench*. While this accurately characterizes the actual OS cost to save/restore state, one must in addition use another microbenchmark to measure cache performance and then combine the two measurements in an unspecified manner in order to obtain a realisitic projection of actual application performance. Furthermore, none of these OS micro-benchmarks directly addresses response to interrupts, which is of prime importance to low latency drivers and multimedia applications.

In contrast to the OS microbenchmarks discussed above, Endo, et. al., develop microbenchmarks based on simple interactive events such as keystrokes and mouse clicks on personal computers running Windows NT and Windows 95 [7]. They also construct activity-oriented *task* benchmarks designed to model specific user actions when using popular applications such as Microsoft Word. These benchmarks do address response to interrupts and detailed distributions are reported for some of the data. However, the authors' focus is on interactive response times, which for low-level input events, such as mouse and keyboard, is generally regarded as being adequately responsive if the latencies are in the range of 50 to 150 ms. [20] . As we have seen above (Table 1), except for video, this is considerably longer than the latency tolerances of the low latency drivers and multimedia applications that we consider here, which have tolerances between 4 and 40 milliseconds, depending on the specific application.

1.3 Real-Time Systems

Efforts to characterize the behavior of real-time systems, both software and hardware, have focused largely on worst-case behavior and assumed that the overall system load is known in advance. Katcher *et. al.*, for example, decompose OS overhead into the four categories of preemption, exit, non-preemption and timer interrupts [13]. *Preemption* is the time to preempt a thread; *exit*, the time to resume execution of a previously preempted thread; and *non-preemption*, the blocking time due to interrupt handling which does not result in preemption (i.e., the thread is added to the ready queue).

For systems with a fixed priority preemptive scheduler, it is common to use Rate Monotonic Analysis (RMA) to determine whether each of the system's threads can be scheduled so as to complete before its deadline. Traditionally this has been done by ignoring OS overhead [15], but recently techniques have been developed to include worst-case OS behavior into the analysis [14]. While such models are comprehensive and adequate for real-time OSs, they are overly pessimistic for Windows, which has worst case times for system services, such as context switching, that are orders of magnitude longer than average case times.

A further complication is that computationally intensive drivers, such as those for host-based signal processing, perform significant amounts of processing at "high priority" (e.g., in an interrupt service routine (ISR)). As an example, the datapump[2] for a software modem will typically execute periodically with a cycle time of between 4 and 16 milliseconds and take somewhat less than 25% of a cycle (i.e., 1 to 4 milliseconds) on a personal computer with a 300 MHz Pentium II processor. Clearly, multi-millisecond computations in an ISR will impact both interrupt and thread latency; they will also render a traditional worst-case analysis still more pessimistically. In previous work we have shown how RMA can be extended to general-purpose OSs that have highly non-deterministic OS service times in order to obtain reasonable estimates of real-time performance [4]. We will return to this subject in section 5.2.

2. Methodology

We sought a small set of microbenchmarks that would encapsulate the effects of OS overhead from a real-time standpoint but could be manageably incorporated into a performance analysis in order to accurately forecast the real-time performance of Windows applications and drivers. Since our goal was for the benchmarks to be applicable to a variety of real-time applications, we avoided task-oriented benchmarks of the type used by Endo, et. al., [7] in favor of general microbenchmarks.

[2] The datapump is the modem physical interface layer, analogous to the OSI PHY layer for networks.

Because user mode applications can be a noticeable impediment to timely response by the operating system, we measured latency in the presence of the stress from unrelated applications. This approach is valid even for assessing the performance that real-time portions of large multimedia applications will receive with no concurrent applications. Indeed, from the standpoint of low level real-time drivers (e.g., a kernel mode soft modem or low latency soft audio codec) the rest of the application (e.g., the user mode video codecs or the GUI display) is, for all practical purposes, an external application load.

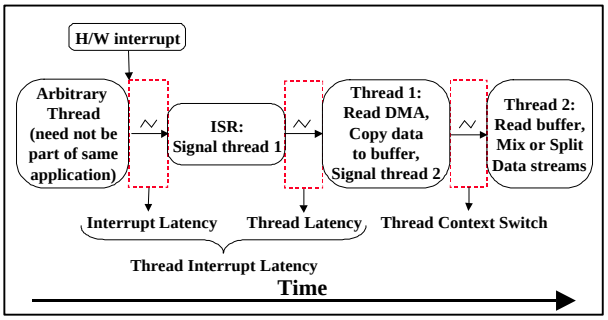

Figure 1: Interrupt Latency, Thread Latency and Thread Context Switch Time

2.1 Latency

Interrupt latency is defined to be the delay from the assertion of the hardware interrupt, *as seen by the processor*, until the first instruction of the software interrupt service routine (ISR) is executed. Thus, it measures the total delay to initial servicing of an interrupt. This encompasses the maximum time during which interrupts are disabled as well as the bus latency necessary to resolve the interrupt, but does not include the bus latency prior to the assertion of the interrupt at the processor. Figure 1 depicts an idealized timeline with interrupt latency, thread latency and thread context switch time marked.

Thread latency is defined to be the delay from the time at which an ISR signals a waiting thread until the time at which the signaled thread executes the first instruction after the wait is satisfied. Thus, it measures the worst-case thread dispatch latency for a thread waiting on an interrupt, measured from the ISR itself to the first instruction executed by the thread after the wait. Thread latency encompasses a variety of thread types and priorities (e.g., kernel mode high real-time priority) and includes the time required to save and restore a thread context and obtain and/or release semaphores. It represents the maximum time during which the operating system disables thread scheduling. An important point to note is that thread latency subsumes thread context-switch time since, in the general case, the proper thread is not executing when an interrupt arrives. We distinguish between thread latency, defined above, and *thread interrupt latency*, defined to be the delay from the assertion of the hardware interrupt until the thread begins execution.

