Improving Interactive Performance Using TIPME

Yasuhiro Endo
Network Appliance
495 East Java Drive
Sunnyvale, CA 94089
408-822-6483
yaz@netapp.com

Margo Seltzer
Harvard University
33 Oxford Street
Cambridge, MA 02138
617-496-5664
margo@eecs.harvard.edu

ABSTRACT

On the vast majority of today's computers, the dominant form of computation is GUI-based user interaction. In such an environment, the user's perception is the final arbiter of performance. Human-factors research shows that a user's perception of performance is affected by unexpectedly long delays. However, most performance-tuning techniques currently rely on throughput-sensitive benchmarks. While these techniques improve the *average* performance of the system, they do little to detect or eliminate response-time variabilities—in particular, unexpectedly long delays.

We introduce a measurement infrastructure that allows us to improve user-perceived performance by helping us to identify and eliminate the causes of the unexpected long response times that users find unacceptable. We describe TIPME (The Interactive Performance Monitoring Environment), a collection of measurement tools that allowed us to quickly and easily diagnose interactive performance "bugs" in a mature operating system. We present two case studies that demonstrate the effectiveness of our measurement infrastructure. Each of the performance problems we identify drastically affects variability in response time in a mature system, demonstrating that current tuning techniques do not address this class of performance problems.

Keywords

Interactive performance, monitoring.

1. INTRODUCTION

In recent years, computer systems have become increasingly interactive, usually employing Graphical User Interfaces (GUI) such as Microsoft Windows and the X Window system. In these systems, users interact with the computer far more frequently than in traditional batch or command-oriented computer systems, and they expect the system to respond to each user request instantaneously. As such, "performance" is determined by the user's opinion. This metric, *user-perceived performance*, differs radically from conventional performance metrics in two ways. First, it is largely subjective and is a function of the perceptual and physical limitations of users. Second, events that affect user-perceived performance are on a time scale of hundreds or thousands of milliseconds, not the microsecond scale that is often the target of detailed performance tuning.

Users' perceptions of performance are closely related to response time and the variability of response time, both of which can be quantified with moderate effort, using some newer tools and techniques [7][8]. However, to the best of our knowledge, there are no tools available for interpreting a collection of event latencies and determining which ones actually irritate users, which is why we rely on user input for this function. For example, if an event's latency is below the threshold of human perception, that latency contributes nothing to user irritation. Once a latency does cross over into the realm of perceptibility, there are no guidelines by which to assess the impact of the delay, but the relationship between delay and irritation is practically guaranteed to be nonlinear. Moreover, previous studies have argued that user expectation is a critical component of user-perceived performance [6][7]. There is a qualitative difference between a five-second delay echoing a keystroke and a five-second delay starting up an application. Unlike latency, expectation is difficult to quantify because of its psychological aspect and because it is partially a reflection of the performance characteristics of the system to which the user has become accustomed. As users become familiar with a system, they become trained to expect certain delays for each type of operation. While these delays may not delight users, users eventually adjust their behavior to long latencies to minimize errors and frustration [13][16][18]. The greatest contributor to "bad" user-perceived performance is when an event takes an unexpectedly long time to complete, without apparent reason [16]. Therefore, the key to improving user-perceived performance is to identify such situations, understand why they occur and modify systems to eliminate them.

The Interactive Performance Monitoring Environment (TIPME) is a measurement system that collects data that enables system experts to identify the cause of user-perceived performance problems that have previously been extremely difficult to diagnose. Unlike conventional performance-improvement techniques, we do not attempt to quantify system performance. Instead, we take advantage of user input to determine when performance becomes unacceptable. TIPME continuously monitors and records data that summarizes the operating system state. When the user experiences unacceptable performance, s/he presses a hot-key sequence which causes all the data currently stored to be saved for postmortem analysis and provides the user with a dialog box in which to enter a problem description. By understanding cases in which the user indicated that the system exhibited bad performance and eliminating their causes, we improve user-perceived performance.

The key contributions of this work are a methodology for attacking interactive performance problems, the design and implementation of a measurement infrastructure capable of capturing such problems, two case studies demonstrating the utility of the system, and a presentation of concrete examples where throughput-based system design decisions are detrimental to user-perceived performance. We also use our methodology to show that platforms other than our target platform demonstrate problems in similar areas.

Once we deployed TIPME, users immediately identified latencies that were annoying, and we were able to identify the problem and deploy simple kernel workarounds within a day or two. This rapid turn around was essential, because we found that users were likely to report the same problems if they were not resolved. The fact that we were able to do this rapidly in a mature operating system supports our hypothesis that the latency aspect of operating system performance has long been neglected. During operating system development, performance "bugs" are frequently introduced into the system. Prior to release of a new operating system, systematic testing and tuning usually enable the detection and elimination of such problems. However, the benchmarks that have been used are more sensitive to system throughput than they are to latency or latency variability. Therefore, these benchmarks help developers remove performance bugs that affect system throughput but do little to enable the diagnosis and removal of latency-related performance problems. Our measurement technique introduces a systematic way to identify and eliminate performance problems that affect latency. We suggest using our techniques during the beta phase of deployment to remove serious interactive performance problems.

In the next section, we discuss related work in performance and measurement methodology and human-computer interaction. We describe our measurement methodology in Section 3. Section 4 presents two case studies, describing how we were able to identify and correct interactive performance problems in the BSD/OS operating system. In Section 5, we demonstrate that some of the problems we identified in BSD/OS also exist in a radically different system (Win32), indicating that these techniques are applicable across different systems. We conclude in Section 6.

2. RELATED WORK

There have been efforts to use response time as the basis for system performance tuning. Application Response Measurement (ARM) measures response time directly by providing API functions that client programs call before and after an operation [8]. In earlier work, we inferred response times from CPU activity and message exchanges between MS-Windows clients and the server [7]. Both of these approaches assist in capturing event latencies, but they do not provide any indication of the cause of long latencies. This is the significant difference between such systems and the one we present here.

Cota-Robles and Held also use an infrastructure somewhat similar to ours to characterize the Windows NT and Windows 98 operating systems' ability to handle real-time workloads. They measure how quickly and reliably the systems deliver hardware interrupts to their corresponding handlers in a loaded system [3]. They find that the difference in real-time performance is not adequately represented by throughput benchmark results. Although Windows NT provided at least an order of magnitude better real-time response than Windows 98, throughput-based benchmark scores obtained by the Winstone benchmark [21] showed that both systems had throughput scores within 20 percent of each other.

DCPI is a continuous monitoring technique that attempts to measure the performance of hardware executing under normal conditions by continuously profiling a variety of hardware statistics [1]. TIPME also uses continuous monitoring, but the two systems are worlds apart in the abstractions with which they concern themselves. DCPI captures information about hardware resource usage, while TIPME captures information about high-level GUI events and transitions in operating system state. The difference in abstractions results from the difference in focus of the two systems: the main focus of TIPME is to identify and remedy operating system impediments to user-perceived performance, while DCPI's focus is to understand hardware behavior.

