Global Journal of Computer Science and Technology: G Interdisciplinary
Volume 14 Issue 5 Version 1.0 Year 2014
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Using Latency to Evaluate Computer System Performance

By Olawuyi J. O., Fagbohunmi S. G., Olawuyi O. M., Mgbole F.
Abia State Polytechnic, Nigeria

Abstract- Building high performance computer systems requires an understanding of the behaviour of systems and what makes them fast or slow. In addition to our file system performance analysis, we have a number of projects in measuring, evaluating, and understanding system performances. The conventional methodology for system performance measurement, which relies primarily on throughput-sensitive benchmarks and throughput metrics, has major limitations when analyzing the behaviour 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 use some 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 behaviour of latency-critical events. We demonstrate our techniques in an analysis of the performance of two releases of Windows 9x and Windows XP Professional. 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.

GJCST-G Classification: B.8.2

USINGLATENCYTOEVALUATECOMPUTERSYSTEMPERFORMANCE

Strictly as per the compliance and regulations of:

Using Latency to Evaluate Computer System Performance

Olawuyi J.O. [α], Fagbohunmi S.G. [σ], Olawuyi O.M. [ρ] & Mgbole F. [ω]

Abstract- Building high performance computer systems requires an understanding of the behaviour of systems and what makes them fast or slow. In addition to our file system performance analysis, we have a number of projects in measuring, evaluating, and understanding system performances. The conventional methodology for system performance measurement, which relies primarily on throughput-sensitive benchmarks and throughput metrics, has major limitations when analyzing the behaviour 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 use some 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 behaviour of latency-critical events. We demonstrate our techniques in an analysis of the performance of two releases of Windows 9x and Windows XP Professional. 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.

I. 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.

Author α ω: Department of Computer Science, Abia State Polytechnic, Aba. e-mails: olawuyijo@yahoo.co.uk, mojirayool use ye2014@yahoo.com
Author σ: Department of Computer Engineering, Abia State Polytechnic, Aba. e-mail: fman707@yahoo.com
Author ρ: School of General Study, Alvan Ikoku Federal College of Education, Owerri. e-mail: oluwasijibomi@Hotmail.com

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 9x, Windows 2000, and Windows XP Professional, 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 9x, Windows 2000, and Windows XP Professional. Sections 6 and 7 discuss the limitations of our work and conclude.

a) The Irrelevance of Throughput

Most macro-benchmarks 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.

i. *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 favourably 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 over all activity of an application. Throughput measures pro vide no way to distinguish between these features and events that are less frequent but have a significant impact on user-perceived performance.

ii. *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.

II. 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 (Ben Shniderman, Designing the user interface, 1992). Distinguishing between wait time and 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.

a) Experimental Systems

We ran our experiments on a personal computer based on an Intel Premiere III motherboard, with the Intel Neptune chip set and a 650 MHz Pentium processor. Our machine was equipped with a 256KB asynchronous SRAM Level 2 cache, 512 MB of RAM, and a Diamond Stealth 64 DRAM display card. We used a dedicated 10GB 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 2000 and Windows XP systems used a NTFS file system, while the Windows 9x system used a FAT32 file system.

b) 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 as described in Intel Corporation Developers manual (1995). 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.

c) 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. 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 1ms 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 1ms 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 key stroke 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.

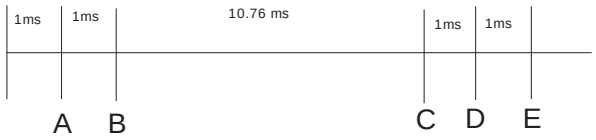


Figure 1 : Validation of Idle loop Methodology. The system spent one ms collecting each of samples A, B, D and E but spent 10.76 ms collecting sample C indicating the system performed 9.76 ms of work during this interval

For the sake of clarity only a few samples are 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 rep resent event latency, but there are several issues that pre vent 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 awaiting 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 real ity, 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 section, we describe how we obtain information about the status of the message queue.