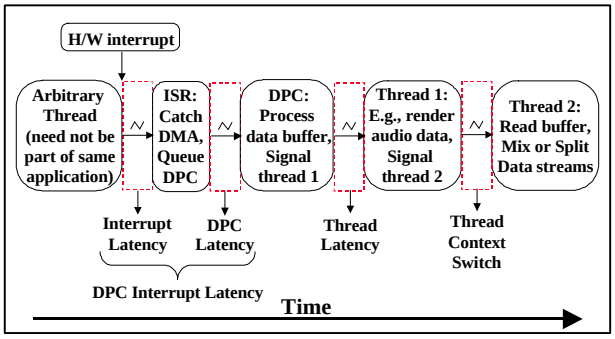

Figure 2: DPC Latency, DPC Interrupt Latency and WDM Thread Latency

In the Windows Driver Model (WDM‡) interrupts are preemptible and are supposed to be very short [1][21]. WDM‡ makes a Deferred Procedure Call (DPC) available for drivers that require longer processing in "interrupt context". We distinguish between *DPC latency*, which is defined to be the delay from the time at which the software ISR enqueues a DPC until the first instruction of the DPC is executed, and *DPC interrupt latency*, which is defined to be the sum of the interrupt and DPC latencies, as shown in Figure 2. Because ordinary DPCs queue in FIFO order, DPC latency encompasses the time required to enqueue and dequeue a DPC as well as the aggregate time to execute all DPCs in the DPC queue when the DPC was enqueued. Because drivers are not supposed to do substantial processing in a WDM‡ ISR, we will measure WDM‡ thread latencies from DPC to thread, and concentrate on DPC interrupt latency and thread latency.

2.2 Latency Measurement Tools

We developed a number of WDM‡ drivers that measure interrupt and thread latency. The thread latency driver is binary portable between Windows 98 and Windows NT, but the Windows 98 interrupt latency driver uses an interface unique to Windows 9x to install its own timer handler and is thus not portable to NT. The drivers have simple command line control applications and use the Pentium processor's time stamp register (TSR) for timing information. The latencies are returned to the application via WDM‡ I/O Request Packets (IRPs) which the application supplies via a call to the Win32† ReadFileEx API. Figure 3 gives an execution timeline and sections 2.2.1 through 2.2.5

present pseudocode for the Windows NT DPC interrupt latency and thread latency tool.

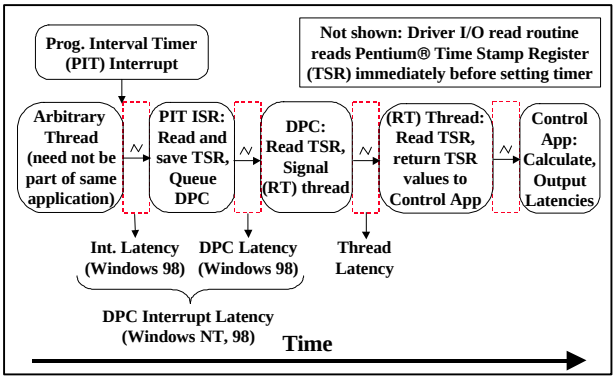

Figure 3: WDM‡ Interrupt, DPC and Thread Latency Measurement Tool

In the case of interrupt latency, because the driver cannot read the time stamp counter at the instant when the hardware interrupt is asserted, the driver I/O read routine reads the time stamp register and sets a timer to expire in a given number of milliseconds. The interrupt latency drivers estimate the time stamp at which the timer expired using the time stamp from the I/O read routine and record this as the estimated time stamp for the hardware interrupt. This approach suffers from limited resolution (basically +/- the cycle time of the Programmable Interval Timer (PIT) timer, whose frequency we have increased to 1 KHz). Because we were mainly interested in characterizing the latency "tail", which on Windows 98 extends past 10 milliseconds, we accepted this imprecision with only minor qualms. To put it another way, we are interested in the frequency of long latency events, so we care about the *magnitude* of long latency events and the *number* of short latency events.

Furthermore, on Windows 98 it is possible, using legacy interfaces, to supply our own timer ISR, whereas on Windows NT this would require source code access. Our NT driver thus records only DPC interrupt latency whereas our Windows 98 driver records interrupt latency, DPC latency, and DPC interrupt latency. This is shown in Figure 3.

As the following pseudocode is highly specific to WDM‡, a few definitions and clarifications are in order:

- **DPC**: Deferred Procedure Call. In WDM‡ an **ISR** can queue a DPC to a FIFO queue to do time-critical work on its behalf. DPCs execute after all ISRs but before paging and threads. This is similar to the Immediate queue for ISR "Bottom Halves" in Linux†.
- **DriverEntry**: This function is called at driver load time and performs all driver initialization.
- **IRP**: I/O Request Packet. Each user mode call to a Win32 driver interface function (e.g., Read) generates an IRP that is passed to the appropriate driver routine.
- **IRP->AssociatedIrp.SystemBuffer**: This is used to transfer data to/from the user mode application. We abbreviate it as **IRP->ASB** and pretend that it is of type `LARGE_INTEGER`.
- **ISR**: Interrupt Service Routine. In the WDM‡ paradigm, ISRs queue **DPC**s to do work on their behalf.
- **PIT**: Programmable Interval Timer. PC hardware timer. By default on Windows 98 and NT it fires at a frequency of 67 to 100 Hz (10 to 15 ms. period). We reset it to 1 KHz (1 ms. period).
- **Real-time Priority**: WDM‡ has 16 real-time priorities, 16 through 31. 24 is the default.
- **Single shot timer**: An OS timer that fires only once. NT 4.0 added periodic OS timers.
- **Synchronization Event**: An event that auto-clears after a single wait is satisfied. Contrast with a **Notification Event**, which satisfies all outstanding waits, as do Unix† kernel events.

