



Improving Interactive System Performance Using TIPME

Citation

Endo, Yasuhiro. 2000. Improving Interactive System Performance using TIPME. Harvard Computer Science Group Technical Report TR-12-99.

Link

<http://nrs.harvard.edu/urn-3:HUL.InstRepos:25235128>

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

<https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFIOWI2M2EwYTg>

Accessibility

<https://accessibility.huit.harvard.edu/digital-accessibility-policy>

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. [Submit a story](#)

Improving Interactive System Performance using TIPME

A thesis presented

by

Yasuhiro Endo

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts

January, 2000

Copyright © 2000 by Yasuhiro Endo

All rights reserved.

Abstract

This thesis presents a new measurement methodology especially designed to improve the performance of interactive systems as perceived by the user (*user-perceived performance*). Current performance measurement and tuning techniques suffer from a multitude of problems when applied to interactive systems. Our reliance on these techniques for interactive system performance tuning has caused the systems to be tuned in a suboptimal manner with systems often failing to provide predictable performance.

Current performance measurement techniques concentrate on improving throughput rather than latency. These techniques also tend to measure system behavior under tightly-controlled situations. This approach makes it highly unlikely to discover infrequent performance problems that occur as a result of unexpected interactions among several agents in the system, such as the operating system kernel, application programs, and various servers and daemons.

We have devised a methodology that addresses the weaknesses of current measurement techniques. Our methodology is designed to determine the causes of performance problems that occur in interactive systems under normal use and plague users. We accomplish this goal using continuous monitoring and postmortem analysis. Once the exact cause of a performance problem is determined, we verify our analysis by constructing microbenchmarks that recreate the load condition that causes the system to exhibit the problem. We then use these microbenchmarks to evaluate possible remedies in a reproducible manner.

Using the methodology devised, we have identified the exact causes of several problems with BSD/OS operating system kernel and common algorithms that would otherwise had gone undetected. In one instance, processing of a pagefault taken by the X server was delayed for several seconds because the system allowed the pageout daemon to monopolize a pre-allocated structure needed to initiate a page-in request. The CSCAN disk head scheduling algorithm further contributed to this problem by not favoring latency-critical, page-in requests over less timing-critical, pageout requests. In another instance, an oversight in the process priority calculation algorithm caused newly-created, compute-bound processes to starve interactive processes.

Acknowledgments

Seven years is a long time to spend in a school, and at times, being a graduate student had been disillusioning and frustrating. Strangely enough, I do not recall ever regretting the choice that I made seven years ago. I was always sure this was something that I just had to do, and I was right. I have learned much and had lots of fun in the process, and for that, I have many to thank.

I owe a debt of gratitude to my advisor, Margo Seltzer, for being a great role model and teaching me how to do research. Without her continuous guidance and encouragement, it would not have been possible to complete this thesis.

My committee, Margo Seltzer, J. Bradley Chen, H.T. Kung, and Mike Smith, provided me with many comments and suggestions that made this thesis stronger.

Nikolas Gloy made my graduate school life fun and interesting. Being a graduate student would not have been as much fun without him. It was great to have had Chris Small and Keith Smith as office mates. Both had great knowledge from which I can draw, and Keith had a great library from which I could always find useful papers. Peg Schafer made sure that our machines kept working and gave us opportunities to take breaks from our work with weekly ice cream parties. Many thanks to Liz Pennell for proofreading this thesis and papers I have written.

This thesis is dedicated to my wife, Pui, and to my parents.

1	Introduction.....	1
1.1	User-perceived performance.....	2
1.2	Identifying problematic situations	3
1.3	The source of perceptible variability in response time	4
1.4	Identifying the source of perceptible variability	10
1.4.1	Information collected	10
1.4.2	Limitations	13
1.5	Thesis contributions	15
1.6	Dissertation outline	19
2	Background.....	21
2.1	Human-Computer Interaction	21
2.2	Common measurement techniques	25
2.2.1	Microbenchmarks	25
2.2.2	Macrobenchmarks.....	26
2.3	User-perceived performance and latency measurement	28
2.4	Continuous monitoring and postmortem analysis	30
2.5	Conclusions.....	31
3	Problems with conventional measurement techniques	33
3.1	Problems of applying throughput measures to interactive systems	35
3.1.1	Information lost.....	35
3.1.2	Changes in application behavior.....	36
3.2	Misleading results	37
3.2.1	The lost-time technique	39
3.2.2	Idle system profiles	40
3.2.3	Window open.....	41
3.2.4	PowerPoint pagedown	43
3.3	Importance of expectation	47
3.4	Reproducing intermittent problems	48
3.5	Conclusions.....	50
4	The Interactive Performance Monitoring Environment	51
4.1	Determining the problem interval.....	53
4.1.1	Keyboard and mouse interrupts	54
4.1.2	X server state and message exchange	54
4.2	Determining the source of the problem	55
4.2.1	Process status	56
4.2.2	Context switch/Sleep/Wakeup information	57
4.2.3	Highly contested locks and queues	57
4.2.4	Exec information.....	58
4.3	Implementation details.....	58
4.3.1	Kernel components	59

4.3.2	X Server modifications	60
4.3.3	User-level helper	61
4.4	The overhead of TIPME	61
4.5	Sample output	63
4.6	Limitations	67
4.7	Conclusions.....	67
5	Measurement results	69
5.1	General strategy	69
5.2	Experimental setup	72
5.3	Performance problems	73
5.3.1	Multi-second console pause.....	73
5.3.2	Sluggish mouse pointer movement.....	78
5.4	Discussion	88
5.5	Conclusions.....	90
6	The Generality of TIPME	91
6.1	Requirements	91
6.2	Collecting necessary information	92
6.2.1	Determining the problematic intervals	93
6.2.2	Aiding diagnosis	95
6.3	Existence of similar problems in other systems.....	97
6.3.1	Reliable response under heavy memory load	98
6.3.2	Priority calculation problem during fork	99
6.4	Conclusions.....	101
7	Conclusions.....	102
7.1	Lessons learned	105
7.2	Future work	106
7.3	Summary	106

Chapter 1 Introduction

We measure computer systems performance for two major purposes. The first is to rate the overall speed of the system. The results of such experiments are often used to compare hardware and software designs and implementations. The second purpose is to tune the performance of existing systems by identifying and understanding various bottlenecks in the hardware or the software. A common underlying assumption for both applications is that the measured results correlate well with the performance characteristics desired by the environment in which the system is actually deployed. In order for the measurement results to be useful, better performance measurements must mean “better” performance as applied to the actual workload of the computer systems.

In recent years, computer systems have become increasingly interactive, usually employing Graphical User Interfaces (GUI) such as the X Windows systems and Microsoft Windows. 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 metric most commonly used, in that it is affected greatly by the subjective judgment and physical limitations of users.

Despite the difference in the way actual performance is determined, we continuously apply and tune interactive systems based solely on conventional, throughput-sensitive measurement techniques [5][29][45]. The results of these benchmarks represent a

system’s average performance, which is only one of many factors that determine user-perceived performance. These benchmarks provide little or no information about other factors that influence user-perceived performance, such as latency¹ and the variability of latency. By basing performance tuning decisions solely on the performance metric that represents only one of many aspects of user-perceived performance, system designers are sometimes misled into making design and implementation decisions that have an overall negative impact on user-perceived performance [3][11]. As a result, interactive systems are tuned suboptimally. “Real” performance problems—performance problems that actually annoy users, such as unexpectedly long response times—remain uncorrected.

The most important performance metric of interactive systems is user-perceived performance. User-perceived performance is highly subjective and difficult to quantify. This warrants a new measurement and performance tuning technique. The main contribution of this dissertation is the development of such a measurement methodology.

1.1 User-perceived performance

In interactive systems, 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 [11][15]. 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. 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 guide-

1. In this thesis, we use latency, response time, and delay interchangeably.

lines by which to assess the impact of the delay, but the relationship between response time and irritation is practically guaranteed to be nonlinear. Moreover, previous studies have argued that user expectation is a critical component of user-perceived performance [11]. 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 in the face of long latencies to minimize errors and frustration [27][35][41]. Instead, what annoys users the most is response time variability. The greatest contributor to “bad” user-perceived performance is when an event takes an unexpectedly long time to complete, without apparent reason [35]. Therefore, the key to improving user-perceived performance is to identify such situations, understand why they occur and modify systems to eliminate them.

1.2 Identifying problematic situations

It is extremely difficult to quantify user-perceived performance because users’ judgements are highly subjective. We eliminate the need to quantify user-perceived performance for our purposes by involving the user in the process of identifying situations in which system performance was unacceptable. We ask the user to inform the measurement infrastructure when s/he experiences bad performance and ask the user to provide us with

a brief description of the operation with which s/he had a performance problem, such as “the menu did not show up quickly enough” or “mouse pointer movement was sluggish.”

In addition to such a description, we must determine the exact time interval during which the user experienced the performance problem. To make such a determination, we must identify the beginning and the ending points of user-visible transactions and also identify which transaction exhibited unacceptable latency. Users initiate transactions by operating an input device such as the keyboard or the mouse. We infer when user-visible transactions begin by examining keystrokes and mouse interrupt arrivals. At the end of the transaction, the system gives feedback to the user by displaying information on the screen. We infer the endings of transactions by examining when operations that involve screen updates complete. By examining the text strings displayed on the screen during the transaction, we determine the type of the transaction.

1.3 The source of perceptible variability in response time

The goal of our measurement methodology is to identify the cause of the system exhibiting unexpectedly long response time to process a user-visible transaction; an unexpectedly long event is one that is normally processed much more rapidly. This goal differs from conventional tuning approaches in several ways. First, we can ignore problems that induce latencies too small to be perceived by users. For example, changes in the first level cache hit ratio are unlikely to cause sufficiently large perturbations to performance that a user would notice them. Second, we are focusing exclusively on operating system factors that contribute to poor user-perceived performance. Tuning individual applications (and user-level libraries, such as user-level thread packages), while certainly important, is left

for future work. Third, we are restricting ourselves to problems whose causes are on the local machine. If the problem is remote, we wish to identify it as such, but we are not yet attacking the distributed tuning problem. Within the framework imposed by these limitations, we have determined the causes for such variability to be one of or a combination of the following:

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

In this section, we describe each of these factors in detail and show that these are the only software-related sources of user-perceptible variability in response time.

The first source of the variability in response time is a change in the amount of the CPU time that the application and operating system require to complete the 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 the average response time that the operation exhibits. Our techniques are designed to capture unexpected delays. These are the cases where the response time deviates significantly from the norm.

Changes in the amount of CPU time that a transaction requires can occur because, for some operations, the amount of computation required is variable, often depending 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 such

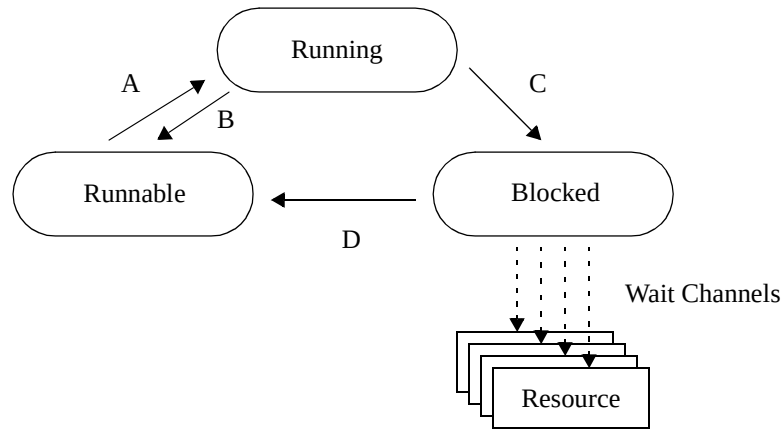


Figure 1: Process Scheduling State Diagram. Each active process in the system is assigned one of the three scheduling states, and a well-defined set of edges connect these states. The process in the running state is actively executing on the CPU. A runnable process has access to all the resources that it needs to make forward progress and is waiting only for the availability of the CPU. A blocked process is waiting for the availability of one or more resources by associating itself to one or more wait channels, which represent the resource (s).

operations and thus can change the amount of computation that they require to complete a transaction.

Other than the changes in the amount of CPU work required to complete a transaction, the only remaining causes for perceptible response time variability are the changes in the amount of time that the transaction spends not executing on the CPU. Most of the operating systems break this time into two states: *Blocked* and *Runnable*. The Blocked state corresponds to the case where the process cannot execute because it is waiting for the completion of an I/O operation or for the availability of some resource. The Runnable state corresponds to the case where a process is not running because the operating system has decided not to allocate CPU time to the particular process. These two states, in addition to the *Running* state (a process actively executing on the CPU) and the edges that correspond to all the allowed state transitions between them comprise the process scheduling model depicted in Figure 1. In this process scheduling model, all active processes in the system are assigned one of these three states. There are a well-defined set of edges con-

necting these three states. A process traverses edge A when the CPU scheduler decides to execute the process on the CPU and traverses edge B when the scheduler takes the CPU away from the process. If, while executing on the CPU, the process performs an I/O, takes a page fault, or accesses a resource that is not available, the process will traverse edge C to enter the blocked state. A blocked process traverses edge D to become runnable when an asynchronous event, such as a disk interrupt notifying the completion of an I/O operation, signals the operating system that the condition for which the process was waiting has been met. There is no edge that takes a process from the *Runnable* state to the *Blocked* state, since a process must be executing a sequence of instructions on the CPU in order to initiate a blocking operation, such as an I/O or a pagefault. Therefore, a process in the *Runnable* state can only reach the *Blocked* state by first entering the *Running* state. This model, in the exact or slightly varied form, is used by most general-purpose operating systems [10][17]. One notable exception to our process scheduling model is MS-DOS, which, in its native form, does not have support for multiprogramming (running multiple processes simultaneously).

In the discussion below, we begin by introducing a simple uniprogrammed execution model without I/O demands and explain how the process model above is derived by introducing complexity to the system. Simultaneously, we explain how it is precisely these changes that introduce the response-time variability that we seek to eliminate.

First, consider executing a transaction that is pure computation on a simple system with no multi-programming or virtual memory support. Such a transaction will spend its entire lifetime executing on the CPU, and the time required to complete the operation

depends solely on the amount of CPU time necessary for the computation. The time that the transaction spends not executing on the CPU is zero.

Next, we extend this model to allow the transaction to include I/O operations such as disk accesses and/or network operations. During these operations, the CPU will stop executing the program, awaiting completion of the I/O. A process in this condition cannot be allowed to execute on the CPU, and the operating system must differentiate this process from the one executing on the CPU. The operating system marks these waiting processes to be in the blocked state. For efficiency, each blocked process is associated with a wait channel, which associates the process with the resource for which the process is waiting. This allows the operating system to selectively wake up (i.e., mark the process no longer blocked) processes depending on the resource that became available. The amount of time the transaction spends waiting for I/O operations can change, introducing variability in the time that the transaction spends not executing on the CPU, which, in turn, affects response time.

Next, let's consider a system with virtual memory. With virtual memory, the program can now take page faults, which also cause the CPU to suspend execution of the program waiting for the completion of the page fault processing. These waits are also examples of the second class of response time variability—delay due to I/O operations. By recording when a process blocks and when it becomes runnable, we can easily determine delays due to such waits.

