The following paper was originally published in the
Proceedings of the USENIX Windows NT Workshop
Seattle, Washington, August 1997

Measuring Windows NT — Possibilities and Limitations

Yasuhiro Endo, Margo I. Seltzer
Harvard University

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Measuring Windows NT—Possibilities and Limitations

Yasuhiro Endo, Margo I. Seltzer
{yaz,margo}@eecs.harvard.edu
Harvard University

Abstract

The majority of today's computing takes place on interactive systems that use a Graphical User Interface (GUI). Performance of these systems is unique in that "good performance" is a reflection of a user's perception. In this paper, we explain why this unique environment requires a new methodological approach. We describe a measurement/diagnostic tool currently under development and evaluate the feasibility of implementing such a tool under Windows NT. We then present several issues that must be resolved in order for Windows NT to become a popular research platform for this type of work.

1. Introduction

In recent years, computer systems have become increasingly interactive, most often based on Graphical User Interfaces (GUI). In these systems, users interact with the computer far more frequently than in traditional computer systems, such as those based on a command-line interface or systems used for scientific computation or transaction processing. Another factor that makes these interactive systems different from conventional systems is that "performance" is determined by the user's opinion. This metric, *user-perceived performance*, is different from the performance metrics most commonly used, in that it is affected greatly by the subjective judgment and physical limitations of users. In order to measure and improve user-perceived performance, we need a new methodology that takes these factors into account. We are currently developing such a methodology and a set of diagnostic tools to gather data that will allow us to improve user-perceived performance.

This paper explains how interactive systems are different from conventional systems and why we must devise a new measurement methodology to evaluate interactive systems. As Windows NT is one of the most commonly used GUI platforms, we will evaluate the feasibility of conducting such research on Windows NT. We begin by identifying the differences between measuring interactive systems and conventional systems. We summarize our past efforts to measure interactive systems in Section 3. Section 4 presents the motivation for our current measurement project and outlines the design of the measurement/diagnostic tool currently under development. We then evaluate the feasibility of implementing this tool on Windows NT in Section 5. Section 6 outlines the problems that need to be addressed if NT is to become a popular research platform for this type of work. We present our conclusions in Section 7.

2. The challenges of interactive system measurement

Typically, we use benchmark programs to rate system performance. The most commonly used technique is to measure how quickly a system handles a sequence of requests; from this data, we calculate the bandwidth or the throughput of the system. Systems that achieve high scores in benchmarks are thought to have good performance, and scoring high in well-known benchmarks helps sell systems—both in the commercial market and in the research community. This is one of the reasons that a great deal of effort goes into making systems achieve high benchmark scores. We also use benchmark programs to guide us in the optimization process. A good set of benchmark programs helps us optimize the system effectively by identifying performance bottlenecks. However, it is often difficult to devise a good benchmark that stresses the system in a realistic manner. A bad benchmarking methodology is misleading, encouraging us to optimize parts of the system that have little or no impact on the performance visible to users.

The task of benchmarking interactive systems, such as Windows NT, is further complicated by adding a user into the performance equation. It is known that human judgment of performance is based roughly on the response time or the latency with which the system handles each user request. Generally speaking, collecting response time information in a benchmark is more difficult than calculating the throughput. Moreover, determining how different response times affect users' perception is difficult because perception is greatly influenced by factors such as the user's attitude, expectation, and physical limitations [10]. These factors make benchmarking interactive systems extremely difficult and unique. It is clear that we cannot simply apply conventional techniques to measure interactive systems.