2.2.1 DriverEntry Pseudocode

```
Create a single shot timer gTimer.
Create a Synchronization Event gEvent.
Create a kernel mode thread executing
   LatThreadFunc() (section 2.2.4).
Initialize global variable ghIRP
   shared by thread, driver functions.
Set PIT interrupt interval to 1 ms.
```

2.2.2 Driver I/O read Pseudocode

```
Procedure LatRead(IRP) {
   GetCycleCount(&IRP->ASB[0])
// The PIT ISR will enqueue
// LatDpcRoutine in the DPC queue
   KeSetTimer (gTimer,
               ARBITRARY_DELAY,
               LatDpcRoutine)
}
```

2.2.3 Timer DPC Pseudocode

```
// This is called by the kernel when
// the DPC is dequeued and executed
Procedure LatDpcRoutine(IRP) {
   GetCycleCount(&IRP->ASB[1])
   ghIRP = IRP
   KeSetEvent(gEvent)
}
```

2.2.4 Thread Pseudocode

```
Procedure LatThreadFunc() {
   KeSetPriorityThread(
      KeGetCurrentThread(),24);
   loop (FOREVER) {
      WaitForObject(gEvent,FOREVER)
      GetCycleCount(&ghIRP->ASB[2])
// This completes the read, sending
// the data to the user mode app
      IoCompleteRequest(ghIRP)
      ghIRP = NULL
   } /* loop */
}
```

2.2.5 GetCycleCount code

Because not all versions of the Visual C++[†] inline assembler recognize the Pentium RDTSC instruction, the following function is provided.

```
//  Name: GetCycleCount
//  Purpose: Read the Pentium® cycle
//           (timestamp) counter
//  Context: Called by driver to get
//           current timestamp
//
// Copyright (c) 1995-1998 by Intel
// Corporation. All Rights Reserved.
// This source code is provided "as
// is" and without warranty of any
// kind, express or implied.
// Permission is hereby granted to
// freely use this software for
// research purposes.
//
GetCycleCount(
   LARGE_INTEGER *pliTimeStamp) {
   ULONG Lo;
   LONG Hi;

   _asm {
      _emit 0x0f
      _emit 0x31
      mov Lo, eax
      mov Hi, edx
   } /* _asm */
   pliTimeStamp->LowPart = Lo;
   pliTimeStamp->HighPart = Hi;
   return;
} /* GetCycleCount */
```

2.3 Latency Cause Tool

The latency measurement tools, while eminently useful for assessing the ability of an OS to meet the latency tolerances of real-time applications and drivers, give little insight as to the causes of long latency. We desired to engage OS and possibly driver vendors in an effort to improve real-time performance. A significant impediment is that in a commercial environment we do not have access to source code for either OS or any of the drivers which could be responsible for one or more of the long latencies.

We began by modifying our thread latency tool to hook the Pentium processor Interrupt Descriptor Table (IDT) entry for the Programmable Interval Timer (PIT) interrupt. To do this we patch the PIT timer Interrupt Descriptor Table (Pentium Interrupt Dispatch Table) entry to point to our hook function. The hook function updates a circular buffer with the current instruction pointer, code segment and time stamp and then jumps to the OS PIT ISR. We then modified the thread latency tool to report only latencies in excess of a preset threshold and to dump the contents of the circular buffer when it reported a long latency. Post mortem analysis produces a set of traces of active modules and, if symbol files are available[3], functions. In spite of the lack of source code the module+function traces are often quite revealing. Endo and Seltzer describe a similar technique for recording information on system state during long interactive event latencies as part of a proposed tool suite for Windows NT, but anticipate that OS source code will be needed for causal analysis [8].

3. Test Procedure

3.1 Application Stress Loads

For each application category we estimated how many hours per week constitute heavy use for applications belonging to that category. As explained below, based on the estimates we estimated how many hours of data we needed to collect using each stress application. To maximize our breadth of application coverage and improve reproducibility some stress applications are actually benchmarks. These benchmarks are driven by Microsoft Test (MS-Test) at speeds much faster than possible for a human, enabling us to collect data over a shorter period of time than would otherwise have been the case. We collected data for periods of hours, capturing events that occur at frequencies as low as 1 in 100,000 in statistically significant numbers.

[3] OS symbols are available with a subscription to the Microsoft Developer's Network (MSDN) [21].

3.1.1 Office Applications

To represent the class of office applications we used the Business Winstone 97 benchmark [22], which executes eight business productivity applications spanning three categories of business computing:

- Database: Access† 7.0, Paradox† 7.0
- Publishing: CorelDRAW† 6.0, PageMaker† 6.0, PowerPoint† 7.0
- Word Processing and Spreadsheet: Excel 7.0, Word 7.0, WordPro† 96

Each application is installed via an InstallShield†[4] script, run at full speed through a series of typical user actions and then uninstalled. On initial launch Winstone performs a number of hardware checks (e.g., interrogation of the floppy drive) which cause a marked spike in observed OS latencies. We therefore launched Winstone, then started our latency measurement tools and finally launched the benchmark.

The Business benchmark is driven by MS-Test at speeds in excess of human abilities to type and click a mouse. As Endo *et. al.* observe, this results in an unnaturally time-compressed sequence of user input events that should not occur in normal use, resulting in abnormally large batched requests for OS services [7]. We agree with Endo et. al. that these batched requests may be optimized away by the OS, resulting in a lower overall system load during the benchmark than during equivalent human user activity. Nevertheless, we note that long spurts of system activity will still occur because of, for example, file copying, both explicit and implicit (e.g., "save as"). In our experience this type of extended system activity is much more likely to impact response to interrupts, causing long latencies, than any of the batched requests discussed by Endo, et. al. might cause individually or collectively were they not batched. Since we are only using the Winstone benchmark to impose load, we exploit this time-compression to collect data for a shorter period of time.

Data as to how long a “typical” user would take to execute the Business Winstone 97 benchmark input sequence are unavailable [23], but we can derive a conservative lower bound to the compression ratio under very weak assumptions. To do this we assume that a typing speed of 120 5-character words per minute (or about 1 character every 100 milliseconds) is the upper limit of sustainable human input speed. Based on the default PC clock interrupt rate of 67 to 100 Hz. (see section 2.2) it is clear that Winstone can drive input at least ten times as quickly as a human, even without compensating for the complete absence of "think time" [20] during the benchmark. Thus we estimate that Business Winstone 97 running continuously will produce at least as much system stress in 4 hours as a heavy user will produce in a 40-hour work week.

3.1.2 Workstation applications

To represent the class of workstation applications we used the High-End Winstone 97 automated benchmark [22], which executes six workstation applications spanning three categories of workstation computing:

- Mechanical CAD: AVS† 3.0, Microstation† 95
- Photoediting: Photoshop† 3.0.5, Picture Publisher† 6.0, P-V Wave† 6.0
- S/W Engineering: Visual C++ 4.1 Compiler

As with the Business benchmark, each application is installed via an InstallShield script, run at full speed through a series of typical user actions and then uninstalled. Again, we first launched Winstone, then started our latency measurement tools and finally launched the benchmark.