Now consider adding multi-programming capability to the system. In a multi-programming environment, the operating system must often select one of many eligible processes to run on the CPU. Conversely, when a process is running on the CPU, the

operating system may decide to prevent it from running to enable some other (higher priority) process to run. In this case, the processes that are eligible to run (i.e., that are not waiting on any resources or I/O) are in a fundamentally different state than processes in the blocked state. They are not on the CPU, but they are not waiting for the completion of any I/O event or the availability of any resource. The operating system must differentiate these ready-to-run processes from those in the blocked state. These ready-to-run processes are commonly referred to as being in the runnable state. The operating system's scheduler dictates which runnable process executes on the CPU and for how long. This decision is an example of the third class of response time variability—process scheduling decisions.

The addition of multi-programming introduces another source to the second class of response time variability—resource contention. With multiple processes executing on the system, it becomes necessary to serialize accesses to shared resources. The operating system blocks processes that request resources, such as buffers and devices, that are already in use. As in the I/O case, such a process will be put in the blocked state until the resource becomes available.

Having presented the standard model of process scheduling and demonstrated how and why wait time is represented in this model, we can now return to our categorization of the variability in response time. It arises by a change in the amount of computation that a transaction requires and/or by a change in the amount of time that the process(es) involved in handling the transaction spent not executing. The amount of time that the transaction spent computing is equivalent to the amount of time the transaction spent in the running state. The amount of time that the transaction spends not executing is equal to the sum of the time spent in the runnable and blocked states.

1.4 Identifying the source of perceptible variability

In the previous section, we described the possible software-related causes of response time variabilities. In this section, we discuss the strategy that we have devised to monitor these sources of variability. In Section 1.4.1, we describe the information that we collect to monitor each source. In Section 1.4.2, we describe the limitations of our strategy.

1.4.1 Information collected

The first source of response time variability is the change in the amount of CPU time that the operation requires. We monitor changes in the CPU time requirement using profiling information. By sampling the value of the program counter at a fixed interval, we can approximate the amount of CPU time consumed by the task and determine where in the system, within the application and/or the operating system, the CPU time is spent. When the system exhibits unusually long latency processing a transaction, we can ascertain the cause by comparing the profiling information collected during the problem interval to that collected when the system processed a similar transaction with acceptable latency.

The second source is the change in the amount of time that a process spends waiting for an I/O or the availability of resources, and the third is a result of process scheduling decisions. A process will be in the blocked state when it is waiting for an I/O or resource(s), and it will be in the runnable state when it is waiting for the CPU scheduler to run the process on the CPU. We ascertain the amount of time that the process(es) involved

in handling the transaction spends in each of these states by capturing when each process traverses the edges connecting the three process-scheduling states shown in Figure 1.

For every blocked process, we record the wait channel with which it is associated. When we find that the cause of a long response time is a process spending an unusually long time in the blocked state, we use this information to identify the resource, which can be a disk queue, a buffer, an IPC socket, etc. This information also allows us to ascertain dependencies between processes when handling a transaction involves multiple processes, as is the case with X Windows user-interface transactions, which uses IPC for client-server communication.

A process spending an unusually long time waiting on a wait channel is a sign that the resource associated with the channel is highly contested, and we must understand how such contention arose and eliminate it. For this task, it is often necessary to understand how the resource is consumed. To this end, once such a resource is identified, we use additional instrumentation points to examine allocation and deallocation of the resource in question. Identifying all possible resources for which such contention can occur a priori is challenging, so we have adopted a strategy of adding instrumentation points as new contested resources are identified. In a commercial system, we would envision providing the ability to capture all resource information. Under initial deployment, this detailed monitoring would be disabled. Whenever a resource contention issue is identified, the monitoring associated with that resource would be enabled.

By recording when each process traverses edges entering and leaving the runnable state, we can determine the scheduling decisions that the process scheduler made. This information alone is sufficient to determine whether the source of the performance prob-

lem is the scheduler. Alone, however, it is not sufficient to indicate to kernel developers how to construct a fix. Instead, this information sufficiently narrows down the source of the problem (i.e., narrowing from an OS problem down to a scheduler problem), that the developers are better able to diagnose and correct it. At this point, the developers may take advantage of TIPME and scheduler-specific knowledge to collect additional, more detailed data that identifies the exact source of the problem. In our target platform, the scheduler bases its decision on process priorities; identifying and recording this class of information enables us to find the exact problem in the algorithm. With a different scheduling algorithm, different information might be collected. For example, under a lottery scheduling system, one would record the current ticket allocations and the result of the lottery for each scheduling decision.

The information we collect allows us to detect changes in the amount of work required to process a transaction. This information also allows us to identify the reason that the process is kept from performing this work, which may be one or a combination of the following causes: scheduling decisions, device wait, and/or resource contention. If the source of the variability is a change in the amount of work that the transaction must perform, we use profiling information to determine where the extra time is being spent and either propose or make algorithmic changes. If the source is a CPU scheduling decision, we determine why the scheduler made such a decision and correct the problem. Finally, if the source is the resource wait, we study how this highly-contested resource is being used and reduce contention.

1.4.2 Limitations

In this section, we discuss the limitations of our measurement strategy in each of the following categories:

1. Measurement granularity.
2. Understanding the source of delays.
3. Diagnosing device problems.

Some limitations are introduced as a result of the focus of the study—to understand user-perceivable delay. Some are introduced for simplicity, and some are an inherent limitation of our software-based approach. We describe the limitations in detail below.

First, 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 the data we collect are sometimes too coarse to diagnose the causes of sub-millisecond delays. This limitation does not pose a serious problem, for humans cannot perceive sub-millisecond delays.

Second, there is also a limit on how much understanding we can gain about the source of the delay. We treat some of the sources of delay as black boxes—some to make the problem simpler and others out of necessity. We assume that application programs do not do their own thread scheduling. While this assumption holds mostly true under UNIX operating systems where multi-programming capability was built into the system from day one, it does not always hold true in Microsoft Windows software, a system initially developed with no multi-programming capability [22][28]. We expect such programs to slowly disappear as newer versions of Microsoft Windows support kernel-level threads

and various coding guidelines encourage use of such facilities [10][30]. We have also decided to 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 combination of network transfer latency and the latency with which the server provided a response, which can be further broken down into CPU time and wait time, in the same way we have broken down the response time on the client machine. A setup that enables such diagnosis is a straight-forward extension of our methodology. Such a setup requires us to deploy the measurement infrastructure on both client and server and coordinate their activity, such as associating requests arriving at the server with a transaction that user initiated on the client. While implementing such infrastructure is interesting and useful, the majority of interactive systems are used mostly as stand-alone machines in people's homes where the only connection to the outside is via a modem line. We have left exploring distributed systems and coordinating distributed data collection as future work.

Finally, we are also unable to determine the cause when a device, such as a disk, misbehaves. Modern disk drives often have intelligence built in, making it possible for the disk to behave in unexpected ways, such as spending several seconds completing an I/O request for no apparent reason. Our infrastructure will identify when such a device is the cause of long latency, but cannot determine why the device behaved in such a manner. These problems are well outside the operating system's control.

1.5 Thesis contributions

This thesis introduces a performance measurement and tuning methodology designed to enable the analysis and elimination of performance problems that conventional measurement techniques are ill-equipped to solve, namely, infrequent, not-easily-reproducible performance problems. Our methodology concentrates on improving user-perceived performance by removing the occurrence of sporadic, long latencies that users find unacceptable.

This thesis identifies and demonstrates the problems that arise from applying throughput-sensitive performance measurement techniques to interactive systems. Conventional benchmarks, whether designed to measure throughput or latency, concentrate on obtaining stable, repeatable results excluding data points that deviate wildly from the norm [4][7][21], when in fact, these data points are exactly the ones that we need to understand in order to improve user-perceived performance. To measure interactive system throughput, benchmark programs commonly saturate the system's input request stream by injecting requests into the system as quickly as the system can accept them. This technique ignores the fact that normal interactive users are incapable of producing such an input stream because of the physical limitations of users and input devices. A major problem with this technique is that we cannot guarantee that a given benchmark performs the same amount of computation across different hardware. Many sophisticated interactive applications change the amount of work they perform depending on the input rate and the machine speed [7][11][22]. We will also show that the speed of features, such as animation, that are programmed to execute at a predetermined rate have a less direct relationship

with system performance and can yield difficult to understand and often misleading results [11].

Our measurement methodology is centered around The Interactive Performance Monitoring Environment (TIPME), a new measurement infrastructure especially geared toward improving interactive system performance. TIPME is a measurement system that enables the diagnosis of events that irritate users. Using TIPME, an operating system developer can isolate and eliminate the causes of these events. 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. By understanding cases in which the user indicated that the system exhibited bad performance and eliminating their causes, we can improve user-perceived performance.

We will describe, apply, and demonstrate the measurement methodology we have developed. This approach uses TIPME to identify performance problems that actually annoyed users and to pinpoint the exact causes of the problems. Once we identify the exact cause of a problem, we construct a microbenchmark that reproduces the identical problem. We then use the microbenchmark to evaluate possible solutions to the problem. Once a solution is achieved, we install the modified system and continuously monitor its performance still using TIPME. As new problems are encountered, the cycle repeats. In this thesis, we present two iterations of this methodology to correct problems that actually irritated users.

The first problem caused the console to pause for several seconds when a heavy memory load was present in the system. TIPME output showed that the X server was blocked for several seconds waiting for a page fault to be handled. There were two causes

of this particular problem. The first was a minor oversight in the way a critical structure was allocated. The second problem was the disk request sorting algorithm, which allowed a relatively short, latency-critical disk request to queue behind many large, non latency-critical disk requests. This disk request sorting algorithm is a prime example of how a system tuned for throughput can adversely affect interactive performance.

The second problem caused the mouse pointer movement to lag for several hundred milliseconds when a large build job was executing in the system. From TIPME output, we determined that the operating system was allocating the CPU to compute-bound compile jobs instead of the interactive X server process. This prevented the X server from handling the mouse requests. This problem uncovered two problems with the way process scheduling is currently handled by the operating system. The first problem is in the assumptions that the scheduler makes about newly created processes. The scheduler assumes that newly created processes are I/O bound (i.e., they tend to give up the CPU voluntarily) unless the process proves otherwise. It often takes several hundred milliseconds for the scheduler to adjust to the true behavior of newly created processes. Meanwhile, these newly created processes are allowed to starve other processes in the system that have been assigned priority levels appropriate for their CPU consumption characteristics. The second problem is the way the scheduler charges CPU usage to processes. Although the X server performs most of its work on behalf of other processes, the X server process is charged for the work and has its priority lowered.

The fact we have found these problems in a mature operating system shows that the latency aspect of operating system performance has long been neglected. During development, various performance problems are introduced into the system. Testers and

developers remove these performance problems using benchmarks prior to releasing the product. The benchmarks most often used are sensitive to changes in system throughput rather than latency, and even in cases where the benchmarks measure latency, the program is geared toward measuring average latency, often throwing away data points that vary wildly from the norm [4][5][21][29][45]. These benchmarks help the developers remove performance bugs that affect system throughput but do little to help the developers identify and remove latency-related performance problems, particularly those that affect latency variability. A measurement technique such as ours introduces a systematic way to identify and remove performance problems that affect latency. Using such a technique in conjunction with throughput-sensitive benchmarks, we can tune operating systems for latency as well as for throughput.

Although we have limited the application of our measurement methodology to the improvement of user-perceived performance, the methodology has further implications and applications to systems for which response time, not just the throughput, is important. Such systems include real-time systems and systems with soft real-time requirements, such as those that perform software-based signal processing. Techniques similar to ours can be used to identify the reasons why a system performs outside its designed constraints. Another class of systems for which latency is important is web-related servers and transaction processing systems. Web servers must be able to handle many requests per unit of time (i.e., must have good throughput) but must also deliver web content to clients with acceptable latency. For transaction processing systems, the latency requirement is explicit. Transaction benchmark suites such as TPC-B measure throughput but also require the system to process each transaction within a specified response time. Again, a methodology

similar to ours can be used to identify the factors that contribute to long latency and make appropriate optimizations. Such techniques should be used in conjunction with conventional throughput-tuning techniques to improve both the latency and the throughput aspect of system performance.

1.6 Dissertation outline

In Chapter 2, we discuss related work in performance and measurement methodology and human-computer interaction.

Chapter 3 discusses and demonstrates the problems with the way we currently measure and tune interactive systems. Throughput-based measurement techniques, which are widely used to rate or tune interactive system performance, introduce myriad problems when applied to interactive systems. We explain these problems in detail and demonstrate how these problems affect the measurement results.

Chapter 4 presents The Interactive Performance Environment (TIPME), which has been specifically designed to help identify, understand, and eliminate unexpectedly long response times that negatively affect user-perceived performance. TIPME accomplishes this task by continuously monitoring all the possible software sources of response time variability.

Chapter 5 demonstrates the effectiveness of TIPME. We describe how we used TIPME to identify and correct performance problems that 1) actually irritated interactive users and 2) had remained undetected by conventional measurement techniques.

Chapter 6 discusses the applicability of TIPME to systems other than the target system. In particular, we use the microbenchmarks to determine if the performance prob-

lems that we observed in the target system are unique to the target system or if they are commonly experienced by other systems. We also discuss how we can make the analysis easier by instrumenting X clients. X clients do not need to be instrumented in order to work in TIPME, however, the postmortem analysis can be made much simpler if clients are modified to provide a bit of extra information. The feasibility of such instrumentation is discussed as well.

Chapter 7 presents conclusions and lessons learned and discusses how we should approach performance measurement and tuning of interactive systems, which are becoming increasingly dominant.

Chapter 2 Background

Our work draws on research from the conventional measurement community as well as research from the human-computer interaction (HCI) community. Improving interactive system performance requires knowledge from both domains. Based on this knowledge, we devise a new measurement and performance tuning technique that addresses the weaknesses of current measurement techniques. Our technique improves user-perceived performance by eliminating latencies that annoy users. To identify the cause of such long latencies, we use continuous monitoring and postmortem analysis.

In Section 2.1, we discuss relevant HCI research results, and show how these results form a basis of our measurement strategy. In Section 2.2, we describe commonly used measurement techniques and their shortcomings. Section 2.3 reviews recent work that concentrated on computer systems' latency, rather than throughput. In Section 2.4, we contrast TIPME with other systems that use continuous monitoring and postmortem analysis. Section 2.5 concludes.

2.1 Human-Computer Interaction

HCI researchers have shown that a user's perception of performance has a direct relationship with response time. They have also shown that decreasing response time below a certain threshold does not result in improved user-perceived performance and that users' judgments about performance are affected greatly by their expectation formed from past experiences. These findings form the basis of our performance measurement and tuning strategy. Our technique specifically aims to measure and improve latency, for users are

sensitive to response time. We argue that throughput-based performance metrics are inappropriate for measuring interactive system performance since throughput measures treat all latency equally regardless of its duration. We have chosen to have users notify the measurement system when they experience a performance problem, because it is nearly impossible to adequately quantify the threshold between good and bad response time. The definition of acceptable response time depends highly on the types of operations and is affected greatly by user expectation.