Nonetheless, the majority of today's benchmarks use

conventional techniques that are inappropriate for interactive system measurement. First, these benchmarks often rely on throughput-based metrics. Interactive users do not evaluate system performance based on how quickly a system can respond to a sequence of requests, but how quickly the system can respond to each individual request. Throughput-based metrics do not capture how quickly the system was able to handle *each* of the requests nor do they report the variance observed for the different events. Second, it is considered good practice to use traditional statistical techniques to present our data. We strive for stable, statistically significant results and make every effort—such as removing the system from the network and rebooting the system before each trial—to ensure that the benchmark produces stable output. These experiments inaccurately model the environment in which the system is actually used and ignore the most important and interesting situations we can measure—anomalies. We, as users of interactive system ourselves, often experience situations in which an operation takes an unexpectedly long time for no apparent reason. We, as researchers, must work to eliminate these anomalies; they frustrate users and reduce the user-perceived performance of the system. Experts on human-computer interaction have long noted that expectation has a significant effect on user-perceived performance. Users are surprisingly forgiving when they are waiting for operations they expect to take a long time but are unforgiving when an operation that they expect to complete quickly does not.

We have been working continuously to narrow the gap between the techniques in use and an ideal interactive system measurement technique. We will explain what we have done so far, what we have learned, and on what we are currently working. We will then evaluate the feasibility of implementing the proposed system on Windows NT.

3. Previous measurement projects

In an effort to make systems research more relevant to the majority of computer users in the world, we have undertaken a number of projects to help us understand the performance of personal computer operating systems, such as Windows and Windows NT, and the applications that traditionally run on these platforms. In this section, we discuss these projects from the perspective of the benefits derived from using such a widely popular platform and the challenges it imposed.

3.1. The Measured Performance of Personal Computer Operating Systems

With the realization that the research community must understand more about commodity systems, we set out to measure and understand the behavior of commodity systems and how they compare to traditional research operating systems [2]. We measured and compared Windows NT 3.50, Windows for Workgroups 3.11, and NetBSD 1.0. We used the Pentium Performance Counters [4] to obtain performance statistics including execution time, on-chip cache miss counts, and segment register loads. The Pentium Performance Counters are accessible only from the supervisor mode. Accessing the performance counters under NetBSD was straightforward, since we could freely modify the kernel to introduce the code to manipulate the counter. Under Windows for Workgroups, we used the VxD interface, which allows any user program to introduce supervisor-mode code into the kernel and invoke it directly. Under Windows NT, we took advantage of its installable device driver interface. This interface allows a third party to implement and install a device driver into the NT kernel dynamically. Following the documentation provided in the Windows NT Device Driver Kit [8], we implemented and installed a kernel-mode driver that makes the Pentium Performance Counter registers appear as ordinary device files.

The tools we built allowed us easy access to the performance counters, but the lack of Windows and Windows NT source code limited our ability to interpret measurement results and draw useful conclusions. Explaining the results of benchmarks was made difficult by the fact that we could not confirm our suppositions by code analysis. In many cases, we had to write several new benchmarks to explain the results of one benchmark, and in many cases, writing and measuring the additional benchmarks did not help us fully explain the results. The lack of source code access also meant that we could not isolate and measure specific parts of the kernel. This limited our ability to explain and further understand interesting behavior that the systems exhibited.

Perhaps the most important lesson we took away from this project was the realization that throughput metrics do not always correlate with the user-perceived performance. One of the benchmarks in this study measured how quickly the systems executed a script using Wish, a command interpreter for the Tcl language that provides windowing support using the Tk toolkit [5]. The results of this benchmark were greatly affected by the aggressive optimization that NetBSD and the X-Windows system applied to the input stream. When many requests arrive at the server in a short period of time, the system tries to minimize the server-client communication overhead by sending multiple requests per communication round trip. We observed similar behavior from Windows

NT running Microsoft Powerpoint when processing ten page-down requests. The processor performed five to six times more work[1] when the requests were fed into the system at a realistic rate of about 10 characters per second than when all the requests were fed as quickly as possible. These optimizations help systems perform better on throughput metrics, but often have adverse effects on user-perceived performance. This observation led to our next study.

3.2. Using Latency to Evaluate Interactive System Performance

We set out to establish an appropriate set of techniques for measuring interactive system performance [3]. Previous measurement techniques relied almost entirely on throughput-based measures [1][7], ignoring the fact that throughput and user-perceived performance are different in today's popular GUI environments. User-perceived performance might coincide well with throughput in compute-intensive computations such as scientific computation and compilation, but not necessarily with more interactive applications, such as word processors, spreadsheets, and games.