Although there has been much research in the HCI community evaluating the impact of latency on user performance, the context has been limited. Most of the research addresses typing performance and data-entry scenarios, concentrating on quantifying the relationship between latency and productivity issues, such as the error rate and the amount of work completed, rather than on the connection between latency and user satisfaction. Moreover, most of these studies have been conducted on non-GUI platforms, leaving user sensitivities to operations unique to GUIs, such as using a mouse to select a menu, unresearched [5][13][18].

3. METHODOLOGY

The TIPME system is a measurement infrastructure that relies on the interaction of three different agents. First there is the user, upon whom we rely to notify the system of a performance problem and describe that problem in sufficient detail that the poorly behaved application can be identified. Second, there is a collection of log processing scripts that we use to extract and process relevant information from logs. Third, there is a human system expert who interprets the extracted information and makes the ultimate diagnosis and suggested correction.

The main contributions of the system are the identification of the necessary information to log, a system for collecting them in an unobtrusive and low-overhead way, and an effective partitioning of the problem between human expertise and computer automation. In an ideal system, we would automate all processing and diagnosis, but there is much research to be done before such processing can be automated. First, there is no agreement as to the magnitude of latencies that begin to irritate users and we know that such thresholds are a function of the user's experience. Second, there are few techniques for making systems self-tuning, although we see this as a fruitful and active research area [15].

3.1 Identifying the Source of the Problem

The goal of our measurement methodology is to determine why systems sometimes spend an unexpectedly long time processing a transaction that ordinarily completes with acceptable latency.

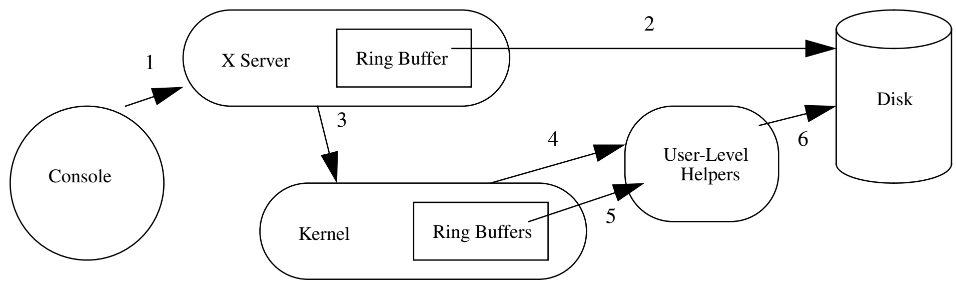

Figure 1. Structure of TIPME. Both the kernel and X Server contain non-paged ring buffer(s). Upon experiencing a problem, the user presses a hot-key combination, which is intercepted by the X Server (1). The X Server writes out the contents of its ring buffer to a file (2) and informs the kernel that the system is experiencing a problem by making a `sysctl` call (3). The kernel sends out `SIGUSR1` to its helper process(es) (4). In response, the user-level helpers read the contents of the in-kernel ring buffers and store the information to files (5, 6).

TIPME is a measurement infrastructure that enables us to collect information to determine the cause(s) of unexpected latency. Since the long latency events we are interested in occur unexpectedly, we use continuous monitoring to gather data about the system state, saving the data to disk only when the user indicates that a problem has occurred, and we then perform postmortem analysis to diagnose the problem.

The task of improving user-perceived performance is inherently iterative. Severe performance problems tend to mask smaller ones. As a result, users tend to report major problems repeatedly before reporting smaller ones, and different users tend to report the same problems. Therefore, we need to dispatch fixes as soon as users detect performance problems, so that they will uncover new problems As a result, the fixes we describe in our case studies are quick workarounds that let us deploy solutions rapidly. All three of the problems we identify in our case studies are research problems in their own right and warrant individual attention.

3.2 Data Collection

TIPME records process state (whether the process is running, runnable, or blocked, and if blocked, the event upon which it is blocked), context switch information, how and when events pass through the X Window server, and the owners of highly contested kernel resources. This information is stored in a collection of in-memory, non-paged ring buffers. These ring buffers are sized to hold 30 to 40 seconds worth of data to give the user enough time to indicate that there was a performance problem (on our system, that requires approximately 32MB of additional memory; on a faster machine, more memory will probably be needed). The user notifies TIPME of a problem by typing the hot-key combination, `Ctrl-Alt-Minus`, at which point, TIPME writes the statistics held in the ring buffers to disk. The overall structure of TIPME and how it writes data to disk in response to the hot-key combination are shown in Figure 1.

We implemented TIPME on BSD/OS 3.0 and X Free86 R6.3 running on Intel Pentium- or Pentium Pro-based personal computers. We chose the hardware platform for its popularity and the software platform for its popularity in our environment and the availability of the source code. The CPU cycle counter, available in both Pentium and Pentium Pro processors, provides cycle-accurate timestamps on all of the records that TIPME generates [9]. These timestamps are used to merge and order the records generated by the system.

TIPME collects data for two major purposes. The first is to identify the time interval during which the user encountered a perceived performance problem, and the second is to determine exactly what was happening in the system during that problem interval. The next two sections describe how the data we collect accomplishes both purposes.

3.3 Determining the Problem Interval

Our first challenge is to identify the start and end times of the system's handling of the problematic user request. At first blush, it seems that the obvious solution is to have the X client generate a record before and after processing a request initiated by the user. While X client assistance is desirable, it is neither necessary nor sufficient to identify the problem interval. The latency the user experiences includes not only the time the client spends processing the request, but also the time the kernel spends delivering user-generated events—such as keystrokes and mouse movements—to the X Server, and the time the X Server spends processing and passing the events to the corresponding client. The measurements taken by a client do not capture the entire processing path. Additionally, client-side measurements do not capture all of the time that the X Server spends processing requests generated by the client in response to the input event. Figure 2 illustrates a typical interaction that occurs as a result of a user input, such as typing a character in a word processing program.

In order to identify the time interval during which the user was waiting for the system to respond, we must determine when the user initiated the problematic transaction by generating a keyboard or mouse interrupt and when the X Server provided the visual feedback that signals the end of the transaction. To do so, we monitor when user input enters the system and how this input is transformed into one or more X events, what request(s) the client generates in response, and when the X Server finishes handling the resulting request(s). From the event log, we automatically extract the keyboard and mouse events and then manually match the appropriate request to the description supplied by the user. In our experience, this manual matching has been nearly instantaneous; it takes only a few seconds' glance at the log to find the triggering event. We also extract messages that reflect the updating of the graphic display so we can identify the end of the user event(i.e.,

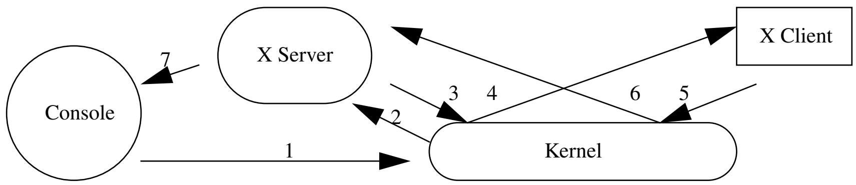

Figure 2. X message exchange. When the user types a key or engages the mouse, the hardware generates an interrupt handled by the kernel (1). The kernel sends a message to the X Server (2) which dispatches the event to the proper client, via the kernel (3, 4). The client then processes the request and sends a message back to the X Server (5, 6), and the X Server updates the display (7).