Rushinek and Rushinek report the importance of providing good response time in keeping computer users happy [32]. Their findings are based on the result of a controlled survey that asks the question, “What makes users happy?” The authors evaluate the impact of 17 factors such as response time, number of systems and users in an installation, how well the system meets users’ expectations, the type of the system (mainframe, mini-computer, or micro-computer), and cost, on overall user satisfaction. They found that of all the factors evaluated, providing good response time was the most important factor in making users happy.

Guynes studied the relationship between response time and state anxiety [13]. This study classified 86 test subjects as having either Type A or Type B personality. Type A personality is described to be “composed primarily of competitiveness, excessive drive, and an enhanced sense of time urgency,” and Type B personality is defined as the relative absence of the Type A characteristics.

These users were given a text editing task with varying response time patterns—short, long, and variable. Increases in the users’ anxiety levels after the task were evaluated using a State-Trait Anxiety Inventory [37]. This study found that there is a positive

relationship between state anxiety and system response time regardless of the personality type. Users of both personality types experienced an increase in state anxiety with both long and variable response time patterns.

Shneiderman discusses many aspects of designing a user interface from interface organization to performance [35]. In the chapter dedicated to performance, Shneiderman states that “acceptable” response time is highly dependent on the type of task influenced by 1) users’ past experiences, 2) the type of task and each user’s tolerance level, and 3) how well a user adapts to long and variable delays. He also discusses the impact of variability on response time. Although modest variability (within 50 percent of the mean) is often tolerated by users, a large variability of at least twice the anticipated response time is shown to cause user frustration.

Shneiderman also presents 2 seconds as an appropriate response time for tasks such as hearing a dial tone on a telephone or seeing a picture on a television set, while users expect response time of 0.1 seconds or less for tasks such as typing [35]. Table 1 summarizes the response time guideline proposed by Shneiderman.

Table 1: Response time guidelines proposed by Shneiderman [35]

Type of the task	Response Time Guideline
Typing, cursor motion, and mouse selection	50 to 150 milliseconds
Simple frequent tasks	Less than 1 second
Common tasks	2 to 4 seconds
Complex tasks	8 to 12 seconds

In a study that evaluated the impact of mouse movement response time on human performance, MacKenzie and Ware show a more precise relationship between latency and human performance [18]. They examine how a delay between input action and output response affects the task of moving a mouse pointer to the target area. Although the target

of their study is to extend Fitts' Law [35] to account for the delay in response², their measurement results provide us with insight about the limits of human perception. They measure the time required to complete mouse pointing tasks under six different difficulty conditions (determined by the target size and movement required) and four different delay conditions. They find that at delays of 8.3 ms. and 25 ms., test subjects completed the task with nearly identical speed and accuracy. Under test cases with 75 ms. delay, the results show a measurable drop in the speed and the accuracy. At a delay of 225 ms. the speed and accuracy suffered drastically. Their findings show that human performance is dependent on latency and suggest that the human perceptual limitation for mouse movement lies somewhere between 25 ms. and 75 ms.

There has been other research in the HCI community evaluating the impact of latency on user performance [6][27][41]. The fact that short response time is an important determinant of user satisfaction and performance is well documented [6][13][32][35]. It is also suggested that user satisfaction and performance are dependent on expectations formed by past experiences and how well an individual user adapts to slow response time [35]. What is lacking is research that adequately quantifies latency thresholds between good and bad performance for each of the many types of operations we perform under a GUI environment.

2. Fitts' Law describes the relationship between time required for pointing and the difficulty of the pointing task, which is defined by the target size and the distance to the target.

2.2 Common measurement techniques

We constantly struggle to create synthetic benchmarks that yield results relevant to systems under normal use. Such benchmarks are used to predict the performance of a system under actual use, rate system performance, and to motivate and evaluate performance tuning efforts. Creating such a benchmark is extremely difficult, and constructing a perfect benchmark is impossible. Some of the biggest challenges in creating such a benchmark are selecting an appropriate workload [14] and defining a performance metric that accurately quantifies how well the system is achieving the desired system performance.

2.2.1 Microbenchmarks

Microbenchmarks measure the cost of simple operations that are believed to be important determining factors of end-to-end performance. By improving the performance of such operations, one hopes to improve the end-to-end performance of the system. However, these operations are not useful in and of themselves and have an indirect, rather than a direct, relationship to the actual tasks introduced into the system. Their results are also sensitive to the behavior of the various caches in the system, often yielding unrealistic results. For these reasons, microbenchmarks often fail to be a good predictor of end-to-end system performance [3].

One of the strengths of microbenchmarks is that their results are easy to understand and analyze. For this reason, microbenchmarks are best used to probe and understand specific aspects of the system or to support and explain the result of a larger-scale benchmark. McVoy and Staelin developed a portable microbenchmark suite, Lmbench, that measures the speed of various operating system and hardware operations [21]. Brown

and Seltzer modified Lmbench to create hBench-OS, and applied this benchmark suite to evaluate how operating system performance scaled with evolutions in the Intel x86 architecture [4]. Chen et al. used microbenchmarks to provide insights into how quickly three different operating systems—NetBSD, Windows NT, and Windows for Workgroups—handled several operating system operations [7]. They also used microbenchmarks results to justify and explain the results of large-scale benchmarks with varying success.

In summary, microbenchmarks measure, in isolation, the performance of small operations that are determinants of end-to-end performance. Microbenchmarks are an effective tool for understanding the performance of a specific operation, but they are poor predictors of end-to-end performance because there is no clear indication to what extent each of the measured operations affects end-to-end performance.

2.2.2 Macrobenchmarks

Unlike microbenchmarks, macrobenchmarks measure how quickly a system completes a non-trivial computation often using a real application program. The resulting figure is the throughput of the system executing the target program(s). Benchmark suites such as SPEC CPU95 [29], Sysmark98 [5], and Winstone99 [45] use the measured performance of non-trivial programs to calculate a small set of numbers that represents the performance of the system.

The SPEC CPU95 suite includes two sets of applications: one that measures the system's integer performance and the other that measures floating-point performance. The integer benchmark suite called SPEC CINT95 consists of eight applications that represent typical integer-intensive tasks such as games, compilations, and database operations per-

formed by computer systems. Similarly, the floating point benchmark suite, called SPEC CFP95, consists of ten applications that represent typical floating point tasks such as simulations and various types of scientific computation. Each suite yields one number that represents the system's performance from each run.

Both Sysmark98 and Winstone99 are specifically designed to rate the performance of Microsoft Windows and/or Windows NT systems. One of their most notable characteristics is their use of commonly used commercial applications, such as Microsoft Word, Excel, Powerpoint [22], and Netscape Navigator [25].

One of the drawbacks of using conventional measurement tools for evaluating user-perceived performance is that measurement results are often gathered under unrealistic conditions. For example, most of the current benchmark programs yield little information about an individual run. In many cases, the result of a benchmark is a single number, such as the number of SpecMarks[29]. The lack of supporting information makes it difficult to explain and understand the results when they are highly variable, so these tests are often run in single-user mode with standard system daemons disabled. These precautions, which are essential for producing reproducible, statistically significant results, make throughput benchmarks unsuitable for discovering and studying the unexpected system behaviors that interactive users often experience.

Most of the existing measurement (and performance-tuning) techniques rely on controlled, throughput-based benchmarks. These benchmarks are useful for quantifying the performance of systems where throughput is important, such as scientific computation or compilation environments. Unfortunately, these throughput measures are not useful in

quantifying or helping to improve the most important performance metric in interactive systems—user-perceived performance.

2.3 User-perceived performance and latency measurement

Throughput-based benchmarks are effective aids in improving throughput. Unfortunately, these throughput measures are not useful in quantifying or improving user-perceived performance. As discussed in Section 2.1, several works in Human Computer Interaction have identified response time, the latency with which the system handles user-initiated requests, as one of the most critical factors that determine user satisfaction along with the users’ expectation of system performance [13][32][35]. We discuss attempts to measure and evaluate system performance based on response time below.

There have been efforts to measure response time on modern, interactive systems. Application Response Measurement (ARM) measures response time directly by providing API functions that client programs call before and after an operation [15]. Currently, most of the applications that take advantage of ARM’s API are large, in-house, business applications, but more commodity applications are expected to take advantage of the API in the near future. In contrast to the ARM methodology, Endo et al. inferred response time from CPU activity and message exchanges between MS-Windows clients and the server [11].

Cota-Robles and Held agree with Endo et al. that throughput alone does not adequately represent the ability of computer systems to promptly handle interactive user requests and other real-time services such as video and audio playback [8]. The authors concentrate on characterizing the Windows NT and Windows 98 operating systems’ ability to handle these real-time workloads by measuring how quickly and reliably the sys-

tems deliver hardware interrupts to corresponding handlers in a loaded system. Both Windows NT and Windows 98 share a driver model known as the Windows Driver Model [24]. In this model, the bulk of computation traditionally handled by an interrupt handler is done by kernel threads or via Deferred Procedure Calls. Either method requires the operating system to make scheduling decisions and introduces a possibility for the operating system to make poor decisions that result in long interrupt delivery latencies. 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 [45] showed that both systems had throughput scores within 10 to 20 percent of each other.

Schmidt et al. use response time to evaluate the performance of the SLIM thin-client architecture [33]. They first measure various components of response time, such as network delay and the latency with which the SLIM clients process different types of protocol packets. They then construct, based on actual measurement, a model that describes how a user would introduce a load to the system. This workload model is used to simulate the stress of supporting multiple users on a server. While they are successful in measuring latencies under various load conditions, they do not present any information on response-time variability, often presenting one latency figure for each test case. Additionally, they fail to take into account user subjectivity to its full extent, treating all latencies equally regardless of the type of event for which the latency was observed. Our approach is different in that we do not try to quantify user-perceived performance—it is too subjective. We are, however, extremely sensitive to users’ subjective judgements, which are partly deter-

mined by response time and response-time variability. Our goal is not to rate system performance but to identify and eliminate latencies that users (subjectively) find unacceptable.

Compared to old screen-based and command-line based interfaces, defining and measuring response time in systems that use a GUI is difficult. Until recently, there has been little effort made to measure response time under GUI environment.

2.4 Continuous monitoring and postmortem analysis

This thesis introduces The Interactive Performance Monitoring Environment (TIPME) which makes use of continuous monitoring of system states and postmortem analysis. This section describes several previous applications of these techniques.

Gprof is an execution profiler that provides hierarchical profiling information [12]. During execution, gprof-instrumented programs construct a function call graph and count how many times each edge is traversed. Gprof estimates how much time is spent inside the various functions by sampling the program counter value during operating system clock interrupts. When program execution is complete, this information is combined to calculate how much time is spent inside each of the functions or within a subtree of the call graph.

McKusick used gprof to tune the performance of the BSD 4.2 operating system [19]. Since operating systems do not terminate, he used an external program kgmon to control various aspects of the measurement process such as turning on and off the sample collection and retrieval of collected data.

Gprof and TIPME are similar in that they both continuously measure systems under normal use, but similarities end here. Besides the detail of information collected, the

most important difference between TIPME and gprof is how the measured time interval is selected. Gprof results reflect all the activity that occurs during the target program's lifetime or the time during which profiling was enabled (if the target program happens to be an operating system kernel). In other words, the results represent the average of the measured interval. TIPME, on the other hand, is designed to allow selection and evaluation of a specific subinterval enclosed within the collection interval. Cota-Robles and Held also used a technique similar to ours to identify where much of the time was spent in the time interval during which the system had difficulty delivering hardware interrupts to an appropriate interrupt handling routine [8].

DCPI is a continuous monitoring technique that attempts to measure the performance of the CPU executing under normal conditions by continuously profiling a variety of hardware statistics [1]. Though similar to TIPME in the sense that both systems use continuous monitoring, these systems are worlds apart in the abstractions with which they concern themselves. DCPI captures information about hardware events such as cache misses and branch mispredicts, 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 how well the hardware is performing in systems under normal use.

2.5 Conclusions

HCI research, as well as our experience, suggests response time and expectation are the critical factors in determining user-perceived performance. The HCI community

has made efforts to determine and understand the effect of changes in systems' response time characteristics on users' ability to perform work. Unfortunately, the work to quantitatively associate tolerable response time and type of operation is seriously lacking, making automatic identification of "unacceptable" response time infeasible.

Current measurement techniques are centered around throughput. These techniques are effective in improving system throughput, but throughput is not the major deciding factor of interactive system performance. Despite these observations, we currently benchmark and optimize interactive systems using system throughput.

Chapter 3 **Problems with conventional measurement techniques**

Most of the current performance tuning techniques rely on controlled, throughput-based benchmarks. These techniques are effective for improving performance of programs for which throughput is important, such as programs for scientific computation. Unfortunately, throughput measures are often unsuitable in rating interactive system performance where better throughput does not necessarily equate to good user-perceived performance [11][35].

The majority of current GUI benchmarks [5][45] determine system performance by measuring how quickly the system handles a sequence of user requests, such as typing, screen update, and displaying of menus. Instead of collecting the latencies of individual user requests, these benchmarks estimate the system performance by how quickly the system processes a series of requests. This technique is problematic for two reasons. First, it treats all latencies equally, ignoring the fact that some latencies are imperceptible to users. Second, it ignores some long latency events that can harm user-perceived performance more than their numeric values suggest.

Today's GUI benchmarks also mask infrequent performance problems, because these rare problems tend not to have a large impact on throughput. This would not be problematic if throughput, as defined and measured by the current GUI benchmarks, correlated well with user-perceived performance. Unfortunately, it does not. The assumption that throughput is the one and only relevant performance metric does not hold up well in

interactive systems, where every single event-handling latency affects user-perceived performance.

Throughput-based benchmarks often produce one number that represents system performance. This simple output usually cannot explain the cause when the same benchmark program produces a different result due to interference from other, unmeasured and unexpected activities in the system. Asynchronous events such as interrupts generated by network packet arrival and periodic executions of daemons can unexpectedly affect the result of the benchmark. In order to keep these unmeasured system activities from influencing the measured result, we often minimize non-deterministic system activities by testing the system in unrealistic conditions—for example, turning off daemons and removing the machine from the network. These precautions we are forced to take make throughput benchmarks unsuitable in discovering, studying, and correcting unexpected system behavior that often frustrates interactive users.

In this chapter, we explain and demonstrate the way conventional benchmarks fail to be accurate predictors of performance and fail to be effective tools for guiding interactive performance tuning. We discuss the problems of the current techniques in Section 3.1. Section 3.2 introduces the lost-time technique, which measures CPU activity by calculating the time taken away from the operating system idle loop. We use this technique to demonstrate cases in which the current techniques yield misleading results. Section 3.3 discusses the importance of user expectation as a determining factor of user-perceived performance. Current measurement techniques' inability to measure and help us understand intermittent problems is discussed in Section 3.4. Section 3.5 presents conclusions of this chapter.

3.1 Problems of applying throughput measures to interactive systems

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

3.1.1 Information lost

The results of throughput benchmarks are often reduced to a single number that indicates how long a system took to complete a sequence of events. Although this can provide information about the sum of the latencies for a sequence of events, it does not provide information about the latencies themselves and their variance, which is an important factor in determining perceived performance [13][32].