In this study, we measured the performance of interactive systems using the response time that users experience when running commonly-used applications. We recognized that the response time or the latency of the system handling each user-initiated event, such as a keystroke, correlates better with user-perceived performance than does throughput. We designed and implemented techniques to measure event-handling latency in commodity operating systems and applications, and used these techniques to measure Windows NT versions 3.51 and 4.0 and Windows 95. Since we could not instrument real applications, we devised measurement techniques based on two assumptions. The first assumption was that the CPU is idle most of the time and becomes busy only when it is handling an event. The second assumption was that applications are single-threaded and only call the Win32 API `GetMessage()` to block waiting for new events, after they have completed handling all previous events.

The combination of these two techniques allowed us to measure the latency of user-initiated events *when our assumptions were met*. While we were able to measure how long a simple application, such as Notepad, spent handling each keystroke event, we were unsuccessful in measuring complex applications such as Microsoft Word, or more complex system states, such as multiple applications running concurrently. In terms of understanding user-perceived performance, we identified the crucial difference between throughput and user-perceived performance. Common cases dictate throughput performance—the parts of the system in which the most time is spent are ultimately reflected in throughput metrics. User-perceived performance is dictated by how frequently the user is annoyed and the extent to which the user is annoyed by each occurrence. No matter how frequently they occur, events with latencies below the threshold of user perception do not annoy the user and are therefore irrelevant in the user-perceived performance equation. Conversely, if an event takes sufficiently long to be annoying, its contribution to user-perceived performance is far greater than is suggested from its frequency or percent of total execution time.

4. A New Measurement Methodology

Although constructing a performance metric that captures the subtleties of user subjectivity is beyond the scope of our expertise, we can use the lessons learned in the prior studies to measure and improve the performance of interactive systems. Since the user's judgment of performance is subjective and events that annoy the user are important, we must capture the situations that annoy users. By understanding how these problems arise and correcting the system to avoid such situations, we can improve user-perceived performance.

The measurement system we are building monitors the system under normal operation with a real user on the console. This allows us to exercise the target system in a realistic manner. Controlled experiments based on artificial benchmarks yield stable, statistically significant, reproducible results but often fail to be realistic. In these test cases, it is common to disconnect the machine from the network and/or reboot the machine before each experiment so that the various caches in the system are in a known state. Unfortunately, this is not how most people use their systems, so the benchmarks do not accurately reflect actual use. Real systems often run multiple programs and daemons simultaneously and are constantly affected by external events such as packets arriving on the network. Many performance problems are created by these unpredictable interactions.

For our measurement system, we rely on the user to decide when performance is unacceptable. The user of the system notifies the measurement tool immediately—by clicking a button on the screen—after or while experiencing unacceptable performance. The tool then records the latency of the operation that frustrated the user and dumps data describing the last several seconds

1. Inferred from CPU occupancy time.

of system state leading up to the unacceptable behavior; data that allows us to diagnose the cause of the long latency event. The technical challenges are to identify the data we need to collect and determine how best to collect such data without imposing high overhead (either in time or storage). In the next section, we discuss the type of information we have determined necessary to collect and evaluate the possibilities of collecting such information under Windows NT.

5. Collecting data

Using the measurement system outlined above, we hope to be able to find instances in which interactions between processes and daemons, process scheduling policy, or disk scheduling policy is causing irritating delays in the processing of user-initiated events. Based on earlier work, we have determined some of the data that we must collect. In the sections that follow, we describe the data and the techniques for collecting such data in the Windows NT environment.