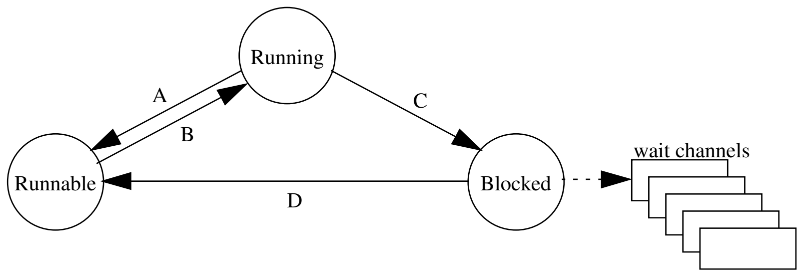

Figure 3. Process Scheduling Model. This simple state diagram depicts the model most frequently used in operating system scheduling. The process currently executing is in the Running state. When the scheduler decides to stop running this process, it traverses the state change depicted by A and enters the Runnable state which represents processes that are ready to run, but not currently scheduled. When those processes are rescheduled, they traverse edge B to re-enter the Running state. When a process cannot make forward progress due to I/O or a request for an unavailable resource, it traverses edge C and enters the Blocked state. Each process in this state is associated with a wait channel that corresponds to the event on which the process is waiting. Transition D represents the resource being made available, allowing the process to become runnable. At such a time, the scheduler may choose to immediately let it run, in which case it traverses edges D and B.

the message that updates the screen telling the user that the action has completed). Once again, we found that identifying the correct event is trivial. With this information in hand, we know the interval during which the delayed event took place. We have also identified the process responsible for this event. Our next task is to determine the cause of the unusually long delay.

3.4 Determining the Source of the Problem

In order to identify the cause of a specific delay, we need to understand the possible causes for all delays. If we consider the simple process scheduling model shown in Figure 3, then we can characterize the causes of unexpectedly long delays to be one or a combination of:

1. A change in the amount of CPU time required to complete the transaction (spending more time in the running state).
2. A change in the amount of time that the program spends waiting for I/O operations and/or the availability of resources (spending more time in the blocked state).
3. A change in process scheduling decisions (spending more time in the ready state).

The first source of variability in response time is the change in the amount of the CPU time that the application and operating system require to complete a transaction. Note that we are more interested in the change, not the absolute amount of CPU time that the operation requires. Users learn to expect average response times. Our techniques are designed to capture unexpected delays in which the response time deviates significantly from the norm.

Changes in the amount of CPU time that a transaction requires can occur for some operations, because the amount of computation required is variable and depends on the tasks previously performed. For example, the cost to search for an item on a linked list is highly dependent on where the target item is located on the list, which depends on the order of past insert operations. Both the application and the operating system perform operations with such variability and therefore, can change the amount of computation that they require to complete a transaction. The change in the amount of CPU time that the operation requires is easily detectable using profiling information.

Other than the changes in the amount of CPU work required to complete a transaction, the only remaining software causes for perceptible response time variability are 1) the program not being runnable while it is waiting for the completion of I/O or for the availability of a resource or 2) the operating system deciding to execute other programs. These correspond exactly to the scheduling states that any active, non-running process can have in the operating system. Therefore, to assist in diagnosing the causes of unexpected latency, we record context switches and changes in process scheduling state. Table 1 describes the information we collect and the following paragraphs elaborate on the collection process.

At every other timer interrupt (every 20 ms under BSD/OS), we collect the status of all the processes in the system. We record which processes are running, which are runnable, and which are blocked and for what reason. This information provides an overview of the system. We also record every context switch, sleep, and wake-up. This completely captures the system's scheduling decisions. Using this information, we can determine not only which process was running at what time, but also how long it took the system to schedule a critical process once it became runnable due to an external event, such as a keystroke.

The kernel records resource ownership by process ID (PID), so we need data that will enable us to construct the proper association between user commands and the PIDs in the system. In order to provide this data, we record the output of *ps*(1) when monitoring is initiated. Henceforth, we record the command line and the environment variables of each `exec`. Unlike the other information collected by TIPME, we cannot discard `exec` records in a simple FIFO manner, because process lifetimes can far exceed the 30–40 seconds of buffer space we maintain. Instead, we retain exec information for 10 minutes past the process lifetime (i.e., 10 minutes after the process has exited), so that we can diagnose problems in processes that have terminated before the user is able to report the problem.

When we receive TIPME output from a problem event, we identify the problem interval as described in Section 3.3. In the process of determining the interval, we also identify the process that was delayed. We then automatically extract the information that shows the amount of time the particular process spent in each scheduling state. Using this information, we manually determine the exact cause of the unexpected delay. If the source of the variability is a change in the amount of work that the transaction performs, we use profiling information to determine where the extra time is being spent and make algorithmic changes. If the problem is a scheduling decision, we examine the set of scheduling decisions and determine what prevented the process from being scheduled in a timely fashion. Finally, if the source is a resource wait, we study how this highly-contested resource is being used. In all the cases we uncovered to date, it took us only a few (5–10) minutes of manual processing to either identify the problem or decide what additional data were needed.

3.5 Implementation Details

There are three major components in TIPME—the kernel component, the X Server component, and the user-level helper. The kernel component collects the operating system statistics described in Section 3.4; the X Server component records X Server statistics and the message exchanges between the kernel and the X Server and between the X Server and X clients (Section 3.3); the user-level helper ties the other TIPME components together and provides the interface to control TIPME and extract the information collected. In order to keep the measurement system tractable, we do not require any instrumentation of client programs. The following subsections explain each of these components in detail.

3.5.1 Kernel Components

The kernel portion of TIPME consumes 24MB of physical memory. TIPME uses its own memory allocator to manage this memory. Whenever possible, we perform allocation and initialization during system start-up, so that we avoid the overhead of dynamic memory management. The only time we are required to allocate space dynamically is when recording `exec` information, because the length of the command-line arguments and the size of the environment is variable.

We modified the console driver to trap the following key sequences. `Ctrl-Alt-Plus` causes TIPME to start the monitoring system. `Ctrl-Alt-Minus` notifies TIPME that the user has experienced unacceptable performance. The kernel sends the user-level helper a SIGUSR1 that instructs it to retrieve and save the contents of the TIPME buffer. `Ctrl-Alt-0` and `Ctrl-Alt-1` disable and enable the keyboard logging portion of TIPME, so that users can prevent TIPME from recording sensitive keystrokes, such as passwords.

