Using Latency to Evaluate Interactive System Performance

Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer
Harvard University

Using Latency to Evaluate Interactive System Performance

Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer
Harvard University, Division of Engineering and Applied Sciences
{yaz, zhwang, bchen, margo}@eecs.harvard.edu

Abstract

The conventional methodology for system performance measurement, which relies primarily on throughput-sensitive benchmarks and throughput metrics, has major limitations when analyzing the behavior and performance of interactive workloads. The increasingly interactive character of personal computing demands new ways of measuring and analyzing system performance. In this paper, we present a combination of measurement techniques and benchmark methodologies that address these problems. We introduce several simple methods for making direct and precise measurements of event handling latency in the context of a realistic interactive application. We analyze how results from such measurements can be used to understand the detailed behavior of latency-critical events. We demonstrate our techniques in an analysis of the performance of two releases of Windows NT and Windows 95. Our experience indicates that latency can be measured for a class of interactive workloads, providing a substantial improvement in the accuracy and detail of performance information over measurements based strictly on throughput.

1 Introduction

Benchmarks are used in computer systems research to analyze design alternatives, identify performance problems, and motivate improvements in system design. Equally important, consumers use benchmarks to evaluate and compare computer systems. Current benchmarks typically report throughput, bandwidth, or end-to-end latency metrics. Though often successful in rating the throughput of transaction processing systems and/or the performance of a system for scientific computation, these benchmarks do not give a direct indication of performance that is relevant for interactive applications such as those that dominate modern desktop computing. The most important performance criterion for interactive applications is responsiveness, which determines the performance perceived by the user.

In this paper, we propose a set of new techniques for performance measurement in which latency is measured in the context of a workload that is realistic, both in terms of the application used and the rate at which user-initiated events are generated. We present low-overhead methods that require minimal modifications to the system for measuring latency for a broad class of interactive events. We use a collection of simple benchmark examples to characterize our measurement methodology. Finally, we demonstrate the utility of our metrics by applying them in a comparison of Microsoft Windows NT versions 3.51 and 4.0 and Windows 95, using realistic interactive input to off-the-shelf applications.

The remainder of this section provides background on the problem of measuring latency, including the motivation for our new methodology based on an analysis of the current practice in performance measurement. Section 2 describes our methodology in detail. In Section 3, we discuss some of the issues in evaluating response time in terms of a user's experience. In Sections 4 and 5, we apply our methodology in a comparison of Windows NT versions 3.51 and 4.0 and Windows 95. Sections 6 and 7 discuss the limitations of our work and conclude.

1.1 The Irrelevance of Throughput

Most macrobenchmarks designed for interactive systems use throughput as the performance metric, measuring the time that the system takes to complete a sequence of user requests. A key feature of throughput as a performance metric is that it can be measured easily, given an accurate timer and a computation that will do a fixed amount of work. Throughput metrics measure system performance for repetitive, synchronous sequences of requests. However, the results of these benchmarks do not correlate directly with user-perceived performance—a critical metric when evaluating interactive system performance. The performance of many modern applications depends on the speed at which the system can respond to an asynchronous stream of independent and diverse events that result from interactive user input or network packet arrival; we call this *event handling latency.* Throughput metrics are ill-equipped to characterize systems in such ways. More specifically, throughput benchmarks fail to provide enough information for evaluating interactive system performance and make inappropriate assumptions for measuring interactive systems.

This work was sponsored in part by grants from Bellcore, Intel, the National Science Foundation, Sun Micsystems Laboratories, and the Sloan Foundation.

Information Lost

The results of throughput benchmarks are often reduced to a single number that indicates how long a system took to complete a sequence of events. Although this can provide information about the sum of the latencies for a sequence of events, it does not provide information about the variance in response time, which is an important factor in determining perceived interactive performance.

The insufficient detail provided by throughput benchmarks can also mislead designers trying to identify the bottlenecks of a system. Since throughput benchmarks provide only end-to-end measures of activity, system activity generated by low-latency events cannot be distinguished from that generated by longer-latency events, which have a much greater impact on user-perceived performance. Worse, if such a benchmark includes sufficiently many short-latency events, these short events can contribute significantly to elapsed time, leading designers to optimize parts of the system that have little or no impact on user-perceived performance. In an effort to compare favorably against other systems in throughput benchmarks, designers may even undertake such optimizations knowingly. In this case, bad benchmarking methodology hurts both system designers and end users.

In addition, user interfaces tend to use features such as blinking cursors and interactive spelling checkers that have (or are intended to have) negligible impact on perceived interactive performance, yet may be responsible for a significant amount of the computation in the overall activity of an application. Throughput measures provide no way to distinguish between these features and events that are less frequent but have a significant impact on user-perceived performance.

Inaccurate User Assumptions

Throughput benchmarks often drive the system by feeding user input as rapidly as the system can accept it—equivalent to modeling an infinitely fast user. Such an input stream is unrealistic and susceptible to generating misleading results. One of the sources for such errors is batching. Client-server systems such as Windows NT and the X-Window system batch multiple client requests into a single message before sending them to the server. This reduces communication overhead and allows the server to apply optimizations to the request stream, such as removing operations that are overridden by later requests. Although batching improves throughput, it can have a negative effect on the responsiveness of the system.

When a benchmark uses an uninterrupted stream of requests, the system batches requests more aggressively to improve throughput. Measurement results obtained while the system is operating in this mode are meaningless; users will never be able to generate such an input stream and achieve a similar level of batching in actual use. Disabling batching altogether is sometimes possible but does not fully address the problem. An ideal test input should permit a level of batching that is likely to occur in response to real user input.

Overall, throughput measures provide an indirect rather than a direct measure of latency, and as such they can give a distorted view of interactive performance. An ideal benchmarking methodology will drive the system in the same way that real users do and give designers a correct indication as to which parts of the system are responsible for delays or user-perceptible latency. Obtaining such figures requires that we drive the system using an input stream that closely resembles one that an interactive user may generate and more importantly, an ability to measure the latency of individual events.

1.2 Related Work

Most of the benchmarks currently used for computer performance measurement can be categorized as either microbenchmarks or batch benchmarks. Microbenchmarks are commonly used to measure the performance of relatively simple, specific events [7,10]. They can be used to measure latency (e.g., 7.7 µs per IPC) or throughput (e.g.,129,000 local IPCs per second). Some examples of current popular microbenchmark suites are Winbench [14], the Bytemarks [3], and *lmbench* [7]. The shortcomings of microbenchmarks for general performance analysis are well documented [2]. The fundamental problem is that the computation they perform is not useful in and of itself and, as such, they fail to accurately reflect system behavior for any realistic situation.

Batch benchmarks address the shortcomings of microbenchmarks by measuring the time to run a complete, non-trivial computation from start to finish. Batch benchmarks are the dominant way of measuring system performance [1,13,14]. Portable benchmarks such as the SPEC95 suite are based on batch computations with non-interactive input. The portability requirement means that it must be possible to adapt these benchmarks to use new system APIs or to accommodate the idiosyncrasies of a given API implementation. To meet these requirements, the SPEC benchmarks are distributed in source form. The central problem with benchmarks such as the SPEC suites is that, although they may represent a realistic batch load, they fail to model the behavior of an interactive user [8].

Benchmarks such as Winstone, BAPCo SYSmark NT and BAPCo SYSmark 32 sacrifice portability in order to use popular interactive applications. Winstone is specific to PC-compatible hardware running Windows 95 or Windows NT. SYSmark NT provides somewhat more hardware portability than the Winstone suite, providing native executables for non-x86 systems that run Windows NT. Although these benchmark suites use interactive applications and simulated interactive input, they report performance in terms of throughput metrics and suffer from the problems presented in Section 1.1. Nontrivial batch computations are sometimes included in these workloads when they are consistent with realistic usage of the application. Examples are circuit board layout in MAXEDA (BAPCo SYSmark NT) and database queries in Microsoft Access (Winstone 95). Although these benchmarks use realistic applications, the input streams used to drive them model an infinitely fast user.