Workstation applications are inherently more stressful than business applications, and are CPU, disk or network bound (i.e., not waiting on user I/O) more of the time than business applications. We therefore assumed a more conservative 5 to 1 ratio of MS-Test input speed to human input speed. Thus we estimate that 6 hours of continuous testing will produce as much system stress as a heavy user will produce in one work week of 30 hours, assuming engineers spend 2 hours daily using non-engineering applications such as email.

3.1.3 Multimedia Applications

We divided the class of multimedia applications into two subcategories: 3D games and Web browsing with enhanced audio/video. In order to compare apples to apples, we limited ourselves to 3D games that run on Windows 95/98 and Windows NT. Two were selected: Freespace† Descent† and Unreal†. Since game demos are essentially canned sequences of game play, we do not assume any speedup when collecting our 3D game data. We estimate that game enthusiasts play on the order of 2 to 3 hours per day, 4 to 6 days per week, concluding that 12.5 hours of data will capture a week of game play by an enthusiast.

Web browsing is dominated by download times. With a modem on a regular phone line a heavy user is bandwidth limited. By using an Ethernet LAN

[4] InstallShield is a standard Windows application that installs and configures other applications. It is driven by scripts written by the software vendor.

connection, downloading occurs at speeds far in excess of those achievable on a regular phone line. As a result, the system is stressed more than would actually occur during normal usage, and it is not necessary to collect data for as long a period of time as would otherwise be the case. Assuming conservatively a 10 to 1 ratio of 10 MBit Ethernet download speed to regular phone line download speed, we estimate an overall 4 to 1 ratio, given that the user also spends time reading Web pages, listening to audio and video clips, etc. We estimate that a heavy user browses the Web about 3 to 4 hours per day, 7 days per week. We conclude that 8 hours of data while browsing with an Ethernet network connection should capture about a week of Web browsing over a regular phone line by a heavy user.

We split our Web testing time between downloading and viewing files and downloading and playing audio and video clips. We used both Netscape Communicator† and Internet Explorer† 4.0 (IE4). The first half consisted of repetitions of the following sequence:

- Browse with Netscape Communicator to www.irs.ustreas.gov/prod/cover.html, view several tax forms with Adobe Acrobat† Reader and download instructions for each form.
- Browse with Netscape Communicator to www.cse.ogi.edu and view a random (typically short) postscript TechReport with Ghostview†.
- Browse with IE4 to www.intel.com and view a processor manual with Adobe Acrobat Reader.
- Browse to www.research.microsoft.com with IE4 and view a technical report with Word 97.

In the second half we first browsed with Netscape Communicator to www.real.com and played news and music clips using RealPlayer†. We then browsed with IE 4.0 to Siskel and Ebert's Web site and played movie reviews using Shockwave†.

3.2 Test System Configuration

To minimize the impact of legacy software and hardware, such as Windows 98 drivers for devices on the old slow ISA (Industry Standard Architecture) bus, we configured our system exclusively with PCI (Peripherial Component Interconnect) bus and USB (Universal Serial Bus) devices. To do this we disabled the ISA Plug and Play Enumerator and motherboard ISA audio devices in the Control Panel System Properties menu (98) and Devices menu (NT).

Table 2 gives the full system configuration, with items that differ between the two systems shaded. The file systems used were different but reflect the "typical" file system for each OS. The audio solutions were of necessity different because Windows NT 4.0 does not support USB, while Windows 98 did not at the time fully support WDM‡ audio drivers on PCI sound cards. A key point, easily overlooked, is that both OSs have been configured to use DMA drivers for the IDE devices (hard drive and CD-ROM). For Windows 98 this is a user configurable option accessible via the System icon on the Control Panel. For Windows NT 4.0 we used the Intel PIIXBus Master IDE Driver.

OS version	Windows NT 4.0	Windows 98
Optional OS Components	Service Pack 3 w. 11/97 rollup hotfix	Plus! 98† Pack w/o opt. Virus Scanner
Filesystem	NTFS	FAT32
IDE Driver	Intel PIIXBus Master IDE Drvr ver. 2.01.3	Default with DMA set ON
Processor & speed	Pentium® II 300 MHz	Pentium II 300 MHz
Motherboard	Atlanta (Intel 440 LX)	Atlanta (Intel 440 LX)
BIOS ver.	4A4LL0X0.86A.0 012.P02	4A4LL0X0.86A.0 012.P02
Memory	32 MB SDRAM	32 MB SDRAM
Hard Drive	Maxtor DiamondMax† 6.4 GB UDMA	Maxtor DiamondMax 6.4 GB UDMA
CD-ROM Drive	Sony CDU 711E 32x	Sony CDU 711E 32x
AGP Graphics	ATI Xpert@Work	ATI Xpert@Work
Resolution	1024 x 768 x 32bit	1024 x 768 x 32 bit
3D games	800 x 600 x 32 bit	800 x 600 x 32 bit
Audio solution	Ensoniq PCI sound card with Prosonic speakers	Phillips DSS 350 USB speakers
Network (only Web Browsing)	Intel EtherExpress™ Pro 100 PCI NIC	Intel EtherExpress Pro 100 PCI NIC

Table 2: Test System Configuration

4. Results

4.1 WDM‡ scheduling hierarchy

For convenience, we abstract the services provided by the Windows Driver Model (WDM‡) into an OS scheduling hierarchy as shown below. Each level of the OS scheduling hierarchy is fully preemptible by the

level(s) with lower numbers on Windows NT, but there are complications on Windows 98 since the legacy Windows 95 schedulers continue to exist[5]. In pure WDM‡ (e.g., on Windows NT) the hierarchy is as follows:

1. Interrupt Service Routines (ISRs)
   Execute at IRQLs from `DIRQL` to `HighLevel`
2. WDM‡ Deferred Procedure Calls (DPCs)
   FIFO/LIFO queue, three priorities (`High`, `Medium`, `Low Importance`), but DPCs cannot preempt other DPCs
3. Real-Time Priority Threads
   Timesliced, execute at Win32 priorities 16 through 31, can raise IRQL from PASSIVE (lowest) to arbitrarily high levels (i.e., block interrupts)
4. Normal Priority Threads
   Timesliced, execute at Win32 priorities 1 through 15, can raise IRQL

Our investigations focused on levels one through three of the scheduling hierarchy. We present data for the following:

- ISR latency for the PIT (timer) ISR.
- DPC latency for a "Medium Importance" WDM‡ DPC enqueued by the PIT ISR. We term this the PIT DPC.
- High real-time priority kernel mode thread latency for a Win32 priority 28 kernel mode thread signaled by from the PIT DPC.
- Medium real-time priority kernel mode thread latency for a Win32 priority 24 kernel mode thread signaled from the PIT DPC.