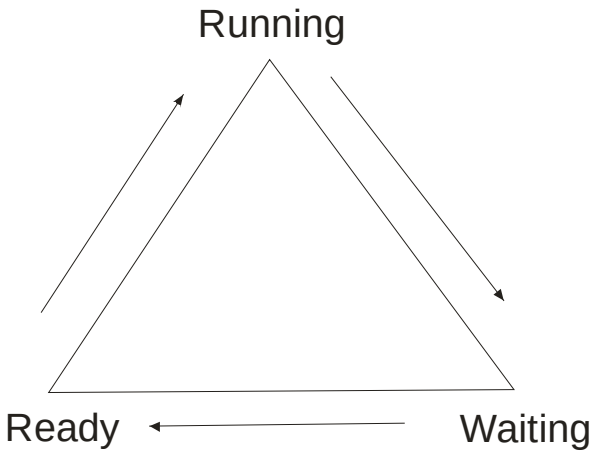


Figure 2 : Showing state transition diagram

d) Monitoring the Message API

Win32 applications use the Peek Message() and Get Message() 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 Get Message() and Peek Message() 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. 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.

e) Idle System Profiles

In this section, we present measurement results for the background activity that occurs during periods of inactivity on Windows 9x and Windows XP. 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 9x shows a higher level of activity in comparison to both versions of the Windows XP system. We do not know what causes this increased activity in Windows 9x.

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 XP was about 800 cycles, or 8 ms.

III. Benchmarks and Metrics

Our benchmark set is organized into three categories. Microbenchmarks are useful for understanding system behaviour for simple interactive operations, such as interrupt handling and user-interface animation. By analyzing microbenchmarks, we develop an understanding of the low-level behaviour 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 PowerPoint 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 behaviour of those applications. This effect is discussed in detail in Section 5.4.

a) 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, whereas events in the 2-4 second range invariably irritate users Ben Shneiderman (1992). 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:

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

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

T – user perception threshold
p – some exponent
$x(i)$ – 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.

IV. Micro-Benchmarks

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 3

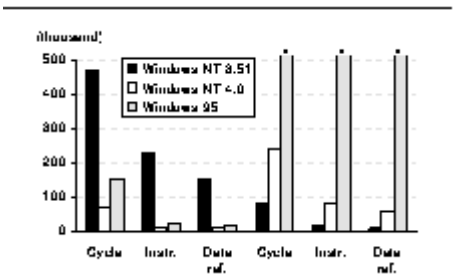


Figure 3 shows the latencies for two simple interactive events, unbound key stroke and mouse click on the screen background, under the three operating systems. We were unable to measure the overhead of

Microsoft Test for these micro-benchmarks, so we were forced to use manual input. To compensate for the potential variability introduced by a human user, we report the mean of 30-40 trials, ignoring cold cache cases. The most significant standard deviations occurred in the key click events for Windows XP and Windows 9x (8%) while all the remaining standard deviations were under 2% of the mean.

On the key stroke test, Windows 9x shows substantially worse performance than Windows XP. This is a reflection of segment register loads (not shown) and other overhead associated with 16-bit and 32–bit windows codes as asserted by Bradley Chen et al, which persist in Windows 9x.

The mouse click results are even more striking. The Windows 9x 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 Windows XP systems and is not indicative of the actual Windows 9x performance.

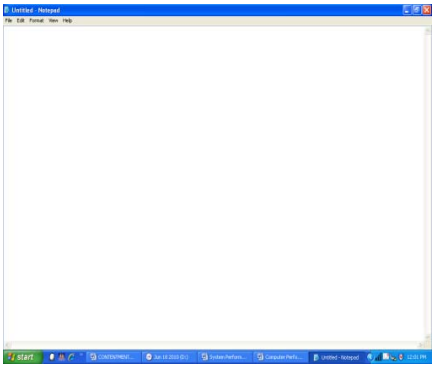
Our methodology provides little guidance in explaining the differences in performance between Windows 2000 and Windows XP Professionals, but it does highlight the fact that instructions and data references occur roughly in pro portion to cycles across the systems for both of the simple interactive events. 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 the two notable releases of Windows.

V. Task-Oriented Benchmarks

In this section, we measure three task-oriented bench marks, 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 micro-benchmarks is that they tend to measure the sys tem 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 inter active 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 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 out put shown in the following sections depicts one of the five runs for each benchmark.

a) 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 com pare system performance. We used the same Notepad executable (the Windows XP 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 key stroke that the user made in a straightforward way. By correlating our idle loop measurement with our monitoring of the Peek Message() and Get Message() API calls, we were able to clearly identify the Test overhead and remove it from the data presented.

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. 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.

b) Microsoft Notepad

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. 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 key strokes 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 below shows the results for the two versions of Windows. We were unable to run this experiment for Windows XP due to limitations of Microsoft Test when manipulating OLE embedded object on that system.

The shortest event (with latency of less than one second), are due to "page down" operations and MS-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.