BAPCo SYSmark for File Servers, Ziff-Davis Netbench, SPEC SFS (LADDIS), and the TPC benchmarks measure performance of a server under client load. Both Netbench and SPEC SFS measure file server performance, and compute their results using the latency observed by clients. The intent of TPC-C is to measure on-line transaction processing performance. It reports performance in terms of complete transactions processed per minute. TPC-D is designed to measure decision-support database performance. TPC-D specifies one metric (QppD) that measures “raw performance” or latency, and another metric (QthD) for throughput.[1] Several of these benchmarks use client-observed latency in reporting performance, but only for client requests to a remote server. Our goal is to measure latency for a more general class of events.

A prior study compared Windows NT, Windows for Workgroups 3.11, and NetBSD to explore how differences in system structure affect overall performance [4]. The study used microbenchmarks to measure the latency of simple events and application workloads. Although some of the application workloads were based on interactive applications, the applications were driven by an uninterrupted input stream, and the results were reported in terms of throughput. There was no attempt to measure interactive performance directly. Because of the requirement that the workloads run on a Unix system, the study included no popular PC applications.

Two prior studies influenced our work. Shand used a free-running counter in a tight loop to measure the latency of interrupts [11]. This methodology is similar to ours in that it measures the computation of interest by detecting lost time. We generalize the prior work by looking beyond interrupt handling to a broader range of events types, by avoiding the need for special-purpose hardware, and by looking at interactive events rather than the latency of interrupt handlers. Second, our techniques for visualizing latency were influenced by the work of O‘Toole et al. [9] on reducing the pause times for garbage collection.

2 Methodology

Our methodology must provide the ability to measure the latency of individual events that occur while executing realistic interactive workloads. This poses the following set of new challenges:

- Interactive events are short in duration relative to the timer resolution provided by clock APIs in modern operating systems such as Windows and UNIX. Whereas a batch workload might run for millions of timer ticks, many interactive events last less than a single timer interval.
- Under realistic load, there will often be only a fraction of a second between interactive events in which to record results and prepare for the next measurement. Therefore the measurement scheme must have quick turnaround time.
- Perhaps the most challenging problem is collecting the requisite data without access to the source code of the applications or operating system. With source code, it is straightforward to instrument an application to generate timestamps at the beginning and ending points of every interactive event, but this is time consuming at best and not possible given our goal of measuring widely-available commercial software.

Analyzing interactive applications is just as challenging as measuring them. The time during which an application is running can be divided into *think time* and *wait time.* Think time is the time during which the user is neither making requests of the system nor waiting for the system to do something. Wait time is the time during which the system is responding to a request for which the user is waiting. Not all wait time is equivalent with respect to the user; wait time intervals shorter than a user’s perception are irrelevant. We call these classes of wait time “unnoticeable.” A good example of unnoticeable wait time is the time required to service a keystroke when a user is entering text. Although the system may require a few tens of milliseconds to respond to each keystroke, such small “waits” will be unnoticeable, as even the best typists require approximately 120 ms per keystroke [12]. Distinguishing between wait time and

1. TPC-A and TPC-B were declared obsolete by the Transaction Processing Council in June of 1995.

think time is non-trivial, and the quantity and distribution of wait time is what the user perceives as an application's responsiveness. Our measurement methodology must help us recognize the wait time that is likely to irritate users.

In the following sections, we describe the combination of tools and techniques that we use to measure and identify event latency.

2.1 Experimental Systems

We ran our experiments on a personal computer based on an Intel Premiere II motherboard, with the Intel Neptune chip set and a 100 MHz Pentium processor. Our machine was equipped with a 256KB asynchronous SRAM Level 2 cache, 32 MB of RAM, and a Diamond Stealth 64 DRAM display card. We used a dedicated 1GB Fujitsu disk (model M1606SAU) for each of the operating systems we tested. These disks were connected via a NCR825-based SCSI II host adapter. Both Windows NT systems used a NTFS file system, while the Windows 95 system used a FAT file system. The significant differences between Windows NT 3.51 and NT 4.0 are a change in GUI (NT 3.51 uses the traditional Windows GUI, while NT 4.0 uses a GUI similar to that of Windows 95) and the movement of some Win32 components into the kernel.

2.2 The Pentium Counters

The Intel Pentium processor has several built-in hardware counters, including one 64-bit cycle counter and two 40-bit configurable event counters [5]. The counters can be configured to count any one of a number of different hardware events (e.g., TLB misses, interrupts, or segment register loads). The Pentium counters make it possible to obtain accurate counts of a broad range of processor events. Although the cycle counter can be accessed in user or system mode, the two event counters can only be read and configured from system mode.

2.3 Idle Loop Instrumentation

Our first measurement technique uses a simple model of user interaction to measure the duration of interactive events. In an interactive system, the CPU is mostly idle. When an interactive event arrives, the CPU becomes busy and then returns to the idle state when the event-handling is complete. By recording when the processor leaves and returns to an idle state, we can measure the time it takes to handle an interactive event, and the time during which a user might be waiting.

The lack of kernel source code prevents us from instrumenting the kernel to identify the exact times at which the processor leaves or enters the idle loop. Instead, we replace the system's idle loop with our own low-priority process in each of the operating systems.[2] These low-priority processes measure the time to complete a fixed computation: N iterations of a busy-wait loop. The instrumentation code logs the time required by the loop. The pseudo code is as follows:

```
while (space_left_in_the_buffer) {
    for (i = 0; i < N; i++)
        ;
    generate_trace_record;
}
```

We select the value of N such that the inner loop takes one ms to complete when the processor is idle. In this way we generate one trace record *per millisecond of idle time*. If the processor is taken away from the idle loop, the loop takes longer than one ms of elapsed time to complete. Any non-idle time manifests itself as an elongated time interval between two trace records. The larger we make N, the coarser the accuracy of our measurements; the smaller we make N, the finer the resolution of our measurements but the larger the trace buffer required for a given benchmark run.

We wrote and measured a simple microbenchmark to demonstrate and validate this methodology. It uses a program that waits for input from the user and when the input is received, performs some computation, echoes the character to the screen, and then waits for the next input. We measured the time it took to process a keystroke in two ways. First, we used the idle loop method described above to measure the processing time. Figure 1 shows the times at which the samples were collected. For the sake of clarity only a few samples are

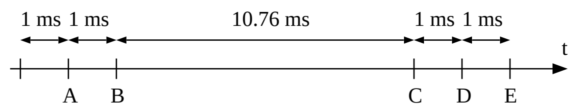


Figure 1. Validation of Idle-Loop Methodology. The system spent one ms collecting each of samples A, B, D, and E but spent 10.76 ms collecting sample C, indicating the system performed 9.76 ms of work during this interval.

2. Since both versions of the Windows NT kernel were preemptible, we were able to run the measurement loop inside the kernel. We executed the measurement loop from user-level under Windows 95 since its kernel is not preemptible. This requires our measurement code to make a system call to access the performance counters every time the measurement code generates a sample. All of our results have been corrected for this extra overhead.

shown. The figure shows that the system spent approximately one ms generating samples A, B, D, and E, indicating that the system was idle during the periods in which these samples were generated, but spent 10.76 ms generating sample C. The difference, (10.76 – 1) or 9.76 ms, represents the time required to handle the event.

Next, we used the traditional approach, recording one timestamp when the program received the character (i.e., after a call to getchar()) and a second timestamp after the character was echoed back to the screen. This measurement reported an event-handling latency of only 7.42 ms. The 2.34 ms discrepancy between the two measurements highlights a shortcoming of the conventional measurement methodology. Our test program calls the getchar() function to wait for user input. When the user enters a character, the system generates a hardware interrupt, which is first handled by the dynamically linked library KERNEL32.DLL. In the traditional approach, the measurement does not start until control is returned to the test program. Therefore, it fails to capture the system time required to process the interrupt and reschedule the benchmark thread. In comparison, our idle loop methodology provides a more complete measurement of the computation required to process the keyboard event.