3.5.2 X Server Modifications

The X Server portion of TIPME uses 6MB of nonpageable (locked) memory for its ring buffer. Events such as the arrival of a character from the keyboard, an X event structure sent to a client,

	Description	Use
Process Status	For all processes, record the state (running/runnable/blocked) and priority.	By examining process states over time, we can identify which state contributed most significantly during a particular interval.
Context Switch Information	When processes acquire/release the CPU.	This allows precise tracking of when processes are running.
Resource Usage	How and when highly contested resources are requested and acquired.	Once we have identified resource contention as the cause of a delay, this data lets us determine why the contention arose, e.g., due to repeated requests or an inadvertently long wait.
Exec Information	A mapping of program name to PID.	We use this to map a user's problem description to one or more processes in the system.

Table 1: Information Collected

and an X request structure received from a client are recorded in this ring buffer.

Ordinarily, the X Server has no information about the process ID (PID) of the clients with which it is interacting. This is understandable since the X Windows protocol allows clients running on one host to connect to an X Server running on another host. In such an environment, the client's PID is of little use as an identifier. However, since most of the clients connected to the X Server are running locally in our environment, the PID of the client can often serve as a useful identifier. Knowing the PID of the client allows us to correlate information collected by the X Server portion of TIPME with information collected by the kernel portion of TIPME.

In order to allow the X Server to associate clients with PIDs, we made a small modification to the X library so that the client passes its PID in an unused pad field of a connection setup packet. This modification required that we relink the standard suite of X clients, including `xterm` and `twm`, distributed with the XFree86.

When the console is executing the X Server, console input is passed to the X Server in raw format where each keystroke is reported, not as a character, but in the form of key-down and key-up events. During the execution of the X Server, we no longer trap various hot-key combinations in the kernel. Instead, we modified the X Server to trap and process the four hot-key combinations described in the previous section. Upon trapping a hot-key combination, the X Server portion of TIPME notifies the kernel portion that the hot-key combination has been pressed by calling `sysctl`. The kernel portion responds to the `sysctl` call as if a corresponding hot-key combination had been pressed. Unlike the kernel portion of TIPME, which relies on the user-level helpers to write the buffer contents to disk, the X Server portion of TIPME writes its own buffer contents.

3.5.3 The User-Level Helper

The user-level helper is a simple process that spends most of its lifetime sleeping, waiting for the SIGUSR1 signal that gets sent on TIPME shutdown. When the user-level helper is awakened, it uses the `kvm`(2) interface to copy data from the in-kernel buffer to user space. The helper then writes this data to disk.

We perform postmortem analysis using several perl scripts linked with the Berkeley DB package [17]. These scripts process the raw data, generating human-readable output.

3.6 Overhead of TIPME

As mentioned in Section 3.5, TIPME consumes a large amount of memory (30MB). The kernel portion of TIPME consumes 24 MB of physical memory, which is allocated at system bootup. The X Server portion of TIPME consumes 6 MB of nonpageable memory acquired via the `mlock`(2) interface. In order to isolate the effects of consuming such a large amount of memory from the performance our users observe, we equip our machines with an extra 32MB of memory before installing TIPME. While such memory consumption may seem excessive, the incremental cost of memory is trivial (e.g., approximately $20 for an extra 32 MB). The kernel and the X Server code expansion are minimal at 9KB and 22KB, respectively.

Event	Event Latency (incl. overhead)	TIPME overhead	%-age TIPME overhead
Moving a mouse pointer	0.3 *ms*	80 *us*	27%
Typing a character in a Xterm Window	2.0 *ms*	340 *us*	17%
Displaying the file menu in Netscape 3.0	470.0 *ms*	5100 *us*	1.1%

Table 2: Latency and TIPME overheads. These latencies were measured using the Pentium cycle counter, which introduces little measurement overhead.

The runtime overhead of TIPME is low and reasonably constant. Table 2 shows the typical cost of generating each type of TIPME record. A more important and useful overhead statistic is how much of the latency that the user experiences is due to the TIPME overhead. To determine this, we label each TIPME record with the time it took to create the record. During the postmortem analysis, we add up the cost of generating all the records between the beginning and the end of the latency that the user experienced. Table 2 shows some typical latencies and TIPME overhead for common events measured using a 100 MHz Pentium PC with 64 MB of memory. As can be seen from Table 2, the TIPME run-time overhead can be a significant percentage of an event's latency when the event is sufficiently short. However the overhead is negligible when compared to the limits of human perception, which are on the order of tens of milliseconds [16].

3.7 Limitations

TIPME was designed to be used in an environment where all users have their own machines and perform most of their daily computation on those machines. Therefore, TIPME measures latency experienced by the console user. While it is possible to use TIPME output to determine the source of problems experienced by remote users, TIPME cannot account for the communication delay between the measured machine and the console at which the remote user is located.

Our methodology concentrates on diagnosing the sources of user-perceivable delays, which are typically at least several tens of milliseconds and often as long as several seconds. The data we collect have sufficient detail to diagnose events with these latencies, but they are sometimes too coarse to diagnose sub-millisecond delays. Fortunately, sub-millisecond delays do not impact user-perceived performance.

There is also a limit to how much understanding we can gain about the source of the delay. We treat some sources of delay as black boxes—some to make the problem simpler and others because it is necessary. We assume that application programs do not schedule their own threads. We also treat network-related delays as a black box. Consider a client-server architecture with our measurement infrastructure deployed on the client machine. Typically, the client program will perform a network I/O waiting for a response from the server. The measurement infrastructure will recognize this delay only as network I/O delay although such latency is a combination of network transfer latency and the latency with which the server provides response, which can be further broken down into CPU time and wait time, in the same way we have broken down the response time of the client machine. Since such a diagnosis requires us to deploy our infrastructure on both client and server machines and coordinate their activity, we have left it for future work.

We are also unable to determine the cause of device (e.g., disk) misbehavior. Intelligent disk drives can sometimes exhibit unexpected behavior, such as taking several seconds to complete an I/O request for no apparent reason. Our infrastructure will identify when such a device is the cause of long latency, but it cannot determine why the device behaved in such a manner.

Finally, our methodology sometimes requires us to re-instrument the system and re-measure the problem. The basic set of data we collect is sufficient to diagnose all detectable performance problems except for resource contention problems. The data allows us to determine on which resource the latency-critical process is blocked, but in order to correct a resource contention problem, we not only need to understand which resource is contested but also how the resource is being consumed. This requires that we collect additional information about how the resource is allocated and freed. Currently, we have adopted a delayed-instrumentation strategy of adding instrumentation points as new contested resources are identified. In a commercial system, we envision including full resource accounting that can be enabled optionally on a per-resource basis. Thus, once resource contention is identified, it would be a simple matter to enable collection of the necessary resource information.