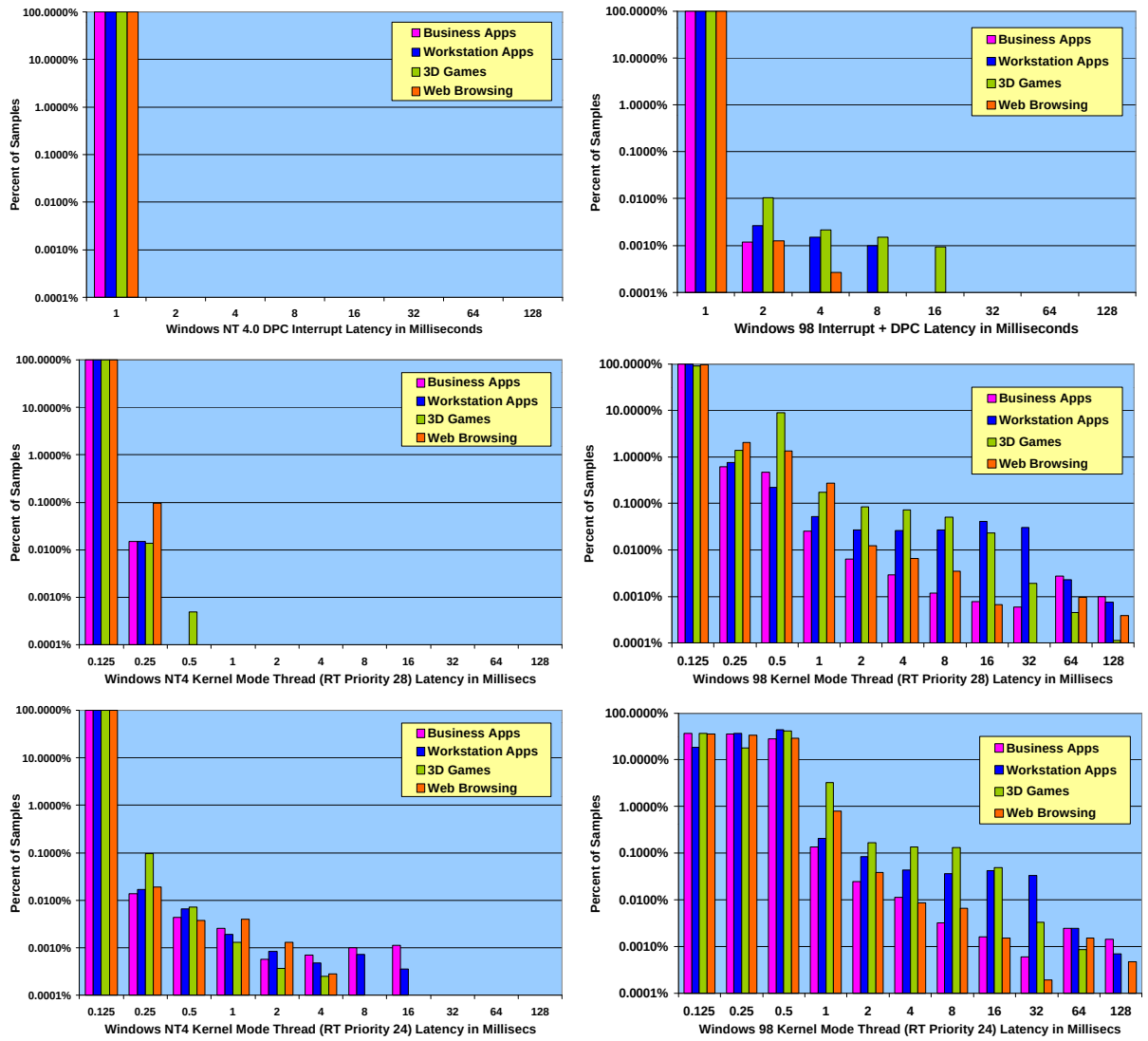

Figure 4: Measured Interrupt and Thread Latencies under Load on Windows NT 4.0 and Windows 98

4.2 Overall WDM‡ Latency Profile

Windows 98 OS latency distributions are highly non-symmetric, with a very long tail on one side, and thus bear little resemblance to a normal distribution. In order to accurately show the latency tail we present our data as log-log plots in Figure 4. We present histograms comparing the latency from a timer interrupt (Programmable Interval Timer, whose ISR runs at extremely high IRQL) to the corresponding WDM‡ DPC and from WDM‡ DPCs to real-time high (28) and default (24) priority threads waiting on WDM‡ synchronization events. We present this data broken out by application workload so that the latency profiles under different workloads on the same OS can be compared.

For NT 4.0 there is almost no distinction between DPC latencies and thread latencies for threads at high real-time priority. The WDM‡ "kernel work item" queue is serviced by a real-time default priority thread, which accounts for the large difference between high and default priority threads under NT 4.0. For Windows 98, on the other hand, there is an order of magnitude reduction in worst-case latencies that a driver obtains by using WDM‡ DPCs as opposed to real-time priority kernel mode threads. NT real-time high priority threads and DPCs exhibit worst-case latencies which are an order of magnitude lower than those of Windows 98 DPCs and Windows NT real-time default priority threads. This view of system performance contrasts sharply with the view one obtains using traditional throughput-based benchmarks.

To verify that throughput-based benchmarks would not reveal the variation in real-time performance that we see in our plots, we ran the Business Winstone 97 benchmark on Windows 98 and on Windows NT 4.0 using our system configurations as specified in Table 2. While reporting requirements (and space here) prevent us from publishing exact figures, the average delta between like scores was 10% and the maximum delta was 20%. In contrast, from a real-time standpoint, we conclude that NT 4.0 exhibits latency performance at least an order of magnitude superior to that of Windows 98 and, for kernel mode high real-time priority threads, two orders of magnitude better.