Our idle loop methodology uses CPU busy time to represent event latency, but there are several issues that prevent this from being an accurate measure of the user's perceived response time. One problem is that most graphics output devices refresh every 12–17 ms. In this research, we do not consider this effect.

Another problem is that CPU busy time and CPU idle time do not equate directly with wait time and think time. First, synchronous I/O requests contribute to wait time, even though the CPU can be idle during these operations. Second, in the case of background processing, the user may not be waiting even though the CPU is busy. The first problem could be solved with system support for monitoring the I/O queue and distinguishing between synchronous and asynchronous requests. In order to address the second problem, we must consider how events are processed by the systems. When the user generates key strokes and mouse clicks, they are queued in a message queue to await processing. Therefore, when there are events queued, we can assume that the user is waiting. By combining CPU status (busy or idle), message queue status (empty or non-empty), and status for outstanding synchronous I/O (busy or idle), we can speculate during which time intervals the user is waiting.

Figure 2 shows a state transition diagram for identifying think time and wait time in our system, using the parameters: CPU state, message queue state, and synchronous I/O status. The diagram omits asynchronous I/O, which we assume is background activity, and assumes that users always wait for the completion of an event. In reality, we can never precisely distinguish think time from wait time, because we cannot know what the user is doing and whether the user is actually waiting for an event to complete or is thinking while an event is being processed. For simplicity, in the rest of this paper, we assume that the user waits for each event and report results in terms of event handling latency. In the next

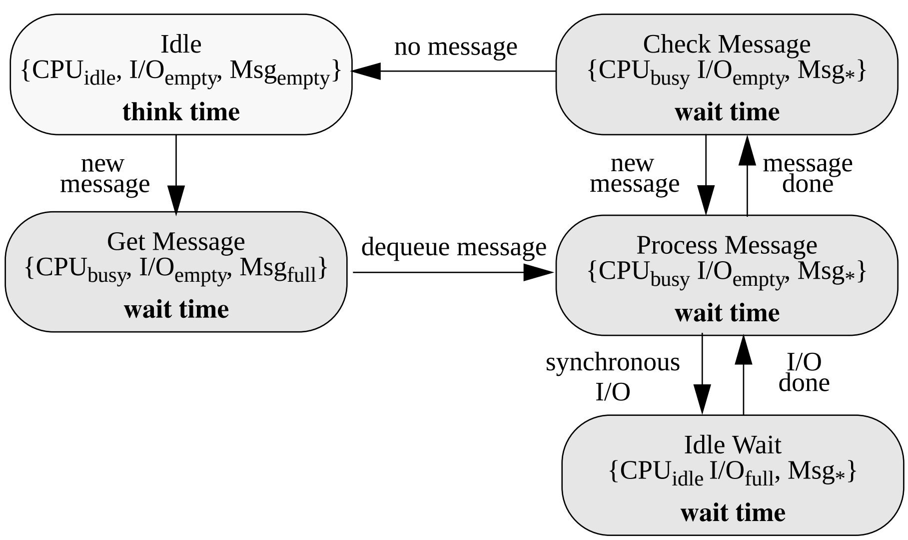


Figure 2. System State Transition Diagram. Monitoring CPU busy time, the I/O queues, and the message queue provides a framework for identifying wait time and think time. We assume that asynchronous I/O is always background activity (and is not shown) and that synchronous I/O causes long-latency events for which users wait.

section, we describe how we obtain information about the status of the message queue.

2.4 Monitoring the Message API

Win32 applications use the PeekMessage() and GetMessage() calls to examine and retrieve events from the message queue. We can monitor use of these API entries by intercepting the USER32.DLL calls. By monitoring use of these API entries, we can detect when an application is prepared to accept a new event and when it actually receives an event. We correlate the trace of GetMessage() and PeekMessage() calls with our CPU profile to determine when the application begins handling a new request and when it completes a request. This allows us to distinguish between synchronous and asynchronous I/O. It is also useful for recognizing situations where asynchronous computation is used to improve interactive response time.

Figure 2 illustrates our design for a finite state machine that distinguishes think time from wait time in a latency-measurement system. In Sections 4, 5, and 6, we will demonstrate how to apply complete information about CPU state and partial information about message queue state to implement part of the FSM. Implementation of the full FSM requires additional system support for monitoring I/O and message queue state transitions. Implementation of such monitoring is part of our continuing work at Harvard.

Next, we will present two simple example measurements to give some insight into some of the non-trivial aspects of interpreting the output of our measurements.

2.5 Idle System Profiles

In this section, we present measurement results for the background activity that occurs during periods of inactivity on Windows NT and Windows 95. This provides intuition about the measurement techniques as well as baseline information, useful for interpreting latency measurements in realistic situations. Figure 3 shows the idle system profiles for the three test systems. To relate non-idle time to elapsed time, we plot elapsed time on the X-axis and the CPU utilization on the Y-axis. Given that each sample represents 1 ms of idle time, the average CPU utilization during a sample interval can be calculated easily. For example, if the system spends 10 ms collecting a sample, and the sample includes 1 ms of idle time, the CPU utilization for that time interval is (10 – 1)/10 = 90%.

Both versions of Windows NT show bursts of CPU activity at 10 ms intervals due to hardware clock interrupts. Correlating the samples with a count of hardware interrupts from the Pentium performance counters shows that each burst of computation is accompanied by a hardware interrupt.

Although we have compensated for the overhead introduced by the user-level idle loop, Windows 95 shows a higher level of activity in comparison to both versions of the NT system. We do not know what causes this increased activity in Windows 95.

By coupling our idle-loop methodology with the Pentium counters, we were able to compute the interrupt handling overhead for various classes of interrupts — measurements difficult to obtain using conventional methods. For example, the smallest clock interrupt handling overhead under Windows NT 4.0 was about 400 cycles, or 4 µs.

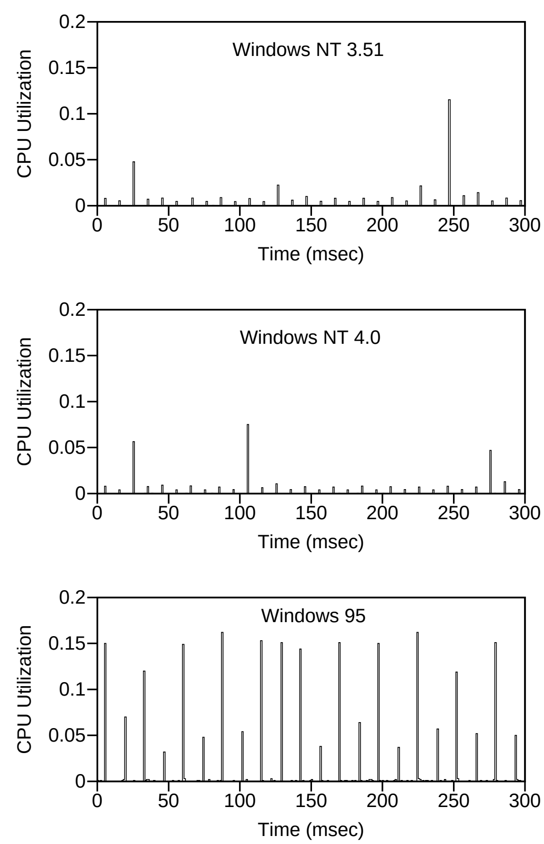
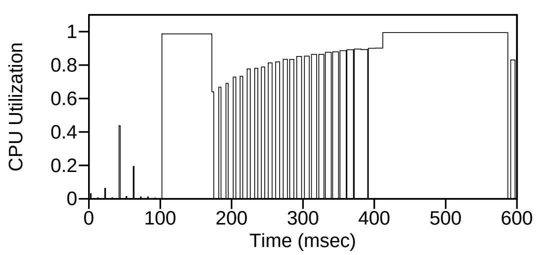


Figure 3. Idle Loop Measurements for the three operating systems. These graphs provide a graphical representation of our idle-loop measurement and illustrate differences in the idle behavior of the three systems. For each recorded sample, we subtract the one ms of idle time that our busy-wait loop requires to obtain the amount of non-idle activity. During an idle period, this difference is zero, and we report a CPU utilization of zero. When there is other CPU activity, more than one ms of time will pass before the completion of one ms of idle time, and the CPU utilization during that interval is non-zero.