4. CASE STUDIES

In this section, we demonstrate TIPME's utility in identifying system problems that lead to poor user-perceived performance. We deployed TIPME on two workstations, one with a 133-MHz Pentium PC processor and a second with a 200-MHz Pentium Pro processor. We asked the users to signal unacceptable performance using TIPME's hot-key combination and then waited to receive data. The typical tasks performed on these machines are editing, compiling, and web browsing. We used a third machine (the 100-MHz Pentium PC mentioned earlier) as a microbenchmarking and test machine. In the remainder of this section, we demonstrate how TIPME helped us to identify problems with the scheduling algorithm and to determine highly contested resources and their use.

4.1 Multi-second Console Pause

The first problem that a user reported was that the console became completely unresponsive for several seconds. This problem was observed when heavy jobs with frequent disk I/O, such as a kernel build, were running in addition to the interactive foreground process. A quick manual inspection of the TIPME logs immediately revealed that the X Server process was blocked during the problematic interval waiting on the `swpgiobuf` wait channel. This made the console unresponsive to user input.

We searched the system sources for the `swpgiobuf` wait channel and learned that when the VM system initiates a page- in or page-out request, it acquires a `buf` structure, which is used to describe the specifics of the I/O. BSD/OS' VM system maintains a pool that contains a fixed number of these structures[1]. When there is a shortage of `buf` structures, processes go to sleep on the wait channel `swpgiobuf` waiting for a buffer to become available. During the problem interval, the TIPME output reported that the X Server was blocked on this wait channel as follows:

```
62007.6920 sec cost 141.9 us
pid: 207 is blocked on f0119610(swpgiobuf)
```

The first line shows the time at which the record was collected and the time required to generate the record. The second line shows that process 207 (the X Server) is blocked on the wait channel at address 0xf0119610 and that the name of the wait channel is `swpgiobuf`.

In order to understand how such contention arose, we modified TIPME to record the usage of these structures. We redeployed TIPME and waited for the problem to reappear. Another quick inspection of the TIPME output indicated that during the problematic interval, the page-out daemon was monopolizing all of the available `buf` structures to initiate page-out requests. Since no `buf` structures were available to initiate the page-in request on behalf of the X Server, it was blocked, rendering the entire console unresponsive to user input.

4.1.1 Reproducing the problem using microbenchmarks

While these problems occur infrequently during actual use, once we understand their cause we can create a microbenchmark that reproduces the exact behavior. We constructed a microbenchmark that created enough memory pressure to cause the page-out daemon to monopolize all the VM `buf` structures, and we verified that the same problems arose by examining TIPME output during the microbenchmark. The benchmark consists of a timing process and a number of child processes. The timing process sleeps for 100 ms, reads a word from a 4MB buffer, and records a timestamp by reading the CPU's cycle counter [9]. The buffer is referenced cyclically with a 4KB stride, which is equal to the page size used by the Pentium and Pentium Pro processors. The child processes generate memory pressure by continuously writing a single word to their own 12MB buffer, also using a 4KB stride.

This benchmark measures how promptly the system processes an event that involves a potentially faulting memory reference while the system is experiencing heavy memory pressure. Ideally, the time stamps generated by the timing thread will be approximately 100 ms apart. Any delay in handling them will lengthen the

1. The number of preallocated `buf` structures is determined by the amount of physical memory in the system. Our two machines pre-allocated 64 and 50 such buffers, respectively.

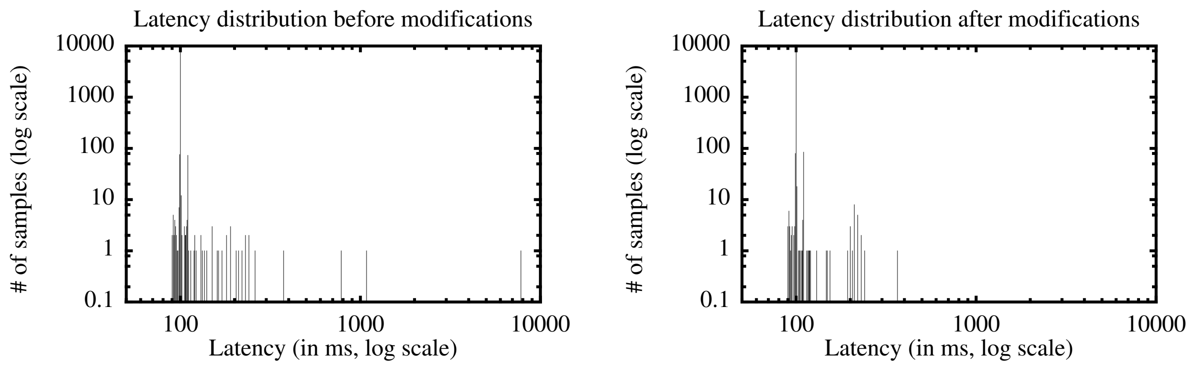

Figure 4. Latency Distributions Before and After System Modification. The left graph shows the latency distribution we observed on the unmodified system using our microbenchmark. The maximum observed latency was over 8 seconds and we had two data points in excess of a second. The right graph shows the latency distribution after changing the buffer allocation and disk scheduling strategies. After the changes, the maximum observed latency is approximately 0.37 seconds.

interval between two timestamps. (The 100-ms delay was selected to model the inter-arrival time of fairly rapid keyboard input.) We ran these benchmarks with the test machine running in single-user mode and disconnected from the network. We believe this artificial environment is justified, since we are trying to reproduce a specific problem that we observed under normal use, and eliminating unexpected external interference allows us to isolate the problem.

We varied the number of child processes from one to eight. When the number of children reached four, the system began to exhibit the problem we were trying to reproduce. The intervals between two time stamps recorded by the benchmark program often grew longer than one second. In some cases, the interval was nearly eight seconds. We used TIPME to examine the state of the system during such problems and confirmed that the cause of the delay was the timing thread blocked on the `swpgiobuf`.

4.1.2 Finding a Remedy

Diagnosis of the problem motivates and enables the creation of better performing algorithms, which can be evaluated using the microbenchmark we created. Although devising a complete solution to this problem is not the goal of this work, it was necessary to devise a simple workaround so that we could uncover other (unrelated) problems. If we did not remove this performance problem, users would have kept reporting the same problem.

We made ten of the `buf` structures unavailable to the page-out daemon. This is sufficient to allow the measurement thread and all the load-generating child processes to make forward progress, even under heavy page-out traffic. We reran the microbenchmark, but were disappointed that we did not observe any significant improvement in the system's behavior.