	Observed Hourly, Daily and Weekly Worst Case Windows 98 Latencies (in ms.)											
	Office Apps			**Workstation Apps**			**Recent 3D Games**			**Web Browsing**		
OS Service	**Max Per Hr**	**Max Per Day**	**Max Per Wk**	**Max Per Hr**	**Max Per Day**	**Max Per Wk**	**Max Per Hr**	**Max Per Day**	**Max Per Wk**	**Max Per Hr**	**Max Per Day**	**Max Per Wk**
H/W Int. to S/W ISR	<1.0	1.4	1.6	2.2	5.6	6.3	8.8	9.7	12.2	1.1	1.7	3.5
S/W ISR to DPC	+ 0.1	+ 0.1	+ 0.4	+ 0.5	+ 0.5	+ 0.6	+ 0.9	+ 2.1	+ 2.1	+ 0.2	+ 0.3	+ 0.3
H/W Interrupt to DPC	1.0	1.5	2.0	2.7	6.1	6.9	9.7	12	14	1.3	2.0	3.8
DPC to kernel RT thread (High Priority)	+ 1.6	+ 5.2	+ 31	+ 21	+ 24	+ 24	+ 35	+ 46	+ 70	+ 14	+ 68	+ 80
H/W Int. to kernel RT thread (High Priority)	2.6	6.7	33	24	30	31	45	58	84	15	70	84
DPC to kernel RT thread (Med. Priority)	+ 3.1	+ 6.7	+ 31	+ 21	+ 23	+ 24	+ 36	+ 47	+ 70	+ 51	+ 68	+ 80
H/W Int. to kernel RT thread (Med. Priority)	4.1	8.2	33	24	29	31	46	59	84	52	70	84

Table 3: Windows 98 Interrupt and Thread Latencies with no Sound Scheme on a PC 99 Minimum System

4.3 Windows 98 Detailed Latency Profile

Because Windows 98 has been recently released, we present a more detailed latency profile in tabular form. Since our focus is on means of forecasting realizable application worst-case behavior, we are especially concerned with the worst-case latency and with comparative measures of the "thickness" of the tail of the latency distribution. We therefore characterized the distributions for Windows 98 in terms of three expected worst case values: hourly, daily and weekly. The hourly value is for continuous usage, whereas the daily and weekly values do *not* represent continuous 24 or 168 hour usage, but rather expected average daily and weekly use by a heavy user. The usage patterns are described in detail in section 3.1. To briefly recap, for the Office and Workstation applications a "day" is 6 to 8 hours long and a (work) week has 5 "days", while for the 3D games and Web Browsing, a "day" is only 3 to 4 hours but a (consumer) week has 7 "days".

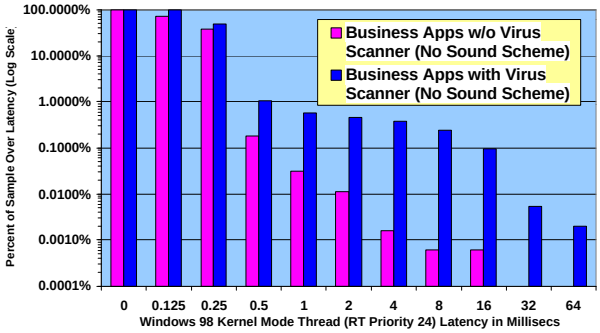

Figure 5: Effect of the Virus Scanner on High Priority Real-Time Thread Latency

During the course of our investigation of Windows 98 we discovered the optional Plus! 98[†] Pack Virus Scanner and the Windows sound schemes had significant impacts on thread latency. The Virus Scanner is particularly egregious in this regard and the data for Windows 98 presented in Figure 4 is for an installation without the virus scanner. Figure 5 presents data with the virus scanner installed and active, but with no sound scheme, and it can be seen that with the virus scanner 16 millesecond thread latencies occur over two orders of magnitude more frequently. Assuming that long latencies are uniformly distributed over time, with the virus scanner on we would expect a 16 millisecond thread latency about every 1000 times that our thread does a `WaitForSingleObject` on a WDM[‡] event, or roughly every 16 seconds for an audio thread with a 16 millisecond period. In contrast, without the virus scanner (and with no sound scheme) we would expect a 16 millisecond thread latency only about once in 165,000 waits, or roughly once every 44 minutes for the same audio thread. Intel's audio experts did not find it surprising that the virus scanner had this effect; they had remarked for some time that the virus scanner causes breakup of low latency audio.

4.4 Windows 98 Thread Latency Causes

As we have seen in the previous section, our tools can successfully predict Quality of Service problems that impact the end user's experience. More importantly, by defining a simple metric that is easy to automatically detect at run-time, our tools and techniques give us the ability to determine the code paths that are responsible for this behavior. This, of course, greatly increases the probability of obtaining a fix from the developers, who now receive a bug of the form "audio breaks up when we turn on your application", but who could receive a bug report with one or more function call traces.

Before discussing more specific results, some background is in order. The Windows 98 Plus! Pack makes a number of sound schemes available. These produce a variety of user-selectable sounds upon occurrence of various "events". These "events" range from typical things, such as popup of a Dialog Box to the more esoteric, such as traversal of walking menus (i.e., EVERY time a submenu appears). As mentioned above, Winstone uses MS-Test to drive applications at greater than human speeds, which results in a lot of sounds being played. During our testing we restricted ourselves to the default and "no sound" sound schemes.

```
Analysis of latency episode number 0
   1 samples in VMM function @KfLowerIrqI
   1 samples in NTKERN function
     _ExpAllocatePool
   1 samples in SYSAUDIO function
     _ProcessTopologyConnection
   2 samples in VMM function
     _mmCalcFrameBadness
   ------------------------------------------------
   5 total samples in episode

Analysis of latency episode number 1
   1 samples in SYSAUDIO function
     _ProcessTopologyConnection
   2 samples in VMM function
     _mmCalcFrameBadness
   2 samples in VMM function _mmFindContig
   1 samples in KMIXER function unknown
   ------------------------------------------------
   6 total samples in episode
```

Table 4: Thread Latency Cause Tool Output, Windows 98 w. Biz Apps, Default Sound Scheme