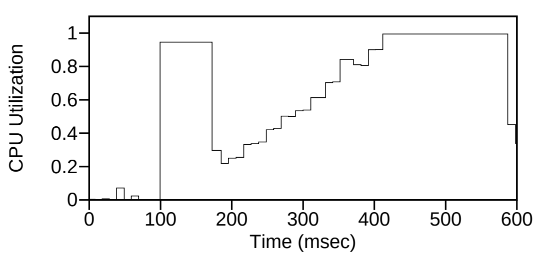
2.6 Window Maximize

It is becoming common for interactive systems to perform extra work, such as animation, for the user's "viewing pleasure." The following experiment shows the effects of such features. We measured the system activity generated when maximizing a previously minimized window; the system displays a window that increases in size gradually as it is maximized. In order for the animation to be visible to the user, the system must pace itself, inserting delays. These delays cause the system to become idle, resulting in multiple idle-loop measurement samples that correspond to the single user event (i.e., maximize window).

Figures 4a and 4b show the results of our measurement in two different resolutions. Figure 4a shows the full 1 ms resolution of the data, while Figure 4b shows the CPU utilization averaged over 10 ms intervals. Both figures clearly show the 80 ms of 100% CPU utilization required to process the input event (from 100 to 180 ms) and another period of 100% CPU utilization starting around 400 ms to redisplay the page. The stair pattern between 180 and 400 ms illustrates the CPU activity required to perform the animation. From Figure 4a, we can observe that the bursts of CPU activity for performing animation are aligned on 10 ms boundaries, suggesting that they are scheduled by clock interrupts. Each step of animation takes progressively longer time to complete as the window outline increases in size.



(a)



(b)

Figure 4. CPU Usage Profile for a Window Maximize Operation under Windows NT 4.0. Figure 4a shows the measurement of computation generated when a window is transformed from an icon in the NT 4.0 icon bar to an open window using 1 ms resolution. The short spikes between 180 ms and 400 ms correspond to the animation that is used as the window outline moves from the menu bar to its destination. The upward slope occurs as the outline increases in size. The period of continuous computation from 400 ms to 600 ms corresponds to the redraw computation for the window.

Figure 4b shows the same data as Figure 4a, but with the CPU utilization averaged over 10 ms intervals. The details in Figure 4a are useful, but become illegible for longer running tests.

This measurement shows that a single user event can correspond to multiple intervals of CPU busy time. Such events complicate the task of precisely identifying event boundaries. Monitoring the Message API (section 2.4) is one of the techniques that helps us pinpoint the beginning and ending of interactive events.

3 Benchmarks and Metrics

Our benchmark set is organized into three categories. *Microbenchmarks* are useful for understanding system behavior for simple interactive operations, such as interrupt handling and user-interface animation. By analyzing microbenchmarks, we develop an understanding of the low-level behavior of the system. We then extend our measurement to *task-oriented benchmarks* in order to understand the real impact of latency on the perceived interactive responsiveness of an application. These task-oriented benchmarks are based on applications from typical PC office suites and are designed to represent a realistic interactive computing workload. We further apply *application microbenchmarks* to evaluate isolated interactive events from the realistic workloads. Our application microbenchmarks include such computations as page-down of a Power-Point document and editing of an embedded OLE object.

We used Microsoft Visual Test to create most of our microbenchmarks and task-oriented benchmarks. MS Test provides a system for simulating user input events on a Windows system in a repeatable manner. Test scripts can specify the pauses between input events, generating minimal runtime overhead. However, in some cases, the way that Test drives applications alters the behavior of those applications. This effect is discussed in detail in Section 5.4.

3.1 Evaluating Response Time

Early in this project, we had planned to develop a new latency metric, a formula that could be used to summarize our measurements and provide a single scalar figure of merit to characterize the interactive performance of a given workload. Events that complete in 0.1 seconds or less are believed to have imperceptible latency and do not contribute to user dissatisfaction[3], whereas events in the 2-4 second range invariably irritate users [12]. Events that fall between these ranges may be acceptable but can correspond to perceptibly worse performance than events under 0.1 seconds. Our intuition is that a user-responsiveness metric would be a summation of the form:

$$\sum_i f_i(x) \quad \text{where}$$

$$f_i(x) = \begin{cases} 0 & t(i) \le T \\ (t(i) - T)^p & t(i) > T \end{cases}$$

T = user perception thresshold
p = some exponent
$t(i)$ = latency of event i

However, we also believe that the threshold, T, is a function of the type of event. For example, users probably expect keystroke event latency to be imperceptible while they may expect that a print command will impose some delay. The issues of event types, user expectation, the precise tolerance of users for delay, and the limitations of human perception are beyond our field of expertise. Presented with these obstacles, we modified our plans, and present latency measurements graphically. We trust that the issues in human-computer interaction can be resolved by specialists. In the meantime, our visualization of latency enables us to compare applications and develop an intuition for responsiveness without risking the inappropriate data reductions that could occur given our limited background in experimental psychology.

3.2 Visualizing Event Latency

Figure 5 is an example of the graphical representation of our raw data. Each vertical bar represents an event that began at the time represented by the X value and lasted for a period represented on the Y axis. Figure 5a shows the data for an entire Microsoft Word benchmark run, while 5b shows a magnification of a two second interval. The complete event latency profile provides a very coarse view of the application, while the magnification provides the detail to explain the periodicity in the overall pattern. By drawing a horizontal line at a given "irritation threshold," the frequency and distribution of irritating events is readily visible.

For our task-oriented benchmarks, we use three graphical representations to capture the responsiveness of an application. First, we present histograms, showing the number of events corresponding to each measured latency. This presents a detailed breakdown of the event latencies and provides some intuition into the different categories of events present in an application. Next, we integrate over the histogram presenting a cumulative latency graph. This provides the quantitative data indicating how events of a particular duration contribute to the overall time required to complete a task. Finally, we plot the cumulative latency as a function of the number of events, providing an intuition about the variance in response time perceived by the user. Note that in each of these cases, the events are sorted by their duration, not by their actual time of occurrence.

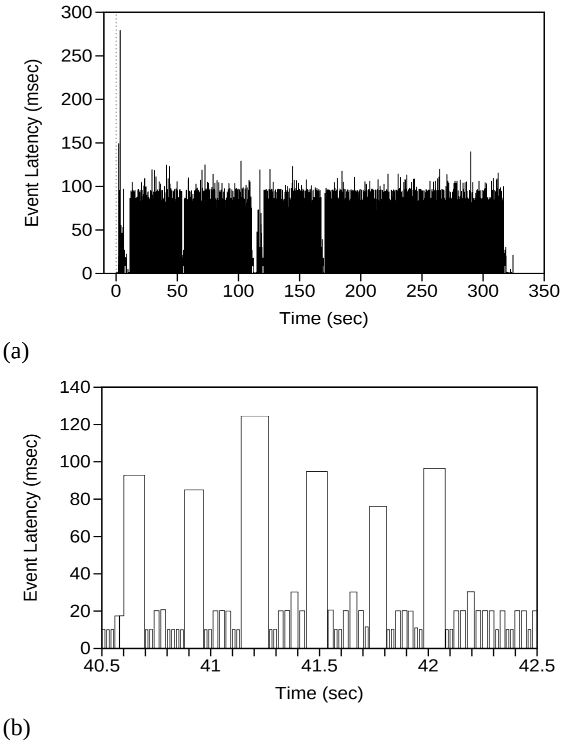


Figure 5. Raw Data Representation. Figure 5a represents a 1000 event trace of Microsoft Word on Windows NT 3.51. The benchmark is sufficiently long that a complete profile does not lend itself to detailed analysis. The magnified profile (b) shows the periodicity of long and short latency events. Notice that the majority of the events fall below the 0.1 second threshold of user perception but that a significant number fall well above the threshold.