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

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

3.1.2 Changes in application behavior

Many popular applications change their behavior in response to the rate of user input and system speed. When an application receives user input that overrides the operation currently being processed, the application may interrupt the processing, saving unnec-

essary work. For example, in response to a sequence of keystrokes, a word processor program may decide to display multiple characters in one screen update instead of updating the screen after every keystroke. This batching behavior is desirable, for it can save unnecessary computation and can allow slow systems keep up with user input. Unfortunately this behavior makes it difficult to benchmark interactive systems accurately.

Throughput benchmarks rely on the assumption that a given task or sequence of tasks performs a fixed amount of computation. The batching behavior breaks this assumption. Moreover, throughput benchmarks often drive the system by feeding user input as quickly as the system can accept it. This technique completely ignores the fact that a human user cannot produce input events more quickly than some upper bound. Such an input stream is unrealistic and susceptible to generating misleading results.

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

3.2 Misleading results

In this section, we present several measurement results that demonstrate how throughput sensitive benchmarks can yield misleading results. We first explain the lost-time technique, which we use to measure how much work is performed by the CPU during

an interactive operation. We demonstrate this technique by presenting the measured activity in a system when the system is sitting idle, handling neither user-initiated events nor network-related events. Finally, we present example cases in which throughput sensitive benchmarks yield misleading results. More specifically, we describe instances in which:

- the system inserts artificial delays for animations making end-to-end measurement a less direct indicator of system performance, and
- an application changes its behavior depending on the input rate.

We conducted this study on Microsoft Windows NT version 4.0 running on a personal computer based on a 100MHz Pentium CPU. Details of the hardware and the software used are shown in Table 2. We take advantage of the Pentium performance counters, which provide a CPU cycle counter for precise time measurement along with the ability to count hardware events such as data accesses, cache misses, and interrupts [16].

Table 2: Test setup

CPU	Pentium 100
Motherboard	Asus T2P5
L2 Cache	512KB pipeline burst cache
RAM	96MB, 60 ns.
Harddisk	Fujitsu M1606S, 5400 rpm, 10 ms avg. seek time
SCSI Host Adapter	NCR825
Display Adapter	Diamond Stealth 64 DRAM
Operating System	Windows NT Version 4.00.1381
Applications	Microsoft PowerPoint for Windows 95 Version 7.0
	Microsoft Notepad Version 4.0 Build 1381
	Simple application

3.2.1 The lost-time technique

For the results we present in this section, we base our comparison on how much work the CPU had to perform for a given task. We infer the amount of work performed from the total time during which the CPU was occupied.

We determine the amount of work performed by calculating how much time was spent outside of the operating system idle loop. We would ordinarily determine this by instrumenting the operating system idle loop, but this is impossible to do directly under Windows NT, for which we do not have access to the source code.

Instead, we replace the system's idle loop with our own low-priority process³. Unlike ordinary idle loop implementations, which either loop infinitely or halt the CPU waiting for an interrupt arrival, this low-priority process repeatedly performs a fixed computation (N iterations of a tight loop) and then records a timestamp. Whenever there is a job to perform in the system, the CPU is taken away from the loop, increasing the elapsed time to complete the computation. So any non-idle time will manifest itself as an elongated time interval between two time stamps. The pseudo code of the low-priority process is as follows:

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

We select the value of N such that the inner loop takes one ms to complete when the processor is idle. In this way, we generate a trace record *per millisecond of idle time*. If the processor is taken away from the idle loop, the loop takes longer than one ms of

3. Since the Windows NT kernel is preemptible, we can run the measurement loop inside the kernel as a long-running system call.

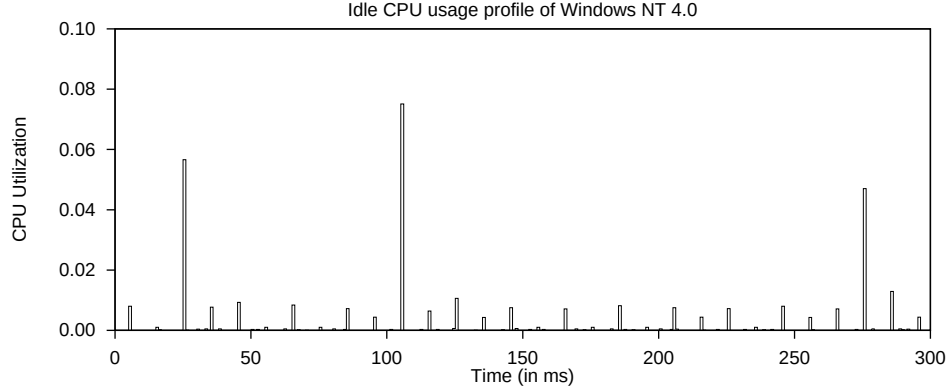


Figure 2: Idle System Measurement. This graph provides a graphical representation of our idle-loop measurement. For each recorded sample, we subtract the one ms of idle time that our busy-wait loop requires to obtain the amount of non-idle activity. During an idle period, this difference will be zero, and we report a CPU utilization of zero. During a long period of computation, substantial computation will occur before the completion of one ms of idle time, and the CPU utilization during that interval will rise.

elapsed time to complete. In this example, a 10 ms interval between two samples means that the system was busy for $(10 - 1) = 9$ ms during the interval.

The value of N can be chosen based on the precision desired and available buffer space. The larger we make N , the coarser the accuracy of our measurements; the smaller we make N , the finer the resolution of our measurements but the larger the trace buffer required for a given benchmark run.

3.2.2 Idle system profiles

In this section, we present measurement results for the background activity that occurs when the system is handling neither user-initiated activities nor network-related activities. This provides intuition about the measurement technique as well as baseline information. Figure 2 shows the system profile of an idle system. To relate non-idle time to elapsed time, we plot elapsed time on the X-axis and the CPU utilization on the Y-axis. Given that each sample represents 1 ms of idle time, the average CPU utilization during a sample interval can be calculated easily. For example, if the system spent 10 ms collecting

a sample, and the sample includes 1 ms of idle time, the CPU utilization for that time interval is $(10 - 1)/10 = 90\%$.

The graph shows that the most of the bursts of CPU activity occur at 10 ms intervals. These bursts are due to hardware clock interrupts. Correlating the samples with a count of hardware interrupts from the Pentium performance counters shows that each burst of computation is accompanied by a hardware interrupt. Some of these bursts involve more computation than the others suggesting that the system was performing extra tasks, such as process priority recalculation, during the interval. The graph also contains noise that shows up as bursts of extremely small (less than 0.001 CPU utilization) CPU activity. These data points are due to our methodology miscounting the time dilation due to hardware events, such as TLB and/or cache misses, as CPU activity.

3.2.3 Window open

With the idle behavior of the system identified, we present how the system consumes the CPU time when it is performing a more complicated task. It is becoming common for interactive systems to perform extra work, such as animation, for the user's "viewing pleasure." The following experiment shows the effects of such features and how such features can cause throughput-based benchmarks to yield less useful and difficult-to-understand results. We measured the system activity generated when maximizing a previously minimized window; the system displays a window that increases in size gradually as it is maximized. In order for the animation to be visible to the user, the system must pace itself, inserting delays. These delays cause the system to become idle, resulting in multiple

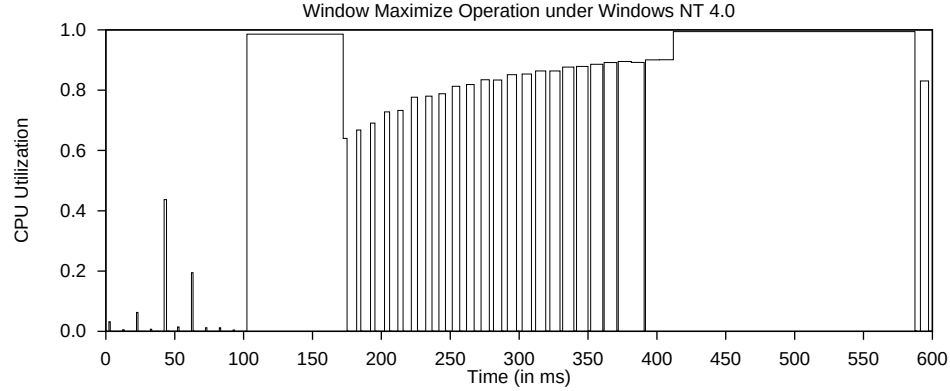


Figure 3: CPU Usage Profile for a Window Maximize Operation. This graph shows the measurement of computation generated when a window is transformed from an icon in the NT 4.0 icon bar to an open window. The short spikes between 180 ms and 400 ms correspond to the animation that is used as the window outline moves from the menu bar to its destination. The upward slope occurs as the outline increases in size. The period of continuous computation from 400 ms to 600 ms corresponds to the final redraw computation for the window.

idle-loop measurement samples that correspond to the single user event (i.e., maximize window).

Figure 3 shows the result of our measurement. At 100 ms we introduce a simulated mouse click into the system using Microsoft Visual Test [23]. The figure clearly shows the 80 ms of 100% CPU utilization required to process the input event (from 100 to 180 ms) and another period of 100% CPU utilization starting around 400 ms to re-display the page. The most interesting portion of the graph is the stair pattern between 180 and 400 ms. This pattern corresponds to the CPU activity required to perform the animation. We can observe that the bursts of CPU activity for performing animation are aligned on 10 ms boundaries, suggesting that they are scheduled by clock interrupts. Each step of animation takes progressively longer to complete as the window outline increases in size.

This pacing behavior makes a throughput measurement difficult to interpret correctly. In non-interactive systems, which do not insert artificial delays, throughput benchmarks do not suffer from such problems. Artificial delays can make the relationship between system performance and benchmark results less direct by introducing latency that

is unaffected by system speed. To make matters worse, it is often difficult to detect the existence of such artificial delays from throughput benchmark results alone.

In the example above, the duration of the time spent for the animation will not change regardless of the actual system performance. A system that is twice as fast as the measured system will reduce the duration of the initial CPU activity and those due to re-displaying of the page by half, but will have little effect on the duration of the animation. The elapsed time for this event is roughly 500 ms, and 220 ms or 44% of this is used for animation. A simple calculation reveals that the same event will still take $(500 - 220) / 2 + 220 = 360$ ms, instead of 250 ms, in a system twice as fast as the measured system. Similarly, an infinitely fast system still spends 220 ms handling this task therefore showing a performance increase of only 127%. Output of throughput benchmarks become increasingly difficult to understand when features such as animations are present in the system.

3.2.4 PowerPoint pagedown

Some complex, interactive applications can change their behavior depending on the rate at which input events are presented. This further complicates the task of measuring system throughput in a meaningful manner. We measured the cost, as implied by the total CPU time consumed, of handling 10 pagedown requests under two different input rates. We selected Notepad and PowerPoint as the target applications. Notepad is a simple editor program bundled with the Windows NT distribution; PowerPoint is complex presentation creation software included in the Microsoft Office Suite. In this experiment, we found that PowerPoint consumed about 1/5th of the CPU time when requests were deliv-

ered at the fastest possible rate compared to the case in which the requests were introduced into the system at a slower rate. Notepad, which is a much simpler application, did not exhibit such behavior. We conducted this measurement using the system described in Table 2. To reduce the amount of background CPU activity, we removed the machine from the network during the test. We did not observe any disk activity during the test.

Using Microsoft Test, we simulated 10 **PageDown** keystrokes. The first test simulates the keystrokes at the fastest possible rate, while the second test plays back keystrokes with a reduced rate using approximately a 0.5 second inter-arrival time.

To isolate the overhead of simulating keystrokes, we measure the amount of CPU activity generated while delivering keystrokes to a simple program we have written. This program responds to basic window operations such as close and minimize, but ignores events such as keystrokes. We report the sum of CPU busy times during a 6 second interval that contains all the keystroke events in both playback rates. This way, all the test results will include the same amount of timing-based background activities, such as clock interrupts. While the main purpose of this measurement is to isolate the cost of simulating keystrokes, the measured data also include the cost of handling background activity such as clock interrupts and the CPU activity generated as a result of Windows NT accepting simulated keystrokes delivering them to the application. Table 3 shows the mean total CPU busy time during the 6 second intervals under the two different arrival patterns—the data labeled “fast” was collected by playing back the keystrokes as rapidly as possible, the

data labeled “paced” was collected using 0.5 second inter-arrival times. We conducted the test five times. Standard deviations were below 5% of the mean for both cases.

Table 3: Overhead measurement

	Fast	Paced
Total CPU busy time	21.91 <i>ms</i>	37.47 <i>ms</i>

Table 4 shows the total CPU busy time of applications measured under the two different arrival rates corrected for the overhead shown in Table 3. Similar to the results shown above, the reported numbers are the mean of 5 runs. Standard deviations were below 4% of the mean in all cases. Because of the correction, these figures include neither the cost of background activity nor the cost of simulating keystrokes and delivering them to the application. These figures represent the sum of work that the application performed in response to the input events and the amount of work that the system performed due to requests generated by the applications in response to the input.

Table 4: Cost of handling ten pagedown requests

	Fast	Paced
Notepad	242.13 <i>ms</i>	245.14 <i>ms</i>
Powerpoint	99.09 <i>ms</i>	497.73 <i>ms</i>

The table shows that Powerpoint performs much less work when keystrokes are played back at a faster rate while Notepad exhibits no such behavior. This is consistent with our visual observation that Powerpoint skipped displaying some, if not all, of the intermediate pages in the presentation and that Notepad displayed all the intermediate pages in the document.

Powerpoint optimizes away operations overridden by following operations. This behavior is desirable because it allows the application to respond to user input quickly and saves unnecessary computation. However, this behavior is problematic when the applica-

tion is used as a part of the benchmark because the application does not always perform a fixed amount of computation—the amount of computation such an application performs can change depending on the rate of the simulated input and the machine speed. Moreover, the unrealistically high rate of simulated user input used by the benchmark causes some applications to apply too much optimization, causing the benchmark to measure behavior that a system under normal use would never exhibit.

Many of the existing GUI benchmarks measure the throughput of GUI-based systems by measuring how quickly a system processes a sequence of tasks such as typing and screen updates. These tasks are introduced into the system using simulated input as fast as the system can accept it. As a result, the application will behave as if it is responding to user input generated by a super-human, applying far more aggressive optimization to the input stream than possible if the input were created by an actual user.

In the test above, we have demonstrated that a sophisticated application can change its behavior depending on the input arrival rate. Currently, popular methodologies that ignore human physical limitations can yield unrealistic and misleading results. As more and more applications attempt to take advantage of the idle time frequently present in interactive systems to perform non-critical work such as interactive spell checking, this problem is likely to worsen.

The lost-time technique that we used in this section is effective in acquiring detailed CPU usage profile and in some cases, inferring the latency to handle user-initiated events. The effectiveness of this technique as a tool to determine event handling latency is limited, however. The first limiting factor is that this technique is unable to distinguish CPU activity due to handling user-initiated events from background activity and/or activ-

ity generated by unrelated applications. This places severe restrictions on the experimental setup. In order to minimize background activity, we are forced to run just one application in the system. The second limiting factor is that sophisticated programs often generate CPU activity unrelated to the handling of user-initiated events. Features that are designed to take advantage of idle time, such as interactive spell checking and document formatting operations, generate extra CPU activity that is indistinguishable from that due to the latency-critical portion of the event processing. This technique cannot be used to determine event handling latency when a target application exhibits such behavior.