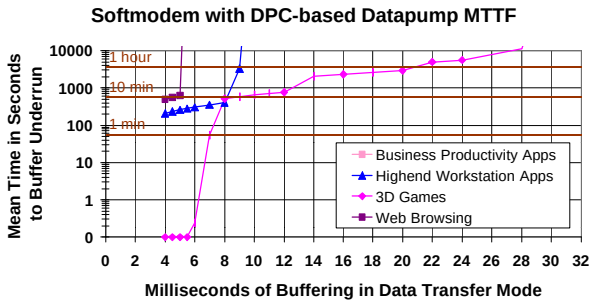

Figure 6: Mean Time to Buffer Underrun for a DPC-based Datapump of a Soft Modem on Windows 98 in Data Transfer Mode

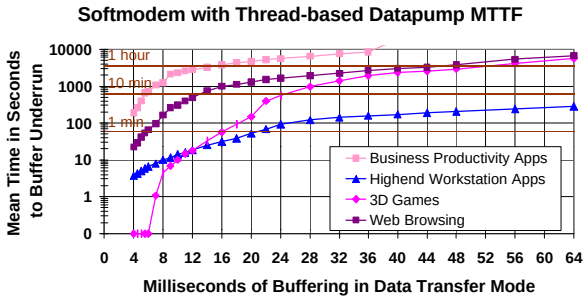

Figure 7: Mean Time to Buffer Underrun for a Thread-based Datapump of a Softmodem on Windows 98 in Data Transfer Mode

Table 4 presents two brief sample traces from an investigation into the causes of long thread latencies during the Winstone Business benchmark when the default Windows sound scheme was enabled. From the traces we see that with the default sound scheme on (presumably the normal state of affairs) two moderately long thread thread latencies were observed. During both a SysAudio function ProcessTopologyConnection was active and the OS appears to have been allocating contiguous memory, possibly in order to accommodate "bad", possibly misaligned, audio frames. We can also see that at least part of this operation is taking place at raised IRQL, which would explain, for example, why both priority 24 and 28 kernel mode threads are affected. Further analysis of these episodes is best left to the authors of the code, but the reader will see that this information can be of great use.

5. Analysis

As an example of how detailed latency data can be used to forecast quality of service for multimedia applications and low latency drivers, we present a brief analysis of soft modem quality of service as a function of the size and the number of buffers (and thus, the allowable latency in servicing the buffers). Here we briefly discuss the Mean Time To Failure (MTTF) plots that we present in the next section. The plots are derived from our tables of latency data by calculating the slack time for each amount of buffering (i.e., $t * (n-1) - c$, where n is the number of buffers, t is the buffer size in milliseconds and c is the compute time for 1 buffer.). This number is used to index into the latency table to determine the frequency with which such latencies occur, and this frequency is divided by an approximation of the cycle time (for simplicity, $(n-1) * t$). Thus the calculation is strictly accurate only for double buffered implementations but is reasonably accurate if n is small.

5.1 Soft Modem Quality of Service

We now present an analysis of soft modem quality of service from a timing standpoint. Figures 6 and 7 show the mean time to failure (i.e., buffer underrun) for the datapump of a soft modem as a function of the amount of buffering in the datapump. We have estimated that the datapump requires 25% of a system with a 300 MHz Pentium II processor during data transmission mode, which is a conservative (high) estimate. To interpret the figures, calculate the total buffering in the datapump. For example, for a triple buffered implementation using 6 millisecond buffers, using Figure 6, we can see that with 12 milliseconds of buffering the Windows 98 DPC-based datapump will miss a buffer roughly once every 12 to 15 minutes (720-900 seconds) while playing an "average" 3D game. With 10 millisecond buffers triple buffered (i.e., 20 milliseconds of buffering), however, the Windows 98 DPC-based datapump would average an hour (3600 seconds) between misses while playing an "average" 3D game[6]. Similarly, a Windows 98 thread-based datapump that uses high-priority, real-time kernel mode threads will require about 48 milliseconds of latency tolerance (e.g., four 16 millisecond buffers) in order to average an hour between misses while playing an

[6] Note that missed buffers need not be catastrophic since design techniques exist whereby the datapump can arrange for hardware to automatically transmit a dummy buffer in the event that a buffer is missed. Such a dummy buffer can be made indistinguishable from line noise or other bit errors to the receiving modem and will result in either a retransmission request at the link layer or a dropped frame. In either case the overall impact on modem connection quality can be kept manageably small relative to the number of buffers which will be damaged due to line noise. Similar considerations (e.g., error correcting codes) apply to other classes of low latency real-time drivers.

"average" 3D game. Since the worst case latencies for Windows NT are uniformly below the minimum modem slack time of 3 milliseconds (= cycle time of 4 ms. – 1 ms. of computation), we forgo the analysis.

5.2 Schedulability Analysis on a Non-Real-Time OS

As we noted above, the MTTF plots in the previous section assume implicitly that double buffering is used, but are reasonably accurate for triple buffering. A more accurate method of making the assessment, including taking into account other "lower level" (i.e., higher priority) drivers that were not present on the measured, is described in our earlier work on Schedulability Analysis [4]. Briefly, the procedure is to use the information from Table 3 as input to a Schedulability Analysis tool. One chooses the worst case latency as a function of the permissible error rate: for example, one dropped buffer every five or ten minutes for low latency audio (video teleconferencing), one dropped buffer per hour for a soft modem, or one dropped buffer per day for a more high-reliability device. The worst-case is then used to calculate a "pseudo worst-case" which is input into a standard schedulability analysis tool such as PERTS [16]. This technique amortizes the overhead of an unusually long latency over a number of "average" latencies to enable analysis techniques designed for deterministic real-time OSs to be applied on a general purpose OS.

6. Conclusions

We have presented a metric for evaluating the real-time performance of non-real-time OSs and platforms. This metric captures an aspect of performance that is completely missed by standard batch and throughput-based benchmarking techniques commonly in use today. The techniques that we have described are destined to grow in importance as emerging workloads such as audio, video and other multimedia presentations are ever more widely deployed and as low latency hard real-time drivers are migrated off of special purpose hardware onto host processors. This process is already well advanced, with applications such as soft MPEG and DVD already under development and soft audio and soft modems already being routinely deployed by vendors of low-cost personal computers. It is likely that this trend will accelerate in the future, further increasing the importance of the latency metric.