We turned to TIPME to help us identify the new problem. Inspection of the TIPME logs revealed that our change did prevent the page-out daemon from monopolizing `buf` structures, but fixing that problem revealed a second problem. The measurement thread was now waiting for paging I/O to complete on the `swpgio` wait channel. By examining the use of the `swpgio` wait channel, we observed that the page-out daemon had initiated 30–40 page-out requests and that they were being queued ahead of the I/O request that was issued in response to the timing thread's page fault. BSD/OS uses the CSCAN [12] algorithm for disk scheduling. CSCAN is designed to improve disk throughput by ordering disk requests to minimize seeks, but does so at the expense of individual request latency. The system stalled for several seconds handling a page fault, because it was queued behind the paging requests.

To temporarily work around this problem, we changed the disk request ordering algorithm into *FIFO with skip ahead*. Under this simple algorithm, all requests except page-faults are queued in a FIFO manner. The page-fault requests are placed ahead of other requests in the disk queue (i.e., "skipped ahead") as long as the following conditions are met: (1) page-fault requests do not skip ahead of other page-fault requests, and (2) when skipping over page-out requests, there must be at least three page-out requests ahead in the queue. The goal of this algorithm is to handle page-faults quickly and, at the same time, allow the page-out daemon to maintain a large enough pool of free physical pages in the system. We determined the value of three experimentally by examining the TIPME output, making sure that threads were not blocked on the `thrd_s` wait channel, which indicates a shortage of free pages.

Figure 4 shows the results of the microbenchmark before and after our modifications. The microbenchmark collects 10000 time stamps from a single trial. The data shown are in histogram format using 10-ms buckets. The results show that our changes greatly reduce the severity of the problem. The maximum response time drops from eight seconds to approximately 0.4 second.

We expected the disk queueing algorithm to reduce system throughput, but the time to build the BSD/OS kernel actually improved from 861.2 seconds to 839.3 seconds[2]. This result was unexpected, but we hypothesize that because kernel builds are single-threaded, faster page-fault handling is more important than potentially lower disk throughput. Though we have not been able to demonstrate the negative impact of our changes to system throughput, we expect that these changes will worsen system

2. These figures are the mean of five runs. The standard deviation was less than 0.2% of the mean.

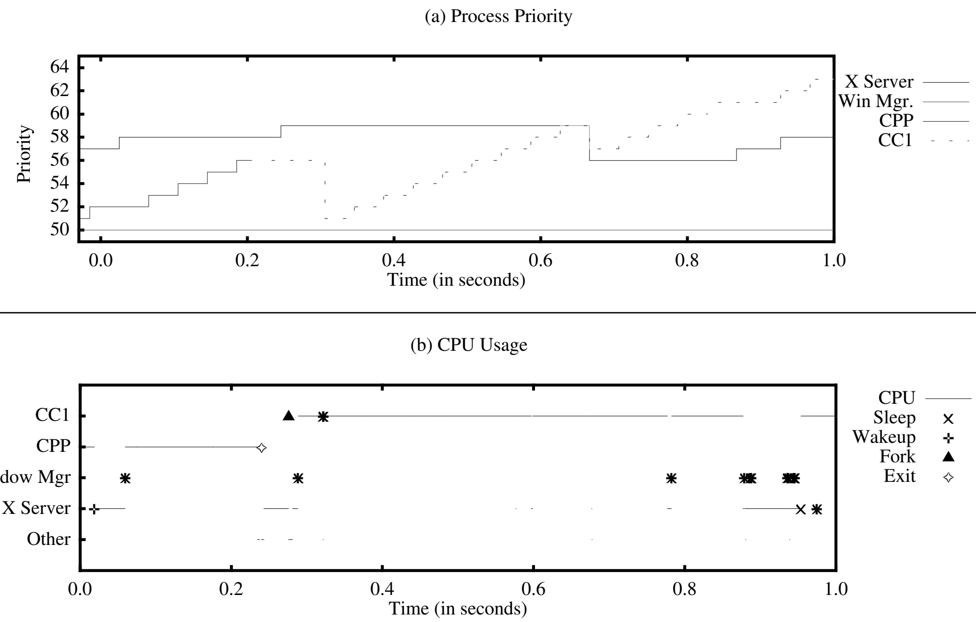

Figure 5. Process priority and CPU usage: These graphs show the priority level (a) of the relevant processes and how they consumed CPU (b) during the problematic time interval. These graphs were derived from the TIPME output which showed that the system spent most of its time executing compiler processes instead of the X Server because the compiler processes were able to attain higher priorities than the X Server.

throughput in some cases. However, we have shown that the throughput-centric disk scheduling algorithm actually does cause a very real and user-perceptible performance problem, and that a different and "worse" (with respect to throughput) disk scheduling algorithm improves user-perceived performance. Although it is beyond the scope of this paper, we believe we can design a better scheduling algorithm that will provide a compromise between throughput and latency, or perhaps, a correct solution may be to employ a different disk scheduling algorithm depending on the specific needs of the system (e.g., using different scheduling algorithms for workstations and servers).

Another point worth noting is that our use of microbenchmarking allowed us to discover and fix both first and second order problems at once. Without our microbenchmarking effort, we would have declared victory once we fixed the buffer allocation problem and would have needed another user's dissatisfaction to alert us to the disk scheduling problem.

4.2 Sluggish mouse pointer movement

The second performance problem we captured was described by the user as, "Sluggish mouse pointer movement when there was a compile job running in the system." During the problem interval, the system spent up to a second updating the mouse pointer in response to the user's movement. Operations such as mouse pointer movement are designed to provide an illusion of physical connection between the input device and what appears on the screen. These continuous operations are more latency critical than discrete operations such as the echoing of keystrokes. MacKenzie and Ware showed that the speed and accuracy of mouse pointer movement does not change in a significant manner when the latency of mouse pointer update changes from 8.3 ms to 25 ms but that both speed and accuracy worsen in a measurable way when latency is increased to 75 ms [11]. This result suggests that the threshold for acceptable mouse pointer update latency is somewhere between 25 and 75 ms.

The data we collected using TIPME showed that during the problematic interval, the process scheduler favored compilation jobs over the X Server, which handles the task of mouse pointer update. As a result, it took 850 ms for the system to process mouse input. This finding is surprising because BSD/OS UNIX uses a priority-based scheduler that favors interactive processes over CPU-intensive processes such as compilation. This scheme does so by monitoring each process' CPU usage and lowering the priority of processes that frequently consume their full scheduling quantum. In this particular instance, the scheduling algorithm was not working as designed.

Figure 5(a) shows the change in the processes' priorities during the problem interval. Under BSD/OS UNIX, a numerically smaller priority indicates a higher priority. Each process executing user-code is given a priority level (numerically) larger than or equal to 50. The system divides all the processes into one of 32 priority classes by putting processes with similar priority levels into one priority class. Processes in a lower priority class are executed only when there are no runnable processes in any of the higher priority classes. Processes within the same priority class are executed in a round-robin fashion.