The cumulative latency graph shows that both versions of Windows 2000 and Windows XP demonstrate similar performance for the short-latency keystrokes, and the majority of the performance difference is a result of the ability of NTFS file system to handle the long-latency events much more efficiently.

	Latency (in seconds)	
	NT 3.51	NT 4.0
Save document	6.062	9.560
Start Powerpoint	7.166	5.773
Start OLE edit session (first time)	7.050	5.644
Open document	5.660	4.151
Start OLE edit session (second object)	2.697	2.009
Start OLE edit session (third object)	2.697	1.305

Table 1.

The standard deviations are all below 3%

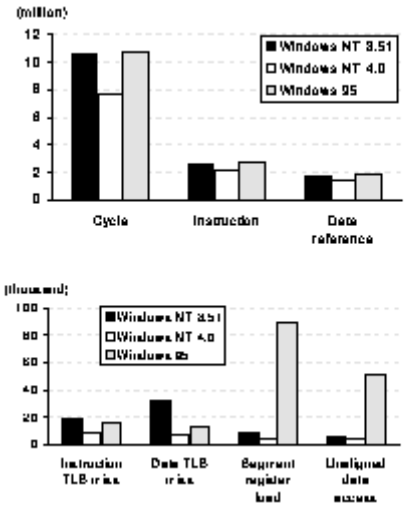


Figure 4 : Latency for simple interactive events

c) 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 not report results for Windows 9x, 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 NTFS based Windows. Compared to Notepad, MS-Word requires substantially more processing time per keystroke, due to additional functionality such as text formatting, variable-width fonts and inter active spell checking. For the majority of interactive events, Windows XP exhibits shorter response time and lower variance than Windows 2000.

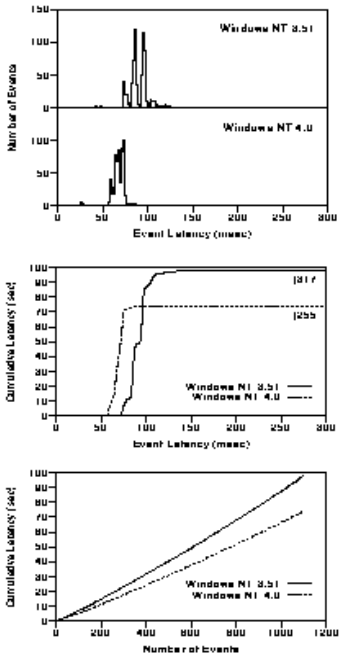


Figure 5 : Notepad Event Latency Summary

The MS-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. How ever, 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 MS-Word is challenging. We examined the results of hand-generated Word input under Windows 2000 OS, 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 back ground 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.

VI. Summary

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 Get Message() and Peek Message() 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 inter-arrival 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 ms	No of Events above Threshold	Inter-arrival times	
		Average (in sec)	Std Deviation (in sec)
100	101	3.1	3.1
110	26	12.4	10.6
120	8	41.1	48.8

Table 2

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 inter-arrival times to match its shorter overall latency.

In the case of Word, the inter-arrival times are clustered because most events have similar latency. In the case of PowerPoint, the inter-arrival times of long-latency events are simply the inter-arrival 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 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 inter-arrival times is not particularly productive.

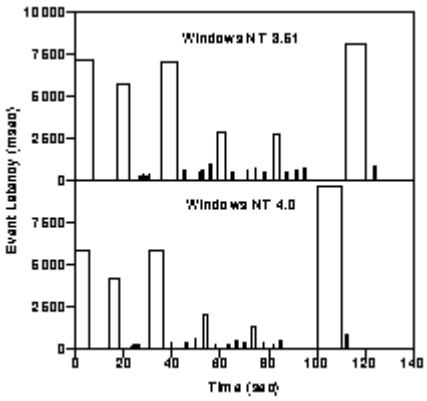


Figure 6 : Counter measurements for PowerPoint page down operation

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?

VII. 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 micro-benchmarks, 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.

VIII. Acknowledgments

The authors have benefited from the works of Brian N. Bershad, et al, John K. Ousterhout, and Jeffrey C. Mogul of Business Applications Performance Corporation. We thank them for their insights and suggestions. The authors are greatly grateful and indebted to our research partners and colleagues in the School of Science, and Computer Engineering, Abia State Polytechnic, Aba, who have being research fellows and have supported our research for some years now. We would also like to thank the Management of both Abia State Polytechnic, Aba, and Alvan Ikoku Federal College of Education, Owerri for providing the platform and the right atmosphere for this work.

References Références Referencias

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 Alessan dro 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. VanNamee 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/.