3.3 Importance of expectation

Current measurement techniques fail to account for user expectation. Experts on human-computer interaction have long noted that expectation has a significant effect on user-perceived performance [35]. Given two operations with the same response time, the user's definition of "acceptable" depends heavily on the user's expectation. 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. For example, a user could happily wait for 5 seconds for a program to start up, but the same user would be irritated waiting just one second for a character that s/he typed to echo. This suggests that while latency is more closely related with user-perceived performance than throughput, it alone cannot be used as the indicator of user-perceived performance. In order to measure user-perceived performance effectively, one must incorporate user expectation into the resulting metric or modify the measurement technique accordingly.

Unlike latency or human physical limitations, expectation is difficult, if not impossible, to quantify. This makes it difficult to create a performance metric that takes user expectation into account. However, understanding how these expectations affect user-perceived performance can provide hints on how we might modify measurement techniques to better understand and improve interactive system performance.

One useful observation is that users tend to form expectations by interacting with computers. If a certain operation takes a long time on the user's computer, s/he will learn to expect long delays from those operations. Similarly, if a user is used to a certain operation completing with no perceptible latency, the user will learn to expect this particular operation to complete quickly. Once users are accustomed to a shorter response time, they become hesitant to accept longer delays [35]. Users are frustrated when system fails to meet their expectations.

3.4 Reproducing intermittent problems

Infrequent performance problems have been traditionally ignored because they have little effect on overall system throughput and, more importantly, on throughput benchmarks. However, we cannot afford to ignore such problems in interactive systems where each and every latency that the user experiences can have a great impact on user-perceived performance. A user's frustration or perception of performance is based neither on throughput nor on statistical significance. Even if a performance problem occurs only once in an hour, it can be sufficiently annoying to the user to reduce considerably the system's user-perceived performance.

Recent systems with client-server architectures, such as the X Windows system and some versions of Window NT, are more likely to exhibit intermittent or irreproducible problems than older systems because they often require complex interactions among several agents. It is not at all uncommon for these systems to require services from three or more agents to process a user request⁴. Moreover, most of the recent systems have more than one process present in the system. Tasks such as pageout processing and various logging and network-related services are handled by numerous daemons running asynchronously to the tasks on which the user is waiting. When and if these agents interact in an unexpected manner, competing over a limited system resource, the system as a whole exhibits performance problems, which are usually difficult to reproduce, and therefore, nearly impossible to correct.

Conventional benchmarking/measurement techniques are ill-equipped to diagnose these problems. They measure too little—the result of a benchmark run is often a single number that represents throughput. The lack of detail makes it extremely difficult to understand and explain unexpected measurement results or anomalies. We avoid such anomalies by testing the system with unrealistic configurations, often disabling daemons and disconnecting the machine from the network. These precautions make it less likely that we discover and understand infrequent performance problems.

4. For example, in order for the `ls` command to complete inside an `xterm` window, `xterm`, the X server, command shell, and the `ls` program must all execute.

3.5 Conclusions

Benchmark programs serve two major purposes. The first is to predict the performance of the system under actual use. The second is to highlight performance problems present in the measured system. Current benchmarking techniques do not excel in meeting either goal when applied to interactive systems.

Current benchmarks fail to be optimal predictors of user-perceived performance because they rely on throughput rather than latency. These benchmarks also make incorrect assumptions about how quickly a user can generate requests to stress the system. Such unrealistic input streams can lead to the system behaving in an unrealistic manner, such as applying an optimization to the request stream to the level that cannot be achieved with a real user at the console, making the measurement result less representative of actual performance observed by the users. Another problem of current benchmarks is that they ignore the subjectivity in users' judgement.

Current measurement techniques also fail to help us diagnose intermittent performance problems. Each run of the benchmark yields too little information to sufficiently explain what happened in the system. The importance of stable and repeatable results often forces benchmarking to be done under unrealistic conditions, making it even more unlikely for these benchmarks to identify and correct intermittent problems.

An ideal interactive system benchmarking technique must reflect the subjective judgement of the users. An effective measurement technique will capture enough information to allow us to understand, diagnose, and correct even infrequent problems.

Chapter 4 The Interactive Performance Monitoring Environment

Conventional measurement techniques fail to produce data that help system designers improve user-perceived performance directly. User-perceived performance is roughly based on response time and on users' subjective expectations, not on the throughput upon which conventional measurement techniques place great emphasis. In addition, the importance that conventional measurement techniques place on reproducibility has made them inappropriate and incapable of measuring and understanding intermittent problems.

The goal of TIPME is to enable the identification and correction of performance problems that negatively affect user-perceived performance. To find these problems, we include users in the performance-tuning process by having them inform TIPME when they experience unacceptable response times. These problems may arise infrequently, and their impact on the overall throughput of the system may not be significant, but they do affect user-perceived performance significantly. Conventional measurement techniques, which deliberately remove outliers from consideration, will tend to overlook these situations entirely.

Since the events in which we are interested occur unexpectedly, we use continuous monitoring to gather data about the system state, saving the data to disk only when the user indicates a problem, and we use postmortem analysis to diagnose the cause of the problem. TIPME records process state (if and why processes are blocked), context switch information, how and when events pass through the X-Window server, and the owners of

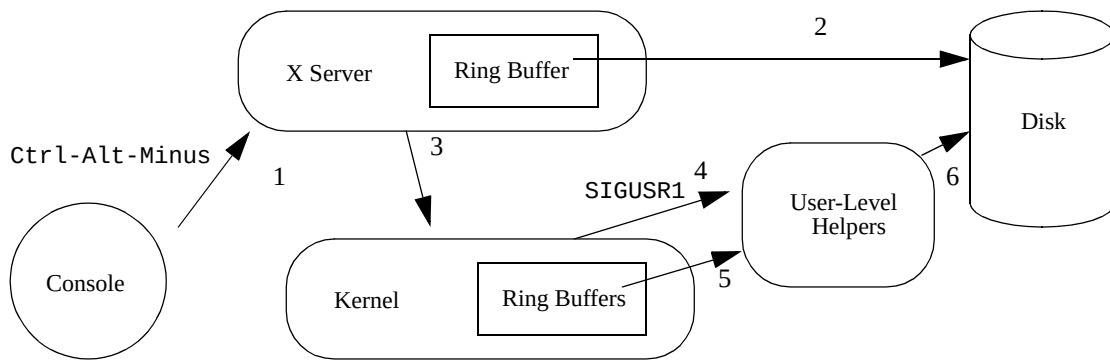


Figure 4: Structure of TIPME. Both the kernel and X server contain non-paged ring buffer(s). Upon experiencing a problem, the user presses a hot-key combination, which is intercepted by the X server (1). The X server writes out the contents of its ring buffer to a file (2) and informs the kernel via a `sysctl` call that the system is experiencing a problem (3). The kernel sends out `SIGUSR1` to all the processes in the system, one of which is a user-level helper that waits forever on `SIGUSR1` (4). In response, the user-level helpers read the contents of the in-kernel ring buffers (5) and write the information to disk (6).

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. 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. Figure 4 shows the overall structure of TIPME.

We have implemented TIPME on BSD/OS 3.0 and X Free86 R6.3 running on Intel Pentium or Pentium Pro-based personal computers. We chose this hardware platform for its popularity and the software platform for its popularity in our environment and the availability of the source code. While the current TIPME implementation is specific to this platform, the techniques and framework are applicable to other platforms. The CPU cycle counter, available in both Pentium and Pentium Pro processors, provides cycle-accurate timestamps on all of the records that TIPME generates [16]. 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.

4.1 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 this 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 the client do not capture the entire processing path. In order to identify the time interval during which the user was waiting for the system to respond, we must determine when a user-initiated event was delivered to the system in the form of a hardware interrupt, how it was transformed into one or more X events, what request(s) the client generated in response, and when the X server finished handling the resulting request(s). To determine when an event begins and ends, we collect information on keyboard and mouse interrupts, the X server state, and the X message exchanges.

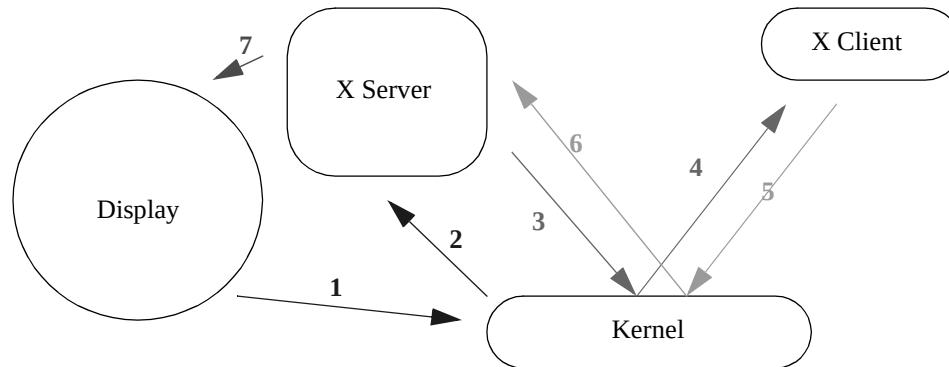


Figure 5: X message exchange. When the user types a key or engages the mouse, the hardware generates an interrupt handled by the kernel (1). The kernel sends a message to the X server (2) which dispatches the event to the proper client, via the kernel (3, 4). The client then processes the request and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

4.1.1 Keyboard and mouse interrupts

Each time a keyboard interrupt is handled, we record the contents of the keyboard buffer. Similarly, we record the contents of the serial buffer to which the mouse is connected when a serial-port interrupt is handled. This information allows us to determine when a user-initiated event entered the system and how promptly the event was retrieved by the X server.

4.1.2 X server state and message exchange

Once the kernel delivers raw input events, such as key up/down events and mouse commands, to the X server, the server turns these events into one or more X events and delivers them to the appropriate client. The client processes these X events, then generates and delivers X requests to the X server, if necessary. Finally, the X server receives and processes these X requests and generates the requested visual feedback. This message exchange is depicted in Figure 5.

We have instrumented the X server to record the arrival and identification of console input, such as keystrokes and mouse input. We also record when and what kind of messages are exchanged between the X server and X clients. Lastly, we record information that helps us determine when all the visible feedback associated with an event are given to the user. We record events such as a mouse pointer update and the processing of X requests. Some X requests, such as `X_ImageText8`, are used to display character strings on the console. By recording when the system finishes displaying a certain text string on the screen, we infer when the user is given sufficient visual feedback to recognize that an event is complete.

This information allows us to determine when the event that frustrated the user began and ended and defines the time interval on which we should focus our analysis. Although much of this identification must currently be done manually, the task can be greatly reduced by instrumenting the client as discussed in Chapter 6.

4.2 Determining the source of the problem

Once we have identified the problematic interval, our next task is to identify the source of the problem. The performance problems that we aim to understand and eliminate cause user-perceivable variability in response time, which are on the order of at least tens of milliseconds. These are most likely due to software rather than hardware. The variations in the execution time introduced by hardware causes, such as memory cache misses and branch mispredicts, are usually on the order of nanoseconds or microseconds per occurrence, so it takes many thousands or millions of such occurrences for the delay to be perceptible. Given a sufficiently complex operation, which all of the operations with user-

perceivable delays are, it is extremely unlikely for the hardware to suddenly experience such an increase. The remaining causes for such variability are 1) a change in the amount of computation that the transaction requires, and 2) a change in the amount of time the process(es) involved in handling a transaction were not running.

We detect changes in the amount of computation that the transaction required using profiling information. To determine if the latency-critical process(es) are executed in a timely manner, we collect the kernel statistics shown in Table 5. In order for latency-critical process(es) to execute, they must first be runnable and the scheduler must decide to execute them. The following information allows us to determine what kind of decisions the scheduler made, which processes were runnable at what point, and if a process is blocked, on what resource the process was blocked.

Table 5: Information Collected

	Description
Process Status	Status (running/runnable/blocked) of processes. Process priority.
Context Switch Information	When processes acquire/release the CPU.
Resource Usage	How highly contested resources are being consumed.
Exec Information	Allow us to associate process IDs with program names.

4.2.1 Process status

Once every two timer interrupts (every 20 ms under BSD/OS), we collect the status of all the processes in the system. In addition to bookkeeping information such as the process ID of the process and its parent, we record which processes were running, which were runnable, and which were blocked and for what reason. We also record the priority of each process in the system. This information allows us to determine whether the system is allowing the processes involved in handling the user request to make progress. In particu-

lar, we look for problems such as a critical process blocked on a wait channel for a substantial amount of time and a runnable, critical process not getting CPU time.

4.2.2 Context switch/Sleep/Wakeup information

We record every context switch, sleep, and wakeup. This provides a finer grain log than the process status information. For example, these records allow us to measure how long it took the system to schedule a critical process once it became runnable, due to an external event such as a keystroke. Additionally, by combining these records with the process status information, we can categorize why processes are prevented from running. The process status information alone does not allow us to differentiate between a resource being held too long and a resource being highly contested, but held for short intervals. In order to determine precisely why a process is not making forward progress, we need the ability to examine these process state transitions.

4.2.3 Highly contested locks and queues

The context switch, sleep, and wakeup logs are useful for determining what kernel resources are highly contested. Once we have identified these resources, we record the owner and waiters for each such contested resource, so we can understand the cause of the contention. For example, as we will show in Section 5.3.1, the VM system's pre-allocated `buf` structures are frequently highly-contested, so we record the owner of the structures at every operating system clock tick. Recording such information enabled us to discover that the VM pageout daemon was often monopolizing all the `buf` structures in order to page out dirty pages, making the structures unavailable for page fault processing.

Another example of what we currently monitor is the disk queues. This allows us to determine the cause when latency critical disk requests are not serviced promptly. The list of monitored resources is expected to grow as we discover more contested resources. In a production system, we suggest monitoring all resources and queues and designing the collection mechanism so that statistics collection for a particular resource can be easily enabled/disabled.

4.2.4 Exec information

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).

4.3 Implementation details

There are three major components in TIPME—the kernel components, the X server components, and the user-level helpers. The kernel components collect the operating system statistics described in Section 4.2; the X server components record X server statistics and the message exchanges between the kernel and the X server and between the X server and X clients (Section 4.1); the user-level helpers tie the other TIPME components together and provide the interface to control TIPME and extract the information col-

lected. In order to keep the measurement system tractable, we do not require any instrumentation of client programs. The following subsections explain each of the three components in detail.

4.3.1 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 dynamically allocate space is when recording exec information, because the length of the command-line arguments and the size of the environment is highly variable.

We have modified the console driver to trap `Ctrl-Alt-Plus`, `Ctrl-Alt-Minus`, `Ctrl-Alt-0`, and `Ctrl-Alt-1` combinations. Pressing `Ctrl-Alt-Plus` causes TIPME to start the monitoring system. `Ctrl-Alt-Minus` notifies TIPME that the user has experienced unacceptable performance and instructs the kernel to send every process in the system a `SIGUSR1` signal, which, in turn, causes the user-level helper to retrieve and save the contents of the TIPME buffer.⁵ Since some programs will exit when receiving an unexpected signal, we have added a `sysctl(5)` option to disable signal delivery to particular processes. `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!

5. We send the `SIGUSR1` to all processes in the system both to make sure that processes such as the X server write their data to disk and to plan for a time in the future when clients may be instrumented to collect statistics. In this case, when TIPME exits, we want all such processes to write their data to disk.