4 Microbenchmarks

In this section, we present some basic measurements of simple interactive events. This helps us explore the character of our tools and understand the kinds of things we can and cannot measure.

Figure 6 shows the latencies for two simple interactive events, unbound key stroke and mouse click on the screen background, under the three operating systems. We were unable to measure the overhead of Microsoft Test for these microbenchmarks, so we were forced to use manual input. To compensate for the potential variability introduced by a human user, we report the mean

3. For multimedia applications and applications that use visual feedback (such as rubber-banding in a drawing program), latency requirements may be smaller. The limitations of the human visual system suggest that 10 ms provides a lower bound on the latencies that can be perceived for continuous media operations [6].

of 30–40 trials, ignoring cold cache cases. The most significant standard deviations occurred in the key click events for Windows NT 4.0 and Windows 95 (8%) while all the remaining standard deviations were under 2% of the mean.

On the key stroke test, Windows 95 shows substantially worse performance than NT 4.0. This is a reflection of segment register loads (not shown) and other overhead associated with 16-bit windows code [4], which persist in Windows 95.

The mouse click results are even more striking. The Windows 95 measurements are off the scale, because the system busy-waits between “mouse down” and “mouse up” events; therefore our measurement indicates the length of time the user took to perform the mouse click. This is much longer than the actual processing times of the NT systems and is not indicative of the actual Windows 95 performance.

Our methodology provides little guidance in explaining the differences in performance between Windows NT 3.51 and NT 4.0, but it does highlight the fact that instructions and data references occur roughly in proportion to cycles across the systems for both of the simple interactive events. Therefore, we conclude that in the warm cache case, the performance differences are a function of the code path lengths. It is possible that the difference in code path length stems from the change in GUI between NT 3.51 and NT 4.0.

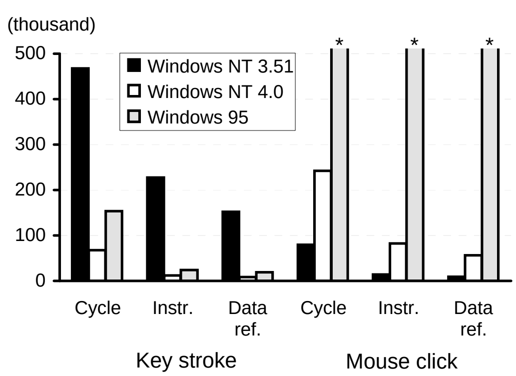


Figure 6. Latency for simple interactive events. This figure shows the latencies for a simple mouse click and a simple key stroke in each of the three systems. In Windows 95, the system remains busy the entire time between a “mouse down” and a “mouse up” event; therefore the latency of the event is a function of the duration of the user’s press on the mouse. This is much larger than the actual processing times of the NT systems and is not indicative of Windows 95 performance. Each microbenchmark was repeated 30–40 times; the standard deviations were less than 8% for key stroke on Windows NT 4.0 and Windows 95 and less than 2% on the remaining tests.

5 Task-oriented Benchmarks

In this section, we measure three task-oriented benchmarks, designed to model realistic tasks that users commonly perform using the target applications. In using these longer running benchmarks we have two specific goals. The first is to measure the system performance for a realistic system state. An often-cited problem of microbenchmarks is that they tend to measure the system when various caches are already warm. However, measuring the system when all the caches are cold is also unrealistic. Neither extreme is representative of the system state in which the target micro-operations are invoked in common practice. By measuring the latency of micro-operations embedded in a longer realistic interactive task, we measure each micro-operation under more realistic circumstances. The second goal is to identify long-latency operations that users encounter as they perform tasks on the systems. Since these long-latency operations have a greater effect on how users perceive system performance than very short events, we will isolate several such operations in our latency record and examine their activity in detail.

We ran each benchmark five times using Microsoft Test and found that the results were consistent across runs. The standard deviations for the elapsed times and cumulative CPU busy times were 1–2%, and the event latency distributions were virtually identical. The graphical output shown in the following sections depicts one of the five runs for each benchmark.

5.1 Microsoft Notepad

Notepad is a simple editor for ASCII text distributed with all versions of Microsoft Windows. Our Notepad benchmark models an editing session on a 56KB text file, which includes text entry of 1300 characters at approximately 100 words per minute, as well as cursor and page movement. With this benchmark, we demonstrate how differences in average response time across the three systems manifest themselves in our visual representation of latency and how they can be used to compare system performance. We used the same Notepad executable (the Windows 95 version) on all three systems and used a Microsoft Test script to drive Notepad. Since virtually all Notepad activity is synchronous, we were able to collect the latency figures for every keystroke that the user made in a straightforward way. By correlating our idle loop measurement with our monitoring of the PeekMessage() and GetMessage() API calls, we were able to clearly identify the Test overhead and remove it from the data presented in Figure 7.

Notice that the Y scale in the histogram in Figure 7 is a logarithmic scale. The cumulative latency graph shows that for all three systems, over 80% of the latency of Notepad is due to low-latency (less than 10 ms) events. These short-latency events are the keystrokes that generate printable ASCII characters. The remaining 20% of the total latency are due to the longer latency (at least 28 ms) keystrokes that cause “page down” or newline operations. These keystrokes cause Notepad to refresh all or part of the screen. The smoothness of the curves in the bottommost graph in Figure 7 shows that there is little variance in either the long latency events or the short-latency events. Events of the same type contribute equally to the total latency.

The latencies measured are relatively small for Notepad and reflect both the simplicity of the application and the relatively fast PC that we used for our experiments. Although these differences in latency are likely to go unnoticed by users of our test system, they might have a significant effect on user-perceived performance on a slower machine.

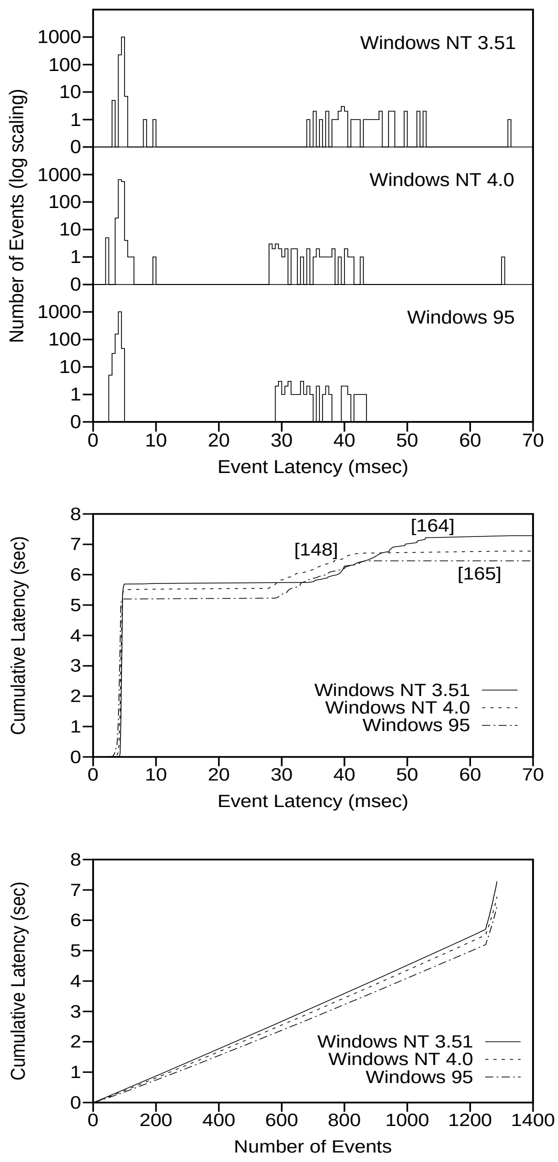


Figure 7. Notepad Event Latency Summary. The bracketed numbers in the second graph report the total elapsed time for the benchmark run in seconds. Notice that Windows 95 has the smallest cumulative latency, but its elapsed time is larger than that of both NT 3.51 and NT 4.0. This seemingly anomalous result is an artifact of Microsoft Test, which generates an extra WM_QUEUESYNC message after each input event. The time to process this message is longer under Windows 95 than under the NT systems. We do not include this time in the event latencies, but it does contribute to elapsed time.