Figure 5(b) shows how processes consumed CPU time during the problem interval. The solid horizontal lines indicate that the corresponding process on the Y-axis was executing during the depicted time interval. The graph also shows the process state changes that affect scheduling decisions such as birth(fork), death(exit), sleep, and wakeup. From this graph, we can observe when the X Server is awakened in response to mouse input and when it went back to sleep after handling the mouse input. The graph shows that during the problem interval, the system spent most of its time executing compile jobs, CPP (C preprocessor) and CC1 (C compiler), instead of the X Server.

Although the X Server is an interactive process, the priority of the X Server (58) is lower than expected. Most of the other interactive processes such as command-line shells usually show the highest possible user priority of 50. The X Server's lower priority reflects the fact that the X Server has been performing computation including previous mouse pointer updates and updates of an `xterm` window in which the compile job is executing. Although the latter task was performed on behalf of an X client, the X Server is charged for the computation, and as a result, its priority is lowered.

In comparison, each compile job is initially given a higher priority level than that of the X Server. Although newly created processes initially inherit their priority from the parent, the scheduler soon recalculates their priorities based on their past CPU usage history. The initial rise of CC1's priority observed around time index 0.34 is due to this recalculation. As CC1 has little or no past CPU usage history, the scheduler assigns high priority to the compute-bound CC1 process. The lack of CPU usage information causes the scheduler to assign a high priority to newly created processes regardless of the processes' true CPU usage characteristics.

As a result of this oversight, a newly spawned compile job initially attains a high priority despite being compute-bound. It takes several hundred milliseconds for the scheduler to build up enough CPU usage history to adjust the compile job's priority to be lower than that of the X Server. In some cases, such processes terminate before the scheduler accumulates enough information to make effective scheduling decisions. This several hundred millisecond delay in adjustment is sufficient to starve the X Server, resulting in perceivable, sluggish mouse pointer movement. The symptom is especially bad in situations in which many compute-bound child processes are created repeatedly, such as during a build. Each child can hinder the progress of a latency-critical process for several hundred milliseconds.

4.2.1 Reproducing the problem using microbenchmarks

The cause of the problem is the scheduler granting newly-created processes high priority regardless of the processes' true CPU usage characteristics. Newly-created, compute-bound processes are allowed to use up more than their appropriate share of the processor until the scheduler collects enough data to adjust the priorities accordingly. This problem is magnified when a stream of new processes is introduced into the system. By continuously introducing compute-bound jobs with high initial priority into the system, a parent process tricks the scheduler into allocating more CPU time to its compute-bound children, starving other processes, including latency-critical ones.

The microbenchmark we constructed consists of a measurement thread and one or more load generating threads. The measurement thread executes a loop that performs a computation that takes approximately 10 ms of CPU time followed by 50 ms of sleep time. We selected the duration of the computation and sleep intervals such that the priority of the measurement process would stay around 58 to approximate the priority level of the X Server when the system experienced the performance problem. At the end of the each loop iteration, this thread records a timestamp. In our benchmark run, we set the number of loop iterations to generate 10000 intervals between timestamps.

If the measurement process is the only process in the system, the timestamps that the process generates should be spaced at about 50 + 10 = 60 ms. A small deviation in this value is expected when other processes are present in the system. However, an excessive (several hundred milliseconds or more) deviation is an indication that the CPU scheduler made a bad decision.

To ensure that the source of the problem is the creation of compute-bound processes and not simply their existence, we run the test under two different load conditions. The first load condition involves one CPU-bound process that executes an infinite loop. The second load condition involves a thread that forks a compute-bound child once every two seconds. The parent thread sleeps between fork operations. The child executes an infinite loop, but the parent always terminates the child process before forking a new one so that there is at most one child present in the system at any time. The time interval is selected to model a job such as a build that repeatedly spawns compute-bound processes.

We ran this benchmark on our test machine. With a single, long-running compute-bound process, all the timestamps reported the expected 60 ms latency. However, with a series of short-lived compute-bound processes, the results were scattered; the recorded intervals ranged between 60 and 600 milliseconds with a significant number of them (nearly 10%) over 150 milliseconds and seven iterations requiring 561 milliseconds to complete. Using TIPME during the benchmark run, we verified that we had recreated the exact problem our users saw in practice.

These benchmarks demonstrate that the source of the performance problem is not the existence of compute-bound processes in the system but the frequent creation of compute-bound processes. The scheduling algorithm treats a newly created process as if it is I/O bound until the process accumulates sufficient CPU usage information. This allows these young processes to delay the execution of other processes in the system.

4.2.2 Finding a remedy

There are two underlying factors to the sluggish mouse movement problem. The first is the way that the scheduler calculates the priority of newly created processes, and the second is the fundamental way in which the scheduler performs CPU-usage calculation. The system charges all the CPU time a process consumes to the process that performed the computation regardless of the beneficiary of the computation. In this particular problem, the X Server was penalized for the computation it performed on behalf of the X client, which is an `xterm` program, that was displaying the output generated by the build process.

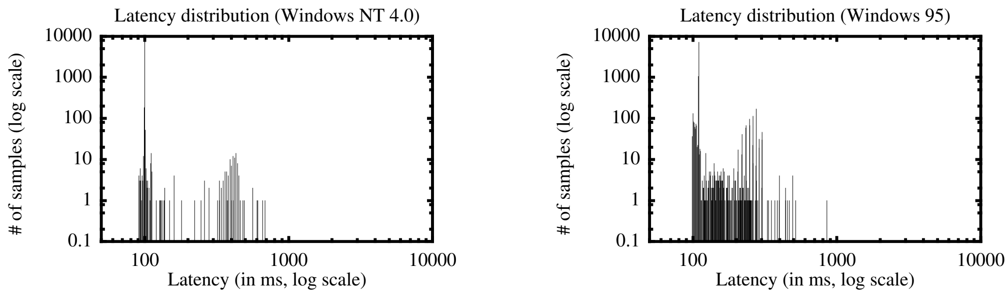

Figure 6. Latency Distribution under Windows 95 and Windows NT 4.0. We executed the microbenchmark described in Section 4.1.1 under Windows 95 and Windows NT 4.0. The left graph shows the latency distribution under Windows NT; the right graph shows the latency distribution under Windows 95. Although the longest latency observed is not as long as those observed under our target platform, the latencies are well above the limits of human perception.

Correcting the above problems completely requires an extensive redesign of the system's scheduler. We believe ideas such as lottery scheduling [20] and resource containers [2] can be used effectively to tackle this problem. Although finding such solutions is not the target of this study, we still must prevent this problem from occurring in order to find other problems. As a temporary workaround, we modified the kernel so that the priority of the X Server is fixed at 49, one level higher than the highest possible user priority. This change has kept the problem from reappearing and made the response of the mouse perceptibly better, even when the system experiences a high rate of process creation.

5. GENERALITY OF TIPME

In this section, we discuss the challenges to overcome in order to use TIPME in other environments and then use the microbenchmarks devised in the previous section to determine if other systems experience performance problems under similar load conditions.