4.3.2 X Server modifications

The X server portion of TIPME uses 6MB of nonpageable (wired) memory for its ring buffer. Events such as the arrival of a character from the keyboard, an X event structure sent to a client, 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 used in graphics mode using 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-downs and key-ups. While the console is operating in this mode, 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.

4.3.3 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 the data from the in-kernel buffer to user space. The helper then writes this data to disk.

The postmortem analysis is performed by about a dozen perl scripts linked with the Berkeley DB package [36], which processes the raw data, generating human-readable reports.

4.4 The overhead of TIPME

As mentioned in Section 4.3, 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. A TIPME-instrumented kernel and X server have larger code sizes, but the code expansion is minimal (9KB and 22KB, respectively). 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.

Table 6: Typical cost of generating a TIPME record

Record Type	Cost (in <i>us</i>)
X server related events	17.1 – 64.6
Process states and buf structure usage recorded during timer interrupts including	106.7 – 158.0
Context switch information	0.5 – 3.5
Keyboard interrupt	2.1 – 3.6
Mouse interrupt	2.1 – 3.7

The runtime overhead of TIPME is low and reasonably constant. Table 6 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 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 ending of the event interval. Table 7 shows some typical latencies and TIPME overheads for common events measured using Muenster, a personal computer based on 100Mhz Pentium processor. As can be seen from Table 7, the TIPME run-time overhead can be significant for operations with a short latency. However the overhead is negligible when compared to the human’s perceptual limitation, which is on the order of tens of milliseconds [35].

Table 7: Latency and TIPME overhead of common events

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

4.5 Sample output

In this section, we show and describe the data collected by TIPME while the system was handling two consecutive mouse pointer move requests. The output shown below is the human-readable version generated by the user-level data-processing scripts that process raw TIPME output. Lines that begin with a ‘#’ character are comment lines. Unindented line shows the time at which these events were recorded (in seconds) and the cost of generating the record (in microseconds). These timestamps are obtained from the Pentium/Pentium Pro cycle counter [16], so the time has no relation to actual time of the day. Indented lines describe the event recorded.

```
#
# <Time index> <TIPME overhead>
# <Record information>
# <Extra record information>
#
2016.7231 sec cost 3.8 us
    Mouse interrupt. 4 bytes <40><01><00><40>
2016.7232 sec cost 0.5 us
    Pid 693 resumed execution
2016.7233 sec cost 31.2 us
    X select returned 1 channels ready 1 for reading, 0 for writing
2016.7234 sec cost 23.0 us
    X server retrieved mouse input
    40:01:00:40
2016.7235 sec cost 19.2 us
    X mouse pointer updated
2016.7236 sec cost 20.7 us
    X calling select waiting for input
2016.7236 sec cost 1.8 us
    Pid 693 going to SLEEP
```

The output above shows the sequence of events that TIPME observed as one mouse move request entered and was processed by the system. The mouse interrupt was delivered to the kernel interrupt handler at time index 2016.7231. The X server acquired the CPU⁶ and returned from *select(2)* at 2016.7232 and 2016.7233, respectively. The X

6. The process ID of the X server is 693. This translation is kept in a separate file not shown here.

server then read the mouse movement data from the kernel at 2016.7234 and finished updating the mouse pointer at 2016.7235 and called *select(2)* and went to sleep at 2016.7236. By subtracting the time index at which the mouse interrupt entered the system (2016.7231) from the time index at which the mouse pointer was updated (2016.7235), we can calculate that the latency of handling the mouse pointer update is about 0.4 ms.

The listing above also shows the cost of generating each record in microseconds. By accumulating the individual costs of generating all the records between the mouse interrupt arrival record and mouse pointer update record, excluding the pointer update record itself (since this record was generated after the system updated the mouse pointer), we determine that TIPME is responsible for 58.5 us. or about 15% of the 0.4 ms latency recorded. As previously discussed, the TIPME overhead for short latency events such as this is significant. However, we find this overhead acceptable since a latency of 0.4 ms. is well below the limits of human perception [35], and therefore, will not alter the way in which the user interacts with computer nor will it be an interesting target for actual analysis.

During each operating system clock interrupt, TIPME records pieces of operating system state including which process, if any, is currently running, the status of all the processes in the system, and the ownership of highly contested resources in the system. The following output shows the information recorded during a clock interrupt immediately following the handling of the mouse pointer update.

```
2016.7295 sec cost 116.6 us
    TICK pid:  idle,  44 procs present
```

The sample above shows that the clock interrupt occurred at time index 2016.7295. At this time, the system was idle, and there were 44 processes present in the system. This output also shows that TIPME spent 116.6 us. collecting this information.

The display below shows the status of all the processes present in the system. All the records show a negative cost value to indicate that the cost of recording these records has already been reported. If a particular process is blocked on a wait channel, we record and report the address and the ascii description of the wait channel.

```
2016.7295 sec cost -116.6 us
    TICK pid: 715 is blocked on f29da0e8(ttyin)
2016.7295 sec cost -116.6 us
    TICK pid: 714 is blocked on f29da000(ttyin)
2016.7295 sec cost -116.6 us
    TICK pid: 710 is blocked on f01130d8(select)
2016.7295 sec cost -116.6 us
    TICK pid: 709 is blocked on f01130d8(select)
2016.7295 sec cost -116.6 us
    TICK pid: 704 is blocked on f01130d8(select)
2016.7295 sec cost -116.6 us
...

```

TIPME also captures the state of highly contested resources during each operating system clock interrupt. This version of TIPME was configured to track the ownership and the usage details of the `SWPGIOBUF` structure, which is used to describe the details of a VM paging I/O request sent to the disk. In this kernel, there are 56 such buffers. We number them 0 to 55 for reporting purposes.

For each buffer, we report whether the buffer is free or in use. We also report the contents of the flag field of the `SWPGIOBUF` structure, which provides additional information about the particular I/O operation for which the structure is used, such as whether the access is read or write. Free buffers are reported as “available” as shown below:

```
2016.7295 sec cost -116.6 us
    TICK SWIO buffer 0 available flag = 00000000
2016.7295 sec cost -116.6 us
    TICK SWIO buffer 1 available flag = 00000000

```

```

2016.7295 sec cost -116.6 us
    TICK SWIO buffer  2 available flag = 00000000
2016.7295 sec cost -116.6 us
    TICK SWIO buffer  3 available flag = 00000000
...
2016.7295 sec cost -116.6 us
    TICK SWIO buffer 54 available flag = 00000000
2016.7295 sec cost -116.6 us
    TICK SWIO buffer 55 available flag = 00000000

```

During this test, TIPME did not record instances in which a SWPGIOBUF structure was in use, which is consistent with the visual observation that system did not experience disk activity during the measurement period. However, if we had experienced disk activity during the test, TIPME would have generated a record similar to the one shown below. For each structure that is occupied, we report the owner of the structure by showing its process ID. We also report the size of the I/O operation that the buffer is used to represent. Sample output is shown below:

```

2016.7295 sec cost -116.6 us
    TICK SWIO buffer 26 owned by 2 handling 4096 byte IO flag = 00100010

```

The information described above is recorded once every two operating system clock interrupts. TIPME spent 116.6 us. collecting this information. Under BSD/OS, operating system clock interrupts occur every 10 ms.

The output below shows the second mouse move request. The processing of this request completed in about 0.3 ms. of which TIPME was responsible for 52.5 us. or about 17% of the latency.

```

2016.7530 sec cost 1.9 us
    Mouse interrupt. 4 bytes <00><01><40><00>
2016.7530 sec cost 0.5 us
    Pid  693 resumed execution
2016.7531 sec cost 27.9 us
    X select returned 1 channels ready 1 for reading, 0 for writing
2016.7533 sec cost 22.2 us
    X server retrieved mouse input
    00:01:40:00
2016.7533 sec cost 19.6 us

```

```
X mouse pointer updated
2016.7534 sec cost 20.5 us
X calling select waiting for input
2016.7535 sec cost 1.8 us
Pid 693 going to SLEEP
2016.7535 sec cost 0.6 us
Pid 693 about to lose CPU
```

4.6 Limitations

We designed TIPME to be used in an environment where users have their own machines and perform most of their daily computation on a personal machine as the console user. Since our purpose is to diagnose the source of user-perceivable delays, the data that TIPME collects are unsuitable for diagnosing the causes of sub-millisecond delays. We do not break down a network-related delay into its subcomponents. For example, if a client program blocks for a long time waiting for a service from a remote server, TIPME output will show the delay only as a network-related delay. A complete breakdown of the delay requires us to deploy a TIPME-like infrastructure on the server and coordinate the data collection process. This is left for future work.

Since our approach is software-based, we are often unable to determine the cause when a device, such as a disk, misbehaves. TIPME will identify when a misbehaving device is the cause of a long latency, but cannot determine why the device behaved in such a manner.

4.7 Conclusions

The definition of performance is unique under interactive systems in that it is decided by users' perceptions, not by an easily quantifiable metric such as throughput. TIPME is designed to improve user-perceived performance by helping us diagnose and correct performance problems that plague interactive users.

To capture infrequent and unexpected problematic situations, TIPME continuously collects operating system and X server state. Storage requirements are kept reasonable by discarding collected information after a specified interval has elapsed. The runtime overhead of TIPME can be a significant percentage for events with short (<10 ms) latencies, however, we can safely ignore this overhead, for these events themselves are not perceptible [35] and not likely to annoy users or affect the way in which the users interact with the system. For events with longer, possibly human perceptible latencies, TIPME overhead is insignificant. To identify unlikely situations in which TIPME is a major contributor to the response time, TIPME records the cost of generating each and every record. During a postmortem analysis, we calculate the total overhead of TIPME based on this information and ensure that TIPME did not cause the performance problem that the user experienced.

Chapter 5 **Measurement results**

In this chapter, we demonstrate TIPME’s utility in identifying system problems that lead to poor user-perceived performance. We will present the details of the performance problems we have discovered using TIPME to continuously monitor the states of the system. As intended, TIPME was successful in collecting information necessary for diagnosing performance problems that would have been extremely unlikely to have been discovered using conventional measurement techniques. These problems are the results of delicate and unexpected interactions between the processes and the services in the system making them difficult to reproduce using conventional benchmarking techniques unless the exact cause of the performance is already known.

In Section 5.1, we describe the general measurement strategy we used. We describe the experimental setup in Section 5.2. In Section 5.3, we describe the performance problems we discovered and how we used TIPME to determine the cause of the problems. We discuss our experience using TIPME as a diagnostic tool in Section 5.4. and present our conclusions in Section 5.5.

5.1 General strategy

The purpose of TIPME is to collect sufficient system state information to enable the diagnosis of performance problems that annoy users. To collect information that enables the diagnosis of these problems, we deployed TIPME on several desktop machines and continuously monitored the system. We instructed the users to indicate their

annoyance with system performance by pressing a hot-key combination that causes TIPME to write the contents of its buffers to disk.

We then examined the data collected. Most of the reported performance problem instances showed that the process or the processes involved in servicing the user-request did not execute in a timely manner. It was expected that the causes of these problems were *not* the raw processing power of the hardware. If the latency of processing were limited by the raw processing power of the hardware, the event would have consistently caused long latency. Users tend not to report such events as performance problems, because they have grown accustomed to the long latencies. They expect and accept such long latencies as the limitation of the hardware that they are using. In this work, we do not concern ourselves with consistent performance problems since these are exactly the sort of problems for which the use of conventional throughput-based benchmarks are effective.

Using the collected data, we determined what kept the latency-critical processes from executing. If the processes were not runnable (e.g., waiting for a VM page or completion of an I/O operation), we identified the exact cause and examined if such conditions can be removed or optimized. If the process was runnable, we examined why the scheduler made a suboptimal decision.

In some instances, it was necessary to modify TIPME to collect additional information. For example, if a performance problem is caused by the latency-critical process blocking waiting for a resource such as a preallocated buffer space, it is not sufficient just to determine on which resource that the process is blocked. We must collect information that indicates how the resource is allocated and held by other agents in the system. This

necessitates that we redeploy an updated version of TIPME and wait for the problem to reappear.

Once we determined the exact cause of the performance problem, we use our newly acquired understanding of the problem to construct microbenchmarks that recreate the problematic condition. It is extremely unlikely that one would construct such benchmarks by chance. One must understand the exact cause of the performance problem to recreate the system states that triggered the problem. Information collected by TIPME makes it possible to carry out such analysis.

The microbenchmarks serve two purposes. First, they act as a sanity check. We construct these benchmarks to reproduce the conditions that we have determined to be the cause of the performance problem. If our diagnosis is incorrect, it is unlikely that we will be successful in constructing such benchmarks. Second, they serve as a tool to evaluate the effectiveness of solutions that we implement to correct the performance problem. They can also evaluate any future solutions. In addition, we have found it extremely useful to use microbenchmarks along with TIPME. This allows us to determine if the microbenchmark is recreating the system state that we observed in a system under actual use. This also helps us identify the problems in the solution when it fails to eliminate the performance problem it was intended to eliminate.

In summary, the general strategy we use is as follows:

1. collect data when the system is performing poorly
2. identify what is keeping latency-critical process(es) from executing
3. if necessary, add monitoring to further identify the cause of (2) and go back to (1).

4. construct microbenchmarks that reproduce the observed problem
5. use the microbenchmarks to evaluate solutions to the problem
6. use TIPME to pinpoint the problems in the solution when it fails to correct the performance problem.

5.2 Experimental setup

We installed TIPME on three personal desktop machines, Riesling, Shiraz, and Muenster. Table 8 contains the description of the machines used for this experiment. Riesling and Shiraz are used as the primary desktop machines of two graduate students, while Muenster is used as a microbenchmarking and test machine. Both of the graduate students are actively involved in operating system development. The typical tasks performed on these machines are editing, compiling, and web browsing. While Shiraz is used almost exclusively by the person seated at the console, Riesling often supports an extra user who uses the machine as a compute server.

Table 8: Description of the test machines

	Riesling	Shiraz	Muenster
CPU	Pentium Pro 200	Pentium 133	Pentium 100
Motherboard	Tyan S1662	Tyan S1562S	Asus T2P5
L2 Cache	256KB on-module	512KB	512KB
RAM	128MB	64MB	96MB
Harddisk	Quantum TM2110S Seagate ST34520N	Quantum TM2110S	Fujitsu M1606S Seagate ST12400N
SCSI Host Adapter	BusLogic BT-946	BusLogic BT-946	NCR825
Display Adapter	Number 9 Vision 330	Number 9 Motion 331	Diamond Stealth 64 DRAM

We instructed each of the console users to press the hot-key combination when the system responded unacceptably slowly. We also asked the users to record a short description of the performance problem they observed. We performed a postmortem analysis

based on the description of the problem provided by the user and the data TIPME generated.

5.3 Performance problems

On average, these machines perform acceptably. However, during the several month testing period, both users experienced performance problems ranging from sluggish mouse pointer movements to the system becoming completely unresponsive for up to 8 seconds. In this section, we describe how we identified and diagnosed these performance problems. In Section 5.3.1, we discuss how we captured and diagnosed a performance problem in which the console became unresponsive for several seconds. In Section 5.3.2, we explain how we determined the cause of the sluggish mouse pointer movement that the user occasionally experienced.