5.2 Microsoft Powerpoint

Powerpoint, from the Microsoft Office suite, is a popular application for creating presentation graphics. In our Powerpoint task scenario, the user starts Powerpoint immediately after powering up the machine and booting the operating system, so that all caches are cold. The user then loads a 46-page, 530KB presentation, and finds and modifies three OLE embedded Excel graph objects[4]. Each of the OLE objects was of similar size and complexity. As with Notepad, we used a Microsoft Test script to drive the application and deliver keystrokes at a realistic rate, with each keystroke separated by at least 150 ms. An important property of the Powerpoint benchmark is that it has a number of events with easily perceptible latencies. Since we were mainly interested in longer events, we pre-processed our data to exclude events with latency of less than 50 ms. Figure 8 shows the results for the two versions of Windows NT. We were unable to run this experiment for Windows 95 due to limitations of Microsoft Test when manipulating OLE embedded object on that system.

The shortest events in Figure 8 (with latency of less than one second) are due to “page down” operations and Excel operations. Both systems exhibited a similar latency distribution for these events. Six events had latencies greater than one second on both systems, in nearly the same relative order. Table 1 lists these long latency events.

All of the long-latency events required disk accesses, which are responsible for the majority of the latency for these events. The effects of the file system cache are most clearly observed in the latency for starting the second OLE edit, as more of the pages for the embedded Excel object editor become resident in the buffer cache.

4. Each OLE edit requires starting the server application (in this case Excel), editing the object, and exiting the server application.

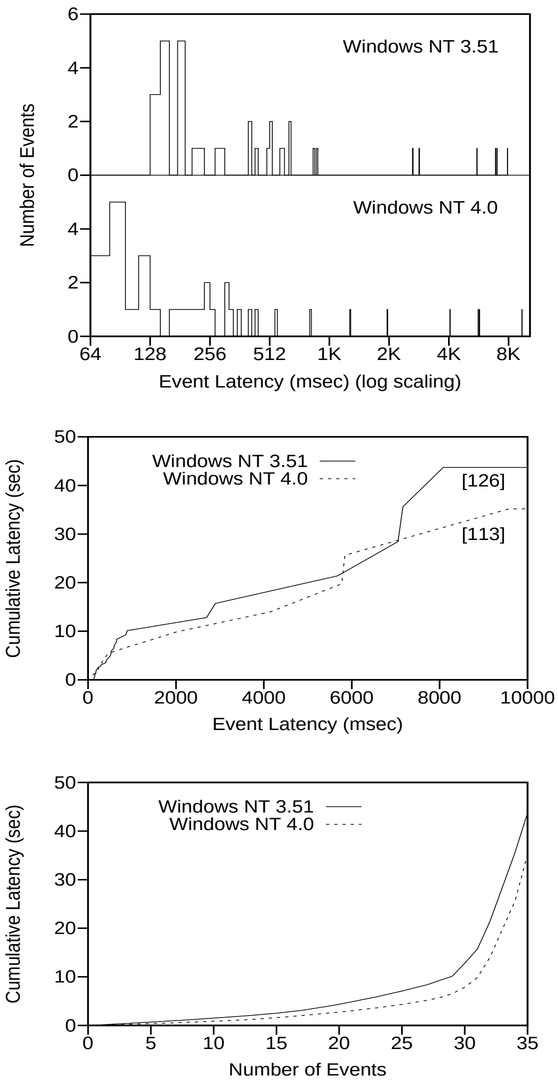


Figure 8. Powerpoint Event Latency Summary. The bracketed numbers in the second graph report the total elapsed time for the benchmark run in seconds. While most of the events for Powerpoint are relatively short (under 500 ms), the majority of the time is spent in long-latency events.

The cumulative latency graph shows that both versions of Windows NT demonstrate similar performance for the short-latency keystrokes, and the majority of the performance difference is a result of the ability of NT 4.0 to handle the long-latency events much more efficiently. We turn to application microbenchmarks to examine this phenomenon in more detail.

5.3 Powerpoint Microbenchmarks

Our Powerpoint task reveals significant differences in latency between the two version of NT but Powerpoint's complexity makes it difficult to reveal the source of these differences. We designed two application

	latency (in seconds)	
	NT 3.51	**NT 4.0**
Save document	8.082	9.580
Start Powerpoint	7.166	5.773
Start OLE edit session (first time)	7.050	5.844
Open document	5.680	4.151
Start OLE edit session (second object)	2.897	2.009
Start OLE edit session (third object)	2.697	1.305

Table 1. Powerpoint events with latency over one second.

microbenchmarks to explore page down and OLE edit operations in detail. Both operations have a perceptible response time and occur multiple times during our Powerpoint task. We measured each operation using Test and collected data from the Pentium performance counters to look for hints to identify the performance differences between the systems. Because of the simplicity of the application microbenchmarks, we were able to run them on the Windows 95 systems as well as the two NT systems.

Page Down

For this benchmark, we measured the latency and the hardware events that occur during a "page down" to a page containing an OLE embedded graph. We repeated the test 10 times for each performance counter, so our measurements reflect warm cache behavior. The standard deviations are all below 3%. Figure 9 shows some of the hardware events observed in each system during the page-down operation. The graph shows that NT 4.0 was able to handle the request in the shortest amount of time followed by Windows 95 and NT 3.51. The difference in the latency between the two versions of Windows NT is explained by the differences in system architecture. In NT 3.51, the Win32 API is implemented by a user-level server. The negative performance effects of this server-based architecture were demonstrated in prior research [4]. In NT 4.0, some components of the Win32 API server are rumored to have been moved into the kernel. The improved locality from this change is reflected in reduced TLB misses for NT 4.0 compared to NT 3.51. A lower TLB miss rate implies fewer protection domain crossings in Pentium processors, which flush the TLB on each crossing [5]. Using 20 cycles per miss as a lower bound on TLB miss handling latency[5], the extra TLB misses that occur for NT 3.51 (both

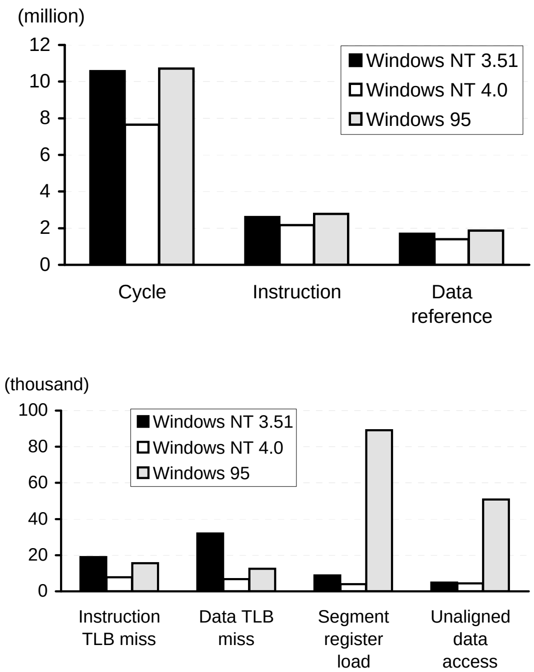


Figure 9. Counter Measurements for Powerpoint page down operation. The larger number of TLB misses in NT 3.51 relative to NT 4.0 indicates a greater number of protection domain crossings. The large number of segment register loads for Windows 95 results from code that runs in 16 -bit mode and accounts for the majority of the difference between Windows 95 and the NT systems.

instruction and data) account for at least 25% of the latency difference between NT 3.51 and NT 4.0.