5.1. Latency of user-initiated events

Since user irritation is caused by slow response time combined with the expectation of fast response time, it is vital that we measure how long the system spends processing each event. We had moderate success collecting this data in our earlier studies, but we relied upon two simplifying assumptions: that there are no background tasks and that users run a single application at a time. We can no longer afford to make such assumptions. Real users often run more than one application concurrently and many applications are multi-threaded and perform background processing. In such an environment, the user and the application are the only parties who know when event-handling begins and ends. Since it is extremely difficult to instrument users, we must rely on applications to provide this information to the measurement system. This can be accomplished in one of three ways. First, the application can keep track of its own event latencies. This is likely to provide the most accurate data, but it is also the least practical in that it requires access to the application source code (not to mention actually modifying all the applications a user is likely to use). Second, we could use interposition to intercept every application call in a measurement library and then deduce the event latencies based on this trace. This process is feasible by substituting our own libraries for the standard DLLs, but interpretation of the trace output is error-prone. The third technique is for the operating system to try to extrapolate event beginning and ending times. This introduces the potential for the greatest margin of error and requires access to the operating system source code. None of these approaches is particularly desirable, and all require source code availability, which makes it troublesome to collect this type of data under Windows NT.

Even if we could capture latencies using one of these techniques, there is a component of response time that cannot be captured by the application alone. The time that an application spends processing a user-initiated event does not include the time that the system spends delivering a hardware event, such as a mouse click, to the application. Although we have yet to measure such a case in our constrained environment, it is likely that, in a real environment with multiple runnable threads, the delivery time could contribute significantly to latency. To measure event-delivery time, we must timestamp every keyboard and mouse interrupt and determine when the system actually delivers the corresponding higher-level event to a particular application. In Windows NT, timestamping keyboard and mouse events is easily accomplished by modifying the keyboard and the mouse drivers. However, associating a keyboard or mouse input to a higher-level event delivered to the application is challenging. Doing so requires instrumentation of multiple points in the path executed as an input event is processed. This cannot be done without OS source code access. While the installable driver interface provides the ability to collect statistics localized to the driver, it fails to provide us a way to examine or modify the other parts of the system.

5.2. Status of threads running in the system

To identify the cause of a performance problem, it is essential to know when the required thread was able to run, when it was not, and why. Windows NT has an extensive performance monitoring interface. Objects in the system such as threads, processes, processors, disks, and network protocols maintain a set of performance statistics that can be retrieved from user-level, as is done by the statlist program [9]. Using this interface, it is possible to determine the state of a thread, whether the thread is blocked, and the reason why it is blocked.

5.3. Queue States

If the performance of a system is limited because a critical thread is blocked, we must eliminate such waits or shorten their duration. One prime example of such a queue is the disk queue. Using the performance measurement interface described in Section 5.2, we can obtain various statistics including the number of read and write requests and the length of the queues. Unfortunately, the NT performance monitoring interface does not reveal the contents of the disk queue, because such information is localized to the disk driver. However, by

replacing the driver, we can identify the contents of the disk queue. Unfortunately, we cannot directly relate a specific device request to the thread that generated it. Determining this information requires that the driver examine thread states in the core of the kernel to which the device driver interface does not allow access.

5.4. Kernel profile

Kernel profiling can provide detailed information about the inner workings of the operating system, revealing where a thread is spending its time inside the kernel. In cases where a major component of the response time is due to the thread executing inside the kernel, profile output can help identify system bottlenecks. For this study, we need to extend conventional profilers, because we are interested in the data from a specific subinterval, namely the interval during which the user was waiting, not the interval during which the profiler was running. Ordinary profilers do not maintain enough information to calculate subinterval profiles. Typically, profiling requires having source code access to the system being profiled. While it might be possible to construct a profiling system using a binary instrumentation tool such as Etch [6], the information produced would be of limited utility without source code with which to analyze the data.

The requirement of the measurement tool described in Section 4 goes beyond the provisions of Windows NT. Table 1 summarizes our needs for constructing an interactive performance monitoring tool and identifies the classes of data for which source code access is required. In most of the cases, some data is available without source code, but source code is required to analyze the data and relate it to appropriate user actions. For example, although the installable device driver interface provides the ability to measure and experiment with some areas of the system, it fails to provide information about the kernel's inner workings. NT's performance monitoring interface also provides a convenient way to monitor parts of the system, but the disconnect between the performance monitoring tools and the device level interfaces limits its utility.