Obtaining the information we needed required instrumenting the X-Server. Unfortunately, such instrumentation must be part of the main event loop, so the code is not easily extracted for use with a different window system. However, the changes are quite localized, and as such, could be applied to other window systems without much difficulty. The kernel changes are, for the most part, more modular. The kernel sampling and ring buffer management code are fully encapsulated as their own small subsystems. Unfortunately, there are hooks into these subsystems from memory management, process handling, and console management. Additionally, the system is tightly integrated with the memory management and process structure of BSD/OS. Fortunately, the total number of lines of code is approximately 2000, which means that porting the system to an entirely new operating system is not an unwieldy task.

Even without porting TIPME, we can assess its generality at finding interactive problems by running the microbenchmarks developed for BSD/OS on other platforms. We ported the microbenchmarks to the Win32 programming environment [14] and ran them on Microsoft Windows 95 and Windows NT 4.0.

The first microbenchmark tested how consistently the system performed a short task introduced every 100 ms when memory-intensive tasks were present in the system. Figure 6 shows the results for Windows 95 and Windows NT 4.0. Although the longest latency is not as long as those we observed under BSD/OS, there are still cases in which the system does not handle the task in a timely manner. In both the Windows 95 and Windows NT measurements, there are cases in which the system spends nearly a second completing the task. Such latency is well above the human perceptual threshold for simple interactive tasks such as echoing keystrokes.

We also ran the benchmark we used in Section 4.2.1 under Windows 95 and Windows NT 4.0. The benchmark results show that neither system experiences the problem, reliably scheduling the measurement process within 60 ms of the process becoming runnable. Custer states that the Windows NT scheduler increases a thread's priority when the thread is unblocked [4], and according to King [10], the Windows 95 scheduler uses a similar policy, boosting the priority of threads when they become runnable. These policies are designed to provide good response time to interactive processes that have a tendency to block frequently. This is in stark contrast to BSD/OS's scheduling policy, in which the processes can only receive an increase in priority indirectly by not consuming the CPU[3]. We believe this policy difference is the reason that both Windows NT and Windows 95 performed better than our target system.

Although only one of the two problems we discussed manifests itself in the Win32 systems, personal experience indicates that these systems also suffer from variable and unacceptably long delays. We believe that the experience we gained in understanding what and how to instrument BSD/OS is directly applicable to these other environments, and that an implementation of TIPME on Windows 95 or NT would help identify and correct problems in those systems.

3. The BSD/OS process scheduler temporarily raises the priority of unblocked processes, but its effect is limited to the time that the thread is executing inside the kernel. The purpose of this priority manipulation is to allow threads that can hold critical kernel locks to exit the kernel quickly—not to improve interactive response time.

6. CONCLUSIONS

The definition of performance is unique under interactive systems in that it is based on users' perceptions, not on easily quantifiable metrics. User-perceived performance is affected by latency. In particular, it is greatly affected by unexpected latencies. In this paper, we have described TIPME and demonstrated how it can be used to identify and help us remedy such long latencies. The causes of the performance problems we discussed here were inappropriate scheduling decisions, resource contention, and a disk scheduling algorithm that favors throughput over latency. Although our simple workarounds are not sophisticated enough to be complete solutions, we have shown that we can reduce both the frequency and the severity of such problems.

The research and commercial communities have been relying heavily on throughput-based benchmarks, tuning systems to improve throughput. These techniques are still useful aids to improving average-case performance. However, the popularity of single-user, interactive systems, such as those based on GUIs, has made user-perceived performance more important than ever. We must recognize that uncommon cases with little effect on overall system throughput or average-case performance are important determinants of user-perceived performance, and we must begin using infrastructures such as TIPME to eliminate the infrequent performance problems that irritate users.

There are a number of ways in which we can improve the sophistication of TIPME. The first is to remove the human user from the evaluation cycle, permitting a much quicker and more extensive evaluation. We are in the process of characterizing user profiles and deriving models of users' tolerances for latency that will enable us to automatically detect "unacceptable" performance. When this work is complete, we can automate much of our system testing and, ideally, produce many more cases of bad system performance. We expect that the diagnosis and correction of these problems will remain a manual process for the foreseeable future, however, ongoing work in self-tuning systems holds promise as a means for automating this process as well.

7. REFERENCES

[1] Jennifer M. Anderson, Lance M. Berc, Jeffery Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoore, Carl A. Waldspurger, and William E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?," *The Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997, pages 1–14.

[2] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul, "Resource containers: A new facility for resource management in server systems," *Proceedings of the Third Symposium on Operating System Design and Implementation*, February 1999, pages 45–58.

[3] Eric Cota-Robles and James P. Held, "A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98," *Proceedings of the Third Symposium on Operating System Design and Implementation*, February 1999, pages 159–172.

[4] Helen Custer, *Inside Windows NT*, Microsoft Press, 1993.

[5] T. W. Butler, "Computer Response Time and User Performance," *Proceedings of ACM CHI'83 Conference on Human Factors in Computing Systems,* 1983, pages 58–62.

[6] Yasuhiro Endo and Margo Seltzer, "Measuring Windows NT—Possibilities and Limitations," *Proceedings of the USENIX Windows NT Workshop*, August 1997, pages 61–66.

[7] 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.

[8] Hewlett Packard Co. and International Business Machines Inc., "Application Response Measurement," `http://www.hp.com/openview/rpm/arm/index_f.html`, 1998

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

[10] Adrian King, *Inside Windows 95*, Microsoft Press, 1994.

[11] I. Scott MacKenzie and Colin Ware, "Lag as a Determinant of Human Performance In Interactive Systems," *Proceedings of ACM CHI'93 Conference on Human Factors in Computing Systems*, 1993, pages 488–493.

[12] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, 1996.

[13] Paddy O'Donnell and Stephen W. Draper, "How Machine Delays Change User Strategies," *ACM SIGCHI Bulletin 1996 v.28 n.2*, pages 39–42.

[14] Jeffrey Richter, *Advanced Windows: The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95*, Microsoft Press, 1995.

[15] Margo Seltzer, Christopher Small, "Self-Monitoring and Self-Adapting Systems", *Proceedings of the 1997 Workshop on Hot Topics in Operating Systems (HotOS-VI)*, May 1997, pages 124–129

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

[17] Sleepycat Software, "The Berkeley DB Reference Manual," `http://www.sleepycat.com`, 1998.

[18] Steven L. Teal and Alexander I. Rudnicky, "A Performance Model of System Delay and User Strategy Selection," *Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems*, 1992, pages 295–305.

[19] Transaction Processing Performance Council, "TPC Benchmark B Standard Specification," Waterside Associates, Fremont, CA, August 1990.

[20] Carl A. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*, Ph.D. thesis, Massachusetts Institute of Technology, September 1995.

[21] Ziff-Davis Inc. "Winstone 99," `http://www.zdnet.com/zdbop/winstone/winstone.html`, 1998.