Comparing NT 4.0 and Windows 95, there are a relatively large number of unaligned data accesses and segment register loads for Windows 95 (Figure 9). The high counts for these events are due to the large components of Windows 95 (such as the graphics API) that are implemented in 16-bit code. Windows 95 also incurs 93% more TLB misses than NT 4.0, although we do not have sufficient information to attribute this behavior to a specific architectural feature in the two systems.

OLE Edit

The long latency events in our Powerpoint task suggest that much of the OLE edit start-up latency is due to disk I/O. In this section, we exclude the effects of disk I/O by measuring an OLE edit start-up with a hot buffer cache. We also repeated this test 10 times for each counter, but we noticed that all of the events and the cycle counter increased steadily on subsequent runs. We speculate that this behavior is unintended, and therefore, report the results from the first run. Figure 10 shows some of the hardware event counts during the OLE edit start-up. Like the page down benchmark, NT 4.0 completes the operation with the shortest latency followed by Windows 95 and NT 3.51. Comparing NT 4.0 to NT 3.51, the observations made for the page down benchmark carry through to the OLE edit. Overhead from elevated TLB miss rates account for at least 23% of the latency difference between NT 3.51 and NT 4.0. In Windows 95, we observed a large number of segment register loads and unaligned data accesses, both of which can be attributed to code executing in 16-bit mode.

5.4 Microsoft Word

Our task-oriented workload for Microsoft Word consists of text entry of a paragraph of approximately 1000 characters. It includes cursor movement with arrow keys and backspace characters to correct typing errors. The timing between keystrokes was varied to simulate realistic pauses when composing a document, and line justification and interactive spell checking were enabled. We do

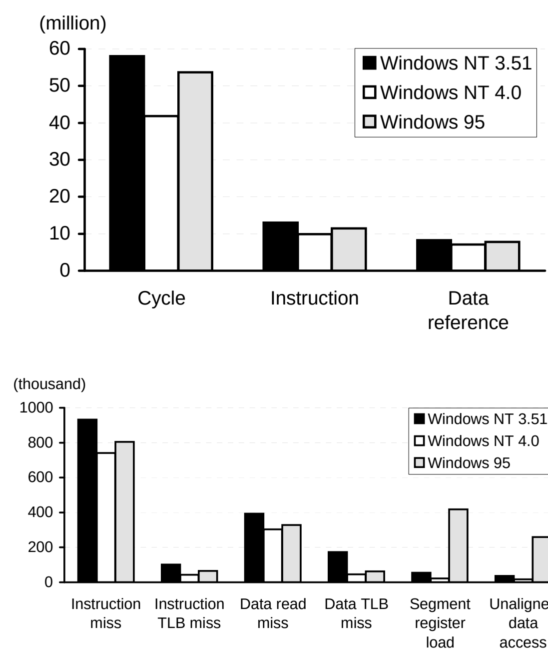


Figure 10. Counter Measurements for Powerpoint OLE edit start-up operation. As in Figure 9, TLB misses account for a significant fraction of the difference between NT 3.51 and NT 4.0, and segment register loads account for the majority of the difference between Windows 95 and NT.

5. Our idle loop measurements give us a good lower bound on the TLB miss penalty. By monitoring both the TLB miss count and elapsed cycles, we can compare measurement intervals where one TLB miss occurred to intervals where no TLB misses occurred. Twenty cycles is the lowest TLB miss handling time observed during our measurements, and would require that both of the memory references required to handle a TLB miss on the Pentium hit in the first level cache.

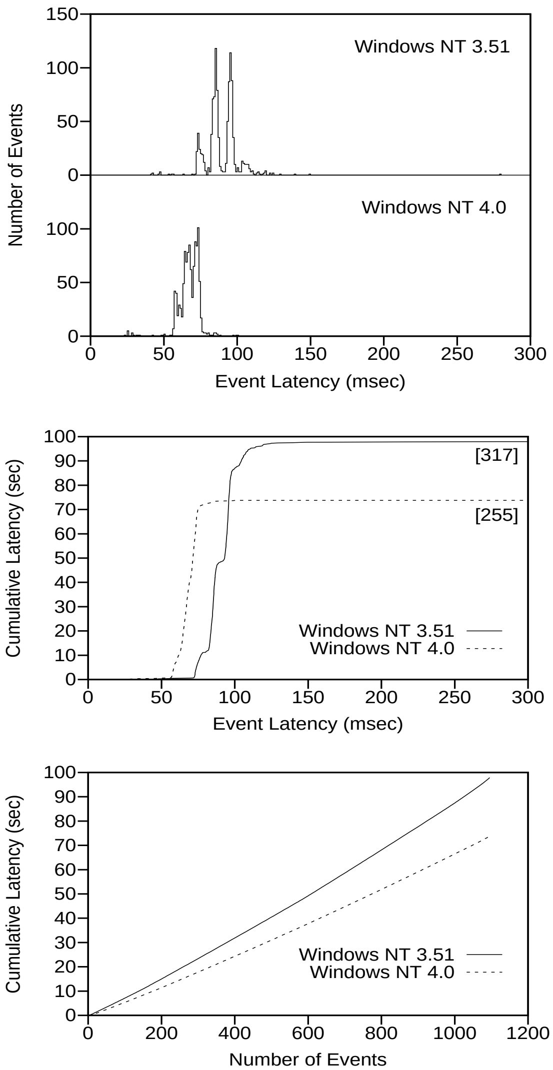


Figure 11. Microsoft Word Event Latency Summary. The bracketed numbers in the second graph report the total elapsed time for the benchmark run in seconds. Although NT 4.0 demonstrates uniformly better response time, both systems have most latencies below the threshold of user perception.

not report results for Windows 95, because the system does not become idle immediately after Word finishes handling an event, making all event latencies appear to be several seconds long.

Figure 11 shows results for Microsoft Test driven simulations on the two versions of Windows NT. Compared to Notepad, Word requires substantially more processing time per keystroke, due to additional functionality such as text formatting, variable-width fonts and interactive spell checking. For the majority of interactive events, NT 4.0 exhibits shorter response time and lower variance than NT 3.51.

The Microsoft Word benchmark demonstrates both the strengths and limitations of evaluating interactive performance using latency. Compared to throughput measurements, our latency analysis provides much more detailed information, such as variations in latency and the distribution of events with different latencies. However, the structural features of Word push us to the limit of the behavior we are able to analyze. Our analysis indicates that Word uses a single system thread, but responds to input events and handles background computations asynchronously using an internal system of coroutines or user level threads.

Distinguishing background activity from foreground activity in Word is challenging. We examined the results of hand-generated Word input under Windows NT 3.51, compared it to the Test-generated results, and found significant differences. For our hand-generated tests, we ran seven trials, with the same typist and input, and found that the event histograms appeared very similar and that the variation in cumulative latency and elapsed time was less than 4% across the runs. While the Test results showed that most events had latency between 80 and 100 ms, we measured a 32 ms typical latency for the hand-generated input. This difference in event latency was accompanied by a compensating difference in background activity. The hand-generated input showed a higher level of background activity than the Test-generated results. We also observed that carriage returns under the hand-generated input took longer than 200 ms to handle while the longest latency events we saw in the Test-generated runs were 140 ms. Our Message API log reveals that Test generates a WM_QUEUESYNC messages after every keystroke. We hypothesize that these messages were responsible for the different behavior under Test and under manual typing. However, with our current tools, the complexity of Word makes it difficult to thoroughly analyze even the simple experiment we present here.