5.3.1 Multi-second console pause

The first observed problem was the console becoming completely unresponsive for several seconds. These problems coincided with heavy jobs with frequent disk I/O, such as a kernel build. TIPME output 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.

When the VM system initiates a page in or page out I/O, it acquires a `buf` structure, which is used to describe the specifics of the I/O, such as which disk block(s) should be read/written and where in memory the data should be stored/taken from. BSD/OS' VM system maintains a pool that contains a fixed number of these structures⁷. When there is a

shortage of `buf` structures, processes go to sleep on the wait channel `swpgiobuf` waiting for a buffer to become available.

TIPME output showed that these `buf` structures were highly contested. In order to understand how such contention arose, we modified TIPME to record the ownership of these structures. We redeployed TIPME and waited for the problem to reappear. The output from the modified TIPME indicated that during such problematic intervals, the pageout daemon was monopolizing all of the available `buf` structures to initiate 64KB pageout requests. Since no `buf` structures were available to initiate a page in request on behalf of the X server, it was kept blocked, rendering the entire console unresponsive to user input.

5.3.1.1 Reproducing the problem using a microbenchmark

From the TIPME output, we determined that the pageout daemon can monopolize `swpgiobuf` when cleaning VM pages trying to keep up with the memory demand. This information is detailed enough to allow us to construct a microbenchmark that reproduces the exact problem.

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 [16]. 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.

7. The number of preallocated `buf` structures is determined by the amount of physical memory in the system. Riesling and Shiraz pre-allocated 64 and 50 such buffers, respectively.

This benchmark measures how promptly the system can process an event that involves a memory reference that can potentially cause a page fault while the system is experiencing heavy memory pressure. Ideally, the time stamps generated by the timing thread will be about 100 ms apart. Any delay in handling the memory access will lengthen the 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. Unlike the problem discovery phase of this process, we have eliminated unexpected external interference, but we believe this is justified in this case, since we are now artificially trying to reproduce a problem that we observed under normal use. Since the two machines used to discover the problem are in constant use by their console users, we ran the microbenchmarks on the third machine, Muenster, described in Table 8.

We ran the benchmark program varying the number of children from one to eight. When the number of child processes 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 exceeded five 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`.

5.3.1.2 Finding a remedy

In order to prevent the pageout daemon from monopolizing all the VM `buf` structures, we made a quick change to the kernel that makes ten of the `buf` structures unavailable to the pageout daemon. This allows other threads to make forward progress, even under heavy pageout traffic. We reran the microbenchmark, but were disappointed that we did not observe any significant improvement in the system's behavior.

Once again, we used TIPME to monitor the system while it was running the microbenchmark. This time, our data revealed that the measurement thread was waiting for the paging I/O to complete on the `swpgio` wait channel. Examination of the source code revealed that the acquisition of a `swpgiobuf` was soon followed by an I/O operation, which caused the thread to block on `swpgio`. TIPME had identified and allowed us to remove one performance bottleneck, but there was another performance bottleneck involved in this particular performance problem.

The BSD/OS operating system uses the CSCAN [35] algorithm to order disk requests. The CSCAN algorithm is designed to improve disk throughput by ordering disk requests to minimize seeks, but does little to minimize latency. During these problematic intervals, the pageout daemon initiates 30 to 40 separate 64KB writes. The system can stall for several seconds handling a page fault if the faulting read request is placed behind the pageout write requests in the disk queue.

To determine if this problem is correctable, we changed the disk request ordering algorithm into a FIFO with *skip ahead*. Under this simple algorithm, incoming requests are queued in a FIFO manner unless the new request was generated as the result of a page fault. These page-in requests are placed ahead of other requests in the disk queue (i.e.,

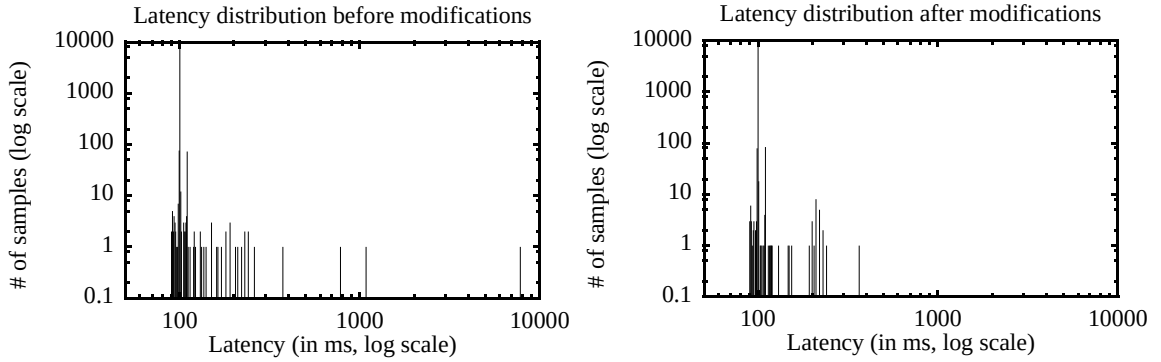


Figure 6: Latency Distributions Before and After System Modification. The left graph shows the latency distribution we observed on the unmodified system using our microbenchmark. Notice that there are several samples with latency longer than one second with one event having latency of about 8 seconds. The right graph shows the latency distribution after changing the buffer allocation strategy and the disk scheduling strategy. These changes have eliminated the long latencies our users experienced.

“skipped ahead”) as long as the following conditions are met: 1) page-in requests do not skip ahead of other page-in requests; 2) when skipping over page-out requests, there must be at least 3 pageout requests ahead in the queue. The goal of this algorithm is to handle page-in requests quickly, and at the same time, allow the pageout daemon to maintain a large enough pool of free physical pages in the system. TIPME output had shown that when the system does not have enough free pages available, threads will block on the `thrd_s` wait channel. By leaving at least three page-out requests in front of the newly queued page-in request, we reduced the performance impact of a thread being blocked on `thrd_s`, while shortening the time during which threads are waiting on `swpgio`. We view this as a short-term solution to demonstrate TIPME’s effectiveness, and do not propose that this is the “right” algorithm to use, in general. Determining the best disk scheduling algorithm for an interactive system is left for future research.

Figure 6 shows the results of the microbenchmark before and after our modifications. The microbenchmark collects 10000 time stamps, thus 9999 intervals, from one trial. The data are shown in the histogram format using 10 ms bucket.

The results show that our simple disk queueing algorithm and small modification to the way VM buffer structures are allocated greatly reduces the severity of the problem our users experienced. 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 down to 839.3 seconds⁸. This was unexpected, but we hypothesize that because this test is single-threaded, the faster pagefault handling was more important than the potentially lower disk throughput. Though we are still unable to demonstrate the negative impact of our changes to system throughput, we expect that these changes will worsen system throughput in some cases. However, we have shown that the throughput-centric disk queueing algorithm actually does cause a very real and user-perceptible performance problem, and that a different and “worse” (with respect to throughput) disk queueing algorithm improves user-perceived performance. Although it is beyond the scope of this thesis, we believe we can design a better queueing algorithm that will provide a compromise between throughput and latency.

5.3.2 Sluggish mouse pointer movement

The second performance problem we captured is the sluggish mouse pointer movement when there was a compile job running in the system. During such problem intervals, the system spent between several hundred milliseconds to one second updating the mouse pointer in response to user input. 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 operations are more latency critical than other opera-

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

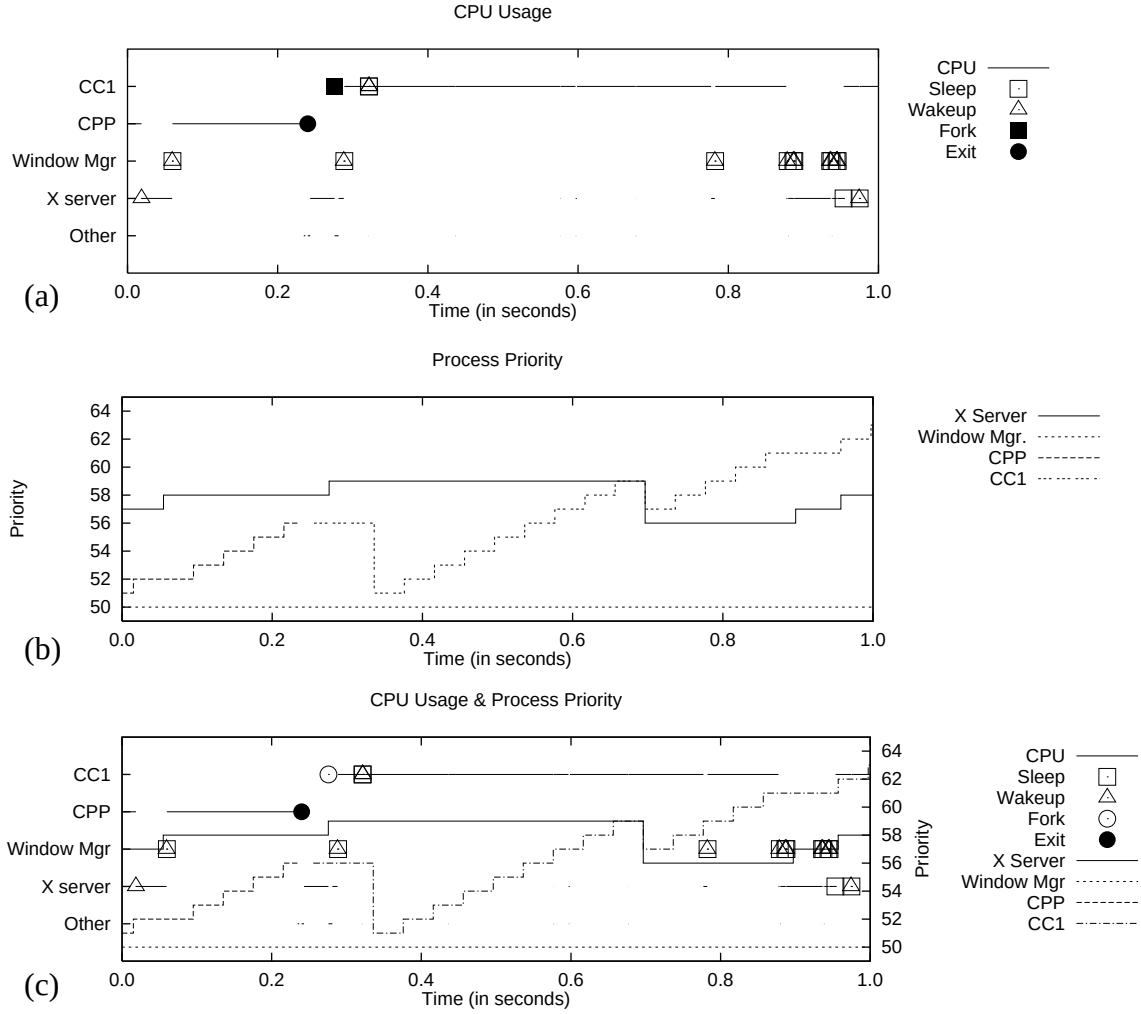


Figure 7: CPU usage and process priority: Above graphs show the priority level of the relevant processes and how they consumed CPU during a problematic time interval. TIPME output shows that the system spent most of its time executing compiler processes instead of the X server because CPU-bound compiler processes were able to attain higher priority levels than that of the X server.

tions such as echoing of keystrokes. According to Shneiderman [35], an acceptable latency for handling mouse movement events are reported to be around 50 to 150 milliseconds.

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. This finding is counterintuitive because BSD/OS UNIX uses a pri-

ority-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, this scheduling scheme was not working as designed.

Figure 7 depicts how CPU time was consumed by the relevant processes and the priority levels of these processes during the problematic time interval. All the graphs show time on the X-axis. The mouse generates a series of data packets that describe the mouse motion. In this particular problem instance, the first of such data packets arrived in the system at time index 0.02. The system processed the first data packet with a good latency of less than a millisecond, but the system spent a substantially longer time providing visual feedback for the subsequent data packets. The longest latency we observed was about 850 ms. At time index 0.95, the X server finally completes the processing of all the pending mouse events and goes to sleep.

Figure 7(a) shows how processes consumed CPU time. Solid horizontal lines denote that the process shown on the Y-axis consumed the CPU during the corresponding time interval. This graph also shows process state changes that affect scheduling decisions such as the birth(fork), death(exit), sleep, and exit. From this graph, we can observe when the X server is awakened in response to the mouse input and when it went back to sleep after completing the mouse input handling. The graph also shows that during this time interval, the system spent most of its time executing compile jobs, CPP (C preprocessor) and CC1 (C compiler), instead of the X server.

Figure 7(b) shows the change in the processes' priorities during this period. Under BSD/OS UNIX, a numerically smaller priority number means higher priority. Each pro-

cess executing user-code is given a priority level (numerically) larger than or equal to 50 based on its past CPU usage. 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 is no runnable process in any of the higher priority classes. Processes within the same priority class are executed in a round-robin fashion regardless of the process priority. Although not shown here, processes are given a higher priority while executing kernel code, for they often hold critical, shared resource, such as locks, inside the kernel.

Although the X server is an interactive process, the priority of the X server is lower than expected at around 58. Most of the other interactive processes such as command-line shells usually show the highest possible user priority level 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 0.33 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 pro-

cesses' true CPU usage characteristics until the process builds up enough CPU usage history for the scheduling algorithm to function effectively.

As a result of this oversight, a newly spawned compile job initially attains a very high priority despite being compute-bound. It takes several hundred milliseconds for the scheduler to build up enough CPU usage history and 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 to cause 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.

Figure 7(c) makes the relationship between process priorities and scheduler decisions clearer by showing the CPU usage information and process priorities together. This graph shows two instances in which the scheduler executes a process with lower priority even though there exists a runnable process with a higher priority. This warrants an explanation. The first of such instances is seen around time index 0.03. The scheduler executes the X server instead of the CPP process even though the CPP process had a higher priority level. This is because the X server was blocked inside the kernel prior to acquiring the CPU. This caused the kernel to assign the X server process a priority level higher than that of any of the processes executing in user-level. As a result, the X server acquired the CPU immediately after it was awoken and was allowed to consume the rest of the scheduling quanta.

The second of such instances is seen beginning around index 0.78. The scheduler executes CC1 even though the X server has a higher priority level. Several factors contribute to this behavior. First, the priority levels of the X server and CC1 are sufficiently close that the scheduler placed both processes into the same priority class. The scheduler executes processes within the same priority class in a round-robin fashion. The second is the timing with which the Window Manager woke up. Because of its high priority level, the Window Manager process interrupted the execution of the X server process immediately after the X server process was scheduled. When the Windows Manager completed its computation and went back to sleep, the system selected a process from the highest priority, non-empty priority class, which contained both X server and CC1. The scheduler executed CC1 instead of the X server, since the X server had been executing just prior to the Windows Manager.

5.3.2.1 Reproducing the problem using microbenchmarks

The cause of the problem is the scheduler granting a newly-created process a high priority regardless of the process' true CPU usage characteristics. A newly-created, compute-bound process is allowed to use up more than its appropriate share of the CPU time until the scheduler collects enough statistics about its CPU usage to adjust its priority accordingly. This process often takes several hundred milliseconds.