Our analysis revealed that the two implementations of the Windows Driver Model, although functionally compatible, are very different in their timing behavior. Using the interrupt and thread latency metrics we are able to characterize the behavior that applications and drivers will experience on Windows 98 even before those applications and drivers are fielded. Our analysis indicates that many compute-intensive drivers will be forced to use DPCs on Windows 98, whereas on Windows NT high-priority, real-time kernel mode threads should provide service indistinguishable from DPCs for all but the most demanding low latency drivers. When one considers the difficulties of "interrupt-level" (i.e., WDM DPCs) driver development and the multitude of benefits obtained from using threads, it is apparent that analyses such as the one we have just presented will become increasingly important from a Software Engineering standpoint.

6.1 Future Work

We have completed evaluations of Windows 98 [5] and Windows NT 4.0 and continue to monitor the performance of Beta releases of Windows 2000[7]. We have also developed a tool that models periodic computation at configurable modalities (e.g., threads, DPCs) and priorities within modalities, and reports the number of deadlines that have been missed. With this tool we can model a soft modem and examine its impact on other kernel mode services. We will also be able to use the tool to validate our quality of service predictions in this paper and expect to report on this work at the conference.

In addition to this work, the latency cause analysis tool is under active development. First, we plan to enhance it to hook non-maskable interrupts caused by the Pentium II performance monitoring counters instead of the PIT interrupt. By configuring the performance counter to the CPU_CLOCKS_UNHALTED event we will be able to get sub-millisecond resolution during both thread and interrupt latencies. Second, we would like to enhance the hook to "walk" the stack so as to generate call trees instead of isolated instruction pointer samples. This would give much more visibility into the actual code paths under execution, greatly increasing the utility of the data.

7. Acknowledgements

Venugopal Padmanabhan and Dorian Salcz collected the lab data; their patience and precision are gratefully acknowledged. Dan Cox provided managerial support and encouragement. Others who assisted at various times include Dan Baumberger, Lyle Cool, Barbara Denniston, Tom Dingwall, Judi Goldstein, Jaya Jeyaseelan, Dan Nowlin, Barry O'Mahony, Jeyashree Padmanabhan, Jeff Spruiel, Jim Stanley, Cindy Ward,

[7] Windows 2000 was previously Windows NT 5.0.

and Mary Griffith. In addition, the support of Tom Barnes, Darin Eames and Sanjay Panditji of the Intel Architecture Labs is gratefully noted, as is the patience of the Program Committee shepherd, Margo Seltzer. Finally, Erik's wife, Judy, was exceptionally patient during the period when this paper was being written.

8. References

[1] A. Baker, *The Windows NT Device Driver Book*, Prentice Hall, Upper Saddle River, NJ. 1997.

[2] B. N. Bershad, R.P. Draves and A. Forin, "Using Microbenchmarks to Evaluate System Performance", *Proc. 3rd Wkshop on Workstation Operating Systems*, Key Biscayne, FL, April, 1992

[3] A.B. Brown and M.I. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture", *Proc. 1997 Sigmetrics Conf.*, Seattle, WA, June 1997.

[4] E. Cota-Robles, J. Held and T. J. Barnes, "Schedulability Analysis for Desktop Multimedia Applications: Simple Ways to Handle General-Purpose Operating Systems and Open Environments", *Proc. 4th IEEE International Conf. on Multimedia Computing and Systems*, Ottawa, Canada, June 1997. URL: http://developer.intel.com/ial/sm/doc.htm

[5] E. Cota-Robles, "Implications of Windows OS Latency for WDM Drivers", *Intel Developer's Forum*, Palm Springs, CA, September 1998.

[6] E. Cota-Robles, "Windows 98 Latency Characterization for WDM Kernel Drivers", *Intel Architecture Lab White Paper*, July 1998. URL: http://developer.intel.com/ial/sm/doc.htm

[7] Y. Endo, Z. Wang, J. B. Chen and M. I. Seltzer, " Using Latency to Evaluate Interactive System Performance", *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, October 1996.

[8] Y. Endo, M. I. Seltzer, "Measuring Windows NT—Possibilities and Limitations", *First USENIX Windows NT Workshop*, Seattle, WA, Aug, 1997.

[9] Intel Corp., *Intel Architecture Software Developer's Manual*, 3 volumes, 1996. URL: http://developer.intel.com/design/intarch/manuals/index.htm

[10] Intel Corp., *Pentium® II Processor Developer's Manual*, 1997. URL: http://developer.intel.com/design/PentiumII/manuals/index.htm

[11] International Telecommunication Union. Draft Reccomendation G.992.2, Splitterless Asymetrical Digital Subscriber Line (ADSL) Transceivers, 1998

[12] M Jones, D. Regehr. "Issues in using commodity operating systems for time-dependent tasks: experiences from a study of Windows NT", Proc. of 8th Intl Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 98), Cambridge, U.K., July 1998.

[13] D. I. Katcher, H. Arakawa, and J. Strosnider. "Engineering and Analysis of Fixed Priority Schedulers", *IEEE Transactions on Software Engineering*, 19(9), September, 1993.

[14] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour, *A Practioner's Handbook for Real-Time Analysis*. Kluwer, Boston, MA, 1993.

[15] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment", *JACM*, 20(1), Jan, 1973.

[16] J.W.S. Liu, J.L. Redondo, Z. Deng, T.S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha and W.K. Shih, " PERTS: A Prototyping Environment for Real-Time Systems". *Proc. of the IEEE Real-Time Systems Symposium*, December 1993.

[17] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis", *Proc. 1996 USENIX Technical Conf.*, San Diego, CA, January, 1996.

[18] Microsoft Corporation, "Windows 98 Driver Development Kit (DDK)" in Microsoft Developer Network Prof. Edition, Redmond, WA, 1998.

[19] J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware", *Proc. of the USENIX Summer Conf.*, June, 1990.

[20] B. Shneiderman, *Designing the User Interface: Strategies of Effective Human-Computer Interaction*. Addison-Wesley, Reading, MA. 1992

[21] D. A. Solomon, *Inside Windows NT Second Edition*, Microsoft Press, Redmond, WA. 1998.

[22] Ziff-Davis Corp., "Labs Notes: Benchmark 97: Inside PC Labs' Latest Tests", *PC Magazine Online*, Vol. 15, No. 21, December 3, 1996.

[23] Ziff-Davis Corp., Winstone webmaster, personal communication, 1998.