6 Discussion

The tools and techniques we have discussed here are a first step towards understanding and quantifying interactive latency, but there remains much work to be done. In the absence of system and application source, better performance monitoring tools would be useful. Our measurements could be improved through API calls that return information about system state such as message queue lengths, I/O queue length, and the types of requests on the I/O queue. Currently, some of this information can be obtained, but it is painful (e.g., monitoring the GetMessage() and PeekMessage() calls).

Even in the presence of rich APIs, the task of distinguishing between wait time and think time is not always possible. There is no automatic way to detect exactly what a user is doing. Without user input, we can never tell whether a user is genuinely waiting while the system paints a complicated graphic on the screen or is busy thinking. For simulations using designed scripts, we can make assumptions about when users think and then analyze performance based on those assumptions, but the most useful analysis will come from evaluating actual user interaction.

One factor that contributes to user dissatisfaction is the frequency of long-latency events. We processed the Microsoft Word profile of Figure 5 to analyze the distribution of interarrival times of events above a given threshold. Since most events in the Word benchmark were very short, we chose thresholds around 100 ms. Table 2 shows the summaries for these thresholds. Note that the standard deviations are of the same order of magnitude as the averages themselves, indicating that there is no strong periodicity between long-latency events.

Threshold (in msec)	Number of events above threshold	Interarrival times	
		Average (in sec)	Std. Dev. (in sec)
100	101	3.1	3.1
110	26	12.4	10.6
120	8	41.1	48.8

Table 2. Summary of interarrival distributions for Microsoft Word benchmark on Windows NT 3.51. Notice that an increase of 10% in the threshold (from 100 ms to 110 ms) reduces the number of above threshold events by a factor of 4.

We then examined the truly long-latency events from the Powerpoint benchmark. Figure 12 shows the event latency profile for all events over 50 ms. Both systems show similar periodicity with the better performing 4.0 system demonstrating smaller interarrival times to match its shorter overall latency.

In the case of Word, the interarrival times are clustered because most events have similar latency. In the case of Powerpoint, the interarrival times of long-latency events are simply the interarrival times of a few particular classes of events. The distribution of these events is entirely dependent upon when we issued such requests in our test script and is not necessarily indicative of the distribution that might be obtained from a real user. In this test, none of the simple keystroke events were responsible for generating long-latency events, rather all the events with latencies over 50 ms result from major operations for which user expectation for response time

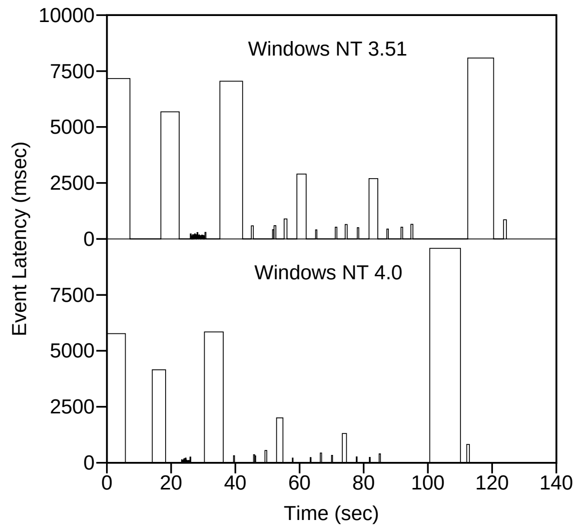


Figure 12. Time Series of long-latency events for Powerpoint. Both systems show similar distributions, with NT 4.0 having slightly shorter interarrival intervals.

is generally longer. Until our tools become sophisticated enough to examine long traces of complex events generated by a real user, further analysis of these interarrival times is not particularly productive.

Over time, our tools will become better able to deal with the sophisticated applications that we seek to analyze, but we need the human factors community to assist us in understanding the limits of human perception and the models of user tolerance. Some of the questions that must be answered are:

- What are the limits of human perception?
- How do the limits vary by task (e.g., typing versus mouse-tracking)?
- How do the user expectation and tolerance for interactive response time vary by task?
- How does user dissatisfaction grow with increasing of latency?
- How does user dissatisfaction grow with the variance of latency?
- What aspects of performance contribute the most to user satisfaction?

7 Conclusions

Latency, not throughput, is the key performance metric for interactive software systems. In this paper, we have introduced some tools and techniques for quantifying latency for a general class of realistic interactive application. To demonstrate our methodology, we applied it to compare the responsiveness of realistic applications running on three popular PC operating systems.

Whereas current measurements of latency are generally limited to microbenchmarks, our approach allows us to measure latency for isolated events in the context of realistic interactive tasks. Our latency measurements give a more accurate and complete picture of interactive performance than throughput measurements.

We have combined a few simple ideas to get precise information about latency in interactive programs. We have shown that using these ideas we can get accurate and meaningful information for simple applications and also, to a degree, for complex applications. The requirements of these techniques are not out of reach; in particular, a hardware cycle counter, a means for changing the system idle loop, and a mechanism for logging calls to system API routines are needed. Additional support for detecting the enqueuing of messages and the state of the I/O queue would provide a more complete framework for latency measurement. We have shown the limitations of our system for applications such as Microsoft Word that use batching and asynchronous computation.

Measuring latency for an arbitrary task and an arbitrary application remains a difficult problem. Our experience with Microsoft Word demonstrates that there are many difficult technical issues to be resolved before latency will become a practical metric for system design. Our graphical representation provides a great deal of information about program behavior to specialists, but is probably not appropriate for more widespread use. The two key components necessary to provide consumers a single figure of merit are further work in human factors and some method for distinguishing user think time from user wait time.

8 References

[1] Business Applications Performance Corporation, "SYSmark for Windows NT," Press Release by IDEAS International, Santa Clara, CA, March 1995.

[2] Brian N. Bershad, Richard P. Draves, and Alessandro Forin, "Using Microbenchmarks to Evaluate System Performance." *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE, Key Biscayne, Florida, April 1992, pages 148–153.

[3] Ben Smith, "Ultrafast Ultrasparcs," *Byte Magazine*, January 1996, page 139. Additional information on the Bytemarks suite is available on the Internet: http://www.byte.com/bmark/bdoc.htm.

[4] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Michael D. Smith, "The Measured Performance of Personal Computer Operating Systems," *ACM Transactions on Computer Systems 14*, 1, February 1996, pages 3–40.

[5] Intel Corporation, *Pentium Processor Family Developer's Manual. Volume 3: Architecture and Programming Manual,* Intel Corporation, 1995.

[6] C. J. Lindblad and D. L. Tennenhouse, "The VuSystem: A Programming System for Compute-Intensive Multimedia," To appear in *IEEE Journal of Selected Areas in Communication*," 1996.

[7] Larry McVoy, "Lmbench: Portable tools for performance analysis," *Proceedings of the 1996 USENIX Technical Conference*, January 1996, pages 179–294.

[8] Jeffrey C. Mogul, "SPECmarks are leading us astray," *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE, Key Biscayne, Florida, April 1992, pages 160–161.

[9] James O'Toole, Scott Nettles, and David Gifford, "Concurrent Compacting Garbage Collection," *The Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993, pages 161–174.

[10] John K. Ousterhout, "Why Operating Systems Aren't Getting Faster As Fast As Hardware." *Proceedings of the Summer 1991 USENIX Conference*, June 1991, pages 247–256.

[11] Mark Shand, "Measuring Unix Kernel Performance with Reprogammable Hardware," Digital Paris Research Lab, Research Report #19, August 1992.

[12] Ben Shneiderman, *Designing the User Interface*, Addison-Wesley, 1992.

[13] Jeff Reilly, "SPEC Discusses the History and Reasoning behind SPEC 95," *SPEC Newsletter*, 7(3):1-3, September 1995.

[14] M. L. VanName and B. Catchings, "Reaching New Heights in Benchmark Testing," *PC Magazine*, 13 December 1994, pages 327-332. Further information on Ziff-David benchmarks is available on the Internet: http://www.zdnet.com/zdbop/.