6. Difficulties in conducting research without source code

In this section, we discuss several of the difficulties we experienced conducting research using Windows NT. Each of the following problems is serious enough to make NT an unsuitable platform for some systems research. In order for NT to become a good research platform, both Microsoft and the research community must work together to address these problems. We have found that detailed performance analysis and understanding requires access to kernel (and application!) source code. First, without source, it is often impossible to confirm hypotheses about system behavior. In both of our previous measurement projects, the lack of source code access made it impossible to draw definite conclusions. In some cases, we were able to make definitive statements by taking several additional measurements to confirm our suspicions, but in many cases, we simply could not substantiate our suppositions—all we could do was increase the probability that our intuition was correct by taking additional measurements. While this is the norm in some natural sciences, it is particularly frustrating when we know that the answer is easily verifi-

Data Collected	Methodology	Source	
		Required for data collection	Required for analysis
Per-event latencies	Detailed accounting of threads' CPU activities.	NO	YES
	Instrumenting the application.	YES	YES
Thread status	NT performance monitoring interface.	NO	YES
Queue status	NT performance monitoring interface.	NO	YES
	Modify device drivers.	NO	YES
Kernel profile	Source code profiler.	YES	YES
	Binary instrumentation tool.	NO	YES

Table 1: Data Collection Summary. This table shows how we might collect the different types of data we need and whether we can collect such data in the absence of source code. Notice that in most cases, the utility of the data is diminished by the absence of the source code.

able.

In some cases, the lack of source code prevented us from pinpointing the exact cause of a performance problem. In systems with source code, we could often find the cause of the performance that we observed and, if desired, modify it and remeasure. In the case of NT, the best we could do was offer vague suggestions for alleviating system bottlenecks. The goal of many research projects is to find an exact answer to a question or solve a problem so that the system will perform better. Windows NT does not allow researchers to achieve this goal in many circumstances. This makes conducting research a frustrating experience.

While Microsoft does offer source code licenses for NT, we find that the restrictions are so limiting that they are in direct opposition to University policy. In particular, requiring students to sign non-disclosure or confidentiality agreements is problematic, particularly in the case of undergraduates. Equally problematic is prohibiting a class of students (those who come from certain foreign countries) from participating in projects that use source code. Finally, there is concern over publication. If we have access to source code, and use that access to explain results that we observe, does our confidentiality agreement prohibit us from explaining our results? That is certainly an untenable situation.

7. Conclusion

The unique manner in which user-perceived performance is determined demands that we develop new techniques to measure and improve interactive system performance. Windows NT is an attractive platform for such research. Conducting practical research that can solve the type of problems that millions of people experience every day excites many researchers. We have demonstrated that it is possible to perform interesting research on NT, but we are approaching the limit that a proprietary system places on the type of measurement research that can be conducted. Researchers expect and require more freedom from the platform they use; we need better hooks into the operating system to provide the detailed information we seek or fewer restrictions on source code licensing.

8. References

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

[2] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Mike Smith, “The Measured Performance of Personal Computer Operating Systems,” *ACM Transactions on Computer Systems 14*, 1, February 1996, pages 3-40.

[3] Yasuhiro Endo, Zheng Wang, J. Bradley Chen and Margo Seltzer, “Using Latency to Evaluate Interactive System Performance,” *Proceedings of the Second Symposium on Operating System Design and Implementation*, October 1996, pages 185–199.

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

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

[6] Dennis Lee, Ted Romer, Geoff Voelker, Alec Wolman, Wayne Wong, Brad Chen, Brian Bershad, and Hank Levy, Etch Overview, http://www.cs.washington.edu/homes/bershad/etch/index.html.

[7] M. L. VanNamee and B. Catchings, “Reaching New Heights in Benchmark Testing,” *PC Magazine*, 13 December 1994, pages 327-332. Further information on the Ziff-Davis benchmarks is available at: http://www.zdnet.com/zdbop.

[8] Microsoft Corporation, *DDK Platform*, Microsoft Developer Network, Redmond, Washington, January 1996.

[9] Microsoft Corporation, *Win32 SDK*, Microsoft Developer Network, Redmond, Washington, January 1996.

[10] Ben Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction (Second Edition*), Addison-Wesley, Reading, Massachusetts, 1992.