This problem is magnified when a stream of new processes are introduced into the system. By continuously introducing compute-bound jobs with high initial priority into the system, a parent process can trick the scheduler into allocating more CPU time to its

compute-bound children starving other processes, including latency-critical ones, in the system.

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. We selected the duration of the computation and sleep such that the priority of the measurement process will 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 such that 10001 timestamps are generated. This yields 10000 intervals between timestamps. The pseudo code for the measurement process is shown below:

```
while (!done) {
    sleep 50 ms.

    for (i = 0; i < N; i++)/* 10 ms. worth of computation */
        ;

    record time stamp
}
```

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 suboptimal decision.

We test how reliably the system schedules the measurement thread using 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 second. 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. This time interval is selected to model a job such as a build that repeatedly spawns compute-bound processes.

We ran this benchmark on the test machine Muenster. In order to prevent unrelated activity from affecting our result, we ran Muenster in single-user mode disconnected from the network during this experiment. Figure 8 contains histograms that show the distribution of the intervals between timestamps for each load conditions. We used a bucket size of one ms to construct the histograms. Figure 8(a) shows the result under first the load condition where there is one, long-running CPU-bound process in the system. Figure 8(b) shows the result under the second condition where a new CPU-bound process is created every two seconds.

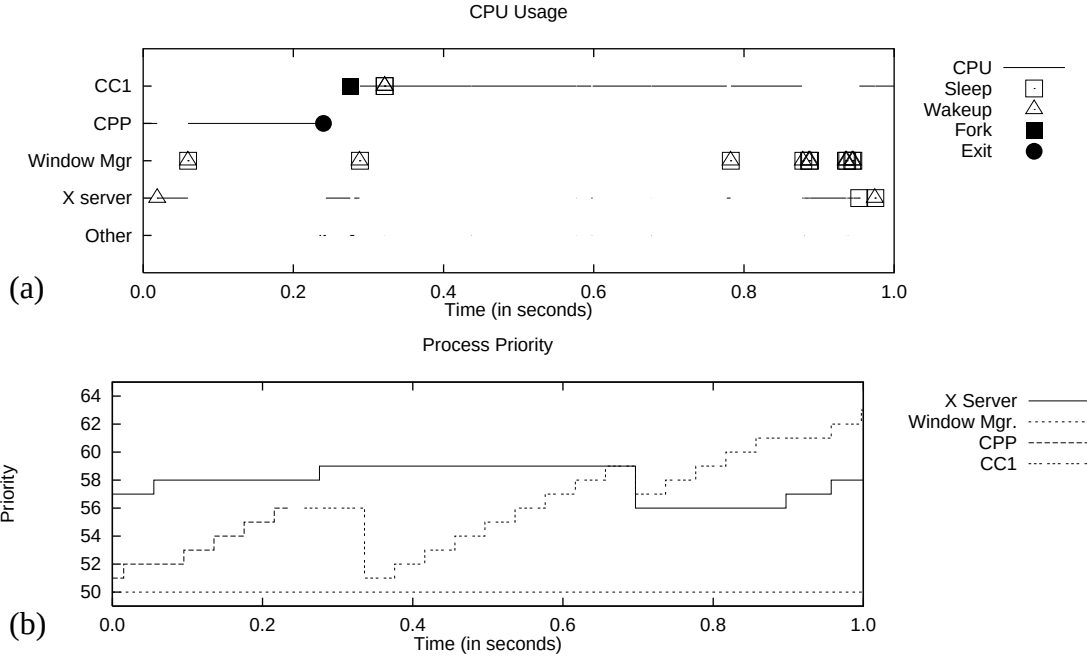


Figure 8: Spawn test: The above graphs show how predictably the system scheduled a process that spent most of its time sleeping when there were CPU bound jobs present in the system. Under ideal conditions, timestamps that the benchmark program generates should be spaced at 60 ms. The system scheduled the measurement thread in an ideal manner when the system was loaded with one long-running, CPU-bound process (Figure (a)). The system experienced problems when the system was loaded with a sequence of short-running, CPU-bound processes (Figure (b)).

Figure 8(a) shows that the system had no problem scheduling the measurement thread reliably. The histogram shows that all the loop iterations took 60 ms. The system was equally successful in executing the measurement thread in a predictable manner when we modified the microbenchmark to add another long-running, compute-bound child process to load the system.

The second load condition successfully recreates the performance problem. Figure 8(b) shows that although most of the loop iterations completed with the ideal delay of 60 ms, there were several hundred iterations that did not. There were seven iterations that required 561 ms to complete. During a separate run of the microbenchmark, we used TIPME to confirm that the benchmark was reproducing the problem we described in Section 5.3.2.

In addition to these two load conditions, we have measured a slightly modified version of the second test to evaluate the impact of the system policy that makes children inherit their initial priority from their parent. In this test, the parent process that spawns compute-bound children performed two seconds' worth of computation between `fork(2)` instead of sleeping in the second test case. This causes the parent process to attain lower priority, which is then inherited by its children. We did not observe any significant change in the result. This result is easily explained by the fact that the priority of a newly created processes is recalculated soon after creation. The BSD/OS operating system performs such recalculations every 40 ms.

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 an I/O-bound process until the process accumulates sufficient CPU usage information, allowing these young processes to interrupt the execution of other processes in the system.

5.3.2.2 Finding a remedy

There are two causes that contribute 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.

Correcting the above problems completely requires an extensive redesign of the system resource management scheme. We believe ideas such as the lottery scheduling [43] and resource containers [2] can be used effectively to tackle this problem. Although finding such a solution is not the target of this study, we still must prevent this problem from occurring in order to find other problems. If the system continues to exhibit this performance problem, users will continue to collect TIPME data that identifies the same problem. In order to prevent this, we modified the kernel such that the priority level of the X server is fixed at 49, one level higher than the highest possible priority level achieved by a process executing in the user-mode. This change kept the problem from reappearing and made the response of the mouse when the system is heavily loaded, such as during a build, perceivably better.

5.4 Discussion

Our experience using TIPME to identify and correct intermittent performance problems indicates that this process is an iterative one. Finding the exact cause of the problem sometimes involves the addition of instrumentation points and re-measurement. The basic set of information outlined in Section 4 has always been successful in narrowing down the possible causes, but in some cases, determining the exact cause of the problem required the collection of additional, problem-specific information. TIPME's framework allows such changes with relative ease, and much of the work involved in adding an instrumentation point can be automated.

The problem we presented in Section 5.3.1 required one such iteration. In this instance, the initial TIPME output showed that the latency-critical thread was blocked because it was unable to acquire buffer space. In order to correct this problem, we needed to understand *why* the thread was unable to acquire the buffer space.

We have also found that constructing microbenchmarks that reproduce observed performance problems is extremely beneficial. These benchmarks can be used to evaluate the effectiveness of possible solutions and also to determine if platforms other than our target system experience problems under similar conditions. In addition, constructing such benchmarks proved to be an effective way to understand the problem further, allowing us pinpoint the exact cause of the problem. We had originally hypothesized the cause of the sluggish mouse movement problem to be the system policy that determines the initial priority level of the child based on the parent's priority level. As discussed in Section 5.3.2.1, we created a modified version of the benchmark so that the child processes are forked from a parent with low priority level. The priority of the parent process proved not to be significant, and thus we were able to determine the true cause of the performance problem.

We have also found that it is necessary to either correct the observed performance problem or apply a quick patch to prevent it from occurring once the analysis is complete. Unless we prevent already-analyzed problems from reoccurring, the users of a TIPME instrumented system will repeatedly present data that identify the same problem.

In addition to diagnosing performance problems that an existing system experiences, we believe that infrastructures such as TIPME should be used in conjunction with the suite of throughput-based benchmarks to evaluate the performance impact of changes

made to the operating system. By testing new changes in systems under normal use, we can ensure that the changes do not introduce undesirable interactive behavior.

5.5 Conclusions

In this chapter, we have demonstrated TIPME’s utility in identifying, understanding, and diagnosing intermittent performance problems. The causes of the problems we observed ranged from easy-to-correct performance bugs (`swpgiobuf` allocation) to ones that require or warrant a fairly extensive change in system policy or structure (disk head scheduling and CPU scheduling). One instance demonstrated the detrimental effects that the throughput-centric performance tuning can have on the latency (disk head scheduling).

More importantly, these performance problems were nearly impossible to identify using conventional benchmarks unless the exact cause of the problem was known. TIPME was successful in collecting information from systems under normal use that allowed us to determine the exact cause of the problem.

Chapter 6 The Generality of TIPME

The current implementation of TIPME is platform specific. However, the methodology we have developed is applicable to other platforms. Platforms other than our target platform also experience intermittent performance problems, which TIPME is specifically designed to diagnose, and some of the problems we have uncovered and reproduced are also present on other platforms. In this section, we discuss the applicability of the TIPME methodology to other systems. First, we discuss TIPME’s hardware and software requirements. Second, we identify the classes of the information that TIPME must extract from the system, and finally, we use microbenchmarks to show that some of the problems we observed on the target platform are present in other systems.

6.1 Requirements

TIPME helps us eliminate intermittent performance problems by capturing the state of the system while it is experiencing a performance problem. We then analyze the information collected to identify the exact cause of the problem, which often is due to the operating system making a suboptimal decision or mismanaging limited resources.

The first requirement for this methodology is access to the operating system source code. While it is possible to facilitate basic data collection by carefully reverse engineering the operating system and by using binary instrumentation tools such as Etch [31] and ATOM [38], source code access is essential for analysis. The data we collect is used to identify problems in the software such as limitations of the operating system policy algorithms and the flaws in operating system implementation itself. This requires an under-

standing of the operating system’s logical structure. This level of understanding is difficult to achieve without studying the source code. It is also true that the source code is often necessary in order to correct identified problems.

As in the example discussed in Chapter 5.3.1, some problems necessitate the collection of additional information. In this particular case, the basic information collected by TIPME identified that the latency-critical process was blocked on a particular resource, but for a complete analysis, it was necessary to collect additional data that show how this resource was consumed. Without source code access, modifying the system to collect such additional information can become an extremely time-consuming task.

The second requirement is the availability of a high-precision, global clock. This clock value is used to label each of the records that TIPME collects from various parts of the system. During the analysis, we use these timestamps for such purposes as identifying the beginning and the ending time of a problematic user event and merging and sorting the data collected from separate parts of the system in chronological order. We found a precision of several microseconds to be sufficient. As hardware-based performance counters are becoming common, this requirement is likely to be met by most of the popular hardware platforms.

6.2 Collecting necessary information

As discussed in Section 4, TIPME collects operating system and X server state for two major purposes: to identify the beginning and ending of problematic intervals, and to provide enough information to make diagnosing the problem possible. In the following sections, we discuss how one might facilitate the collection of such information using

operating system and GUI platforms dissimilar to our target system, the BSD/OS operating system running XFree86.

6.2.1 Determining the problematic intervals

The goal of the TIPME methodology is to identify, understand, and eliminate the causes of long response times that frustrate users. To do so, one must first identify the time interval during which the user was waiting for the system to respond to his/her request. More precisely, we must identify the time at which the user initiated an operation and the time at which the user received necessary feedback from the system for him/her to consider the operation complete.

In the current version of TIPME, we infer the beginnings of events from keystroke and mouse command records and the endings of events from X11 protocol messages and server states. We record the arrival of keystrokes and mouse commands inside the interrupt handlers. By instrumenting these interrupt handlers, we are able to record the input events as soon as the system is notified of their arrival. In order to determine the end point of an event, we modified the X server to record X11 protocol message exchanges and various state transitions inside the X server, such as the handling of X11 requests and input processing. By carefully following the handling of X client requests that involve screen update, such as `X_ImageText`⁹, we ascertain when the final visual feedback associated with the event is handled.

Intercepting and logging keystrokes and mouse events inside interrupt handlers is an accurate, simple, and fairly universal solution to identifying when the user initiated the

9. This request is used to display strings on the screen.

processing of an event. However, identifying when the system completed the processing of the event can be complicated, especially if the target system and application frequently perform extra screen updates that are unrelated to the particular task that the user had initiated, such as blinking cursors and animations. Under such an environment, identifying the end point of an event becomes cumbersome and the result less precise, as we have noted in a previous work [11]. The platform we used and the applications typically used on them do not perform such superfluous screen updates. This made the task of determining the event completion time from X11 protocol message exchanges manageable.

For systems and applications that perform such superfluous screen updates, the task of identifying the event end point is more difficult. In systems such as Microsoft Windows and Windows NT, it is common for the system and/or applications to generate non-critical screen updates. These updates may result from features such as blinking cursors, interactive spell checking, animation, etc. that do not affect, or are not intended to affect, the response time. In such an environment, the assumption that the completion of all the screen update requests signals the end of event handling no longer holds true. In these environments, the only agents, other than the user, that can reliably identify the end of event handling are the client programs. Client programs have sufficient information to determine which screen update requests that they make are latency-critical and which are not. This information is necessary in order to determine when the system completed the handling of an event.

To overcome such difficulties, frameworks such as ARM [15] can be used to leverage the extra knowledge that client programs hold. ARM provides a set of API functions for client programs to call before and after handling an event. By having clients call one of

the API functions immediately after they initiate the last critical screen update, we can easily identify completion of the screen update that signals completion of the event handling. Although client-side instrumentation complicates the measurement process, it may be the only solution to measure and understand performance problems experienced by complex programs.

6.2.2 Aiding diagnosis

With the problematic time interval identified, we examine the various system-state information collected during this interval. Ideally, these system states will identify the exact cause of the performance problem, but determining beforehand the information sufficient to identify the exact cause of many performance problems is nearly impossible. This is why an iterative process of refining the collected states and re-measuring is sometimes necessary. We have, however, identified a set of basic information that either pinpoints the exact cause of the performance problem or at least indicates what extra information we need to collect in order to identify the exact cause of the problem.

In identifying the basic set of statistics to collect, we use a simple yet fundamental observation. In order for a latency-critical process to make progress, it must be scheduled to execute by the operating system. If a process or a group of processes, as often is the case, are responsible for handling a user request, the operating system must execute these processes in a timely manner. Failure to do so results in an unexpectedly long response time that users find irritating. A performance problem can also be caused by a simple lack of processing power, but such problems will be observed consistently. Users will be

trained to expect long latency from experience [35] and will be more tolerant of such problems.

We have identified four types of information as particularly important and useful in determining if the operating system is making appropriate decisions. The first is the status of each process in the system. We need to ascertain whether latency-critical process(es) were running, runnable, or blocked during the problematic interval. If the process(es) were blocked, we must record for what reason the process was blocked. Information that enables us understand the decision made by the scheduler, such as the process priority, makes the determination of the cause of suboptimal scheduling decisions possible.

The second important class of information is the status of queues and locks on which processes frequently block. One example of such queues is the disk queue. A latency-critical process blocked for an unacceptably long time on a queue is a sign that the operating system is managing the queue suboptimally. By recording which processes have created what requests and how the operating system ordered the contents of the queue, we determine if the operating system is managing the queue optimally.

Third, it is useful to collect statistics that helps us determine where the CPU was spending its time during the problematic interval. This is easily accomplished by recording the program counter value during an operating system clock interrupt.

Fourth, we must collect information that enables us to present the above data in a manner that can be understood by a person. We must collect information that allows us to associate process IDs, which are used as the key to most of the in-kernel process informa-

tion, with program names, which humans can more easily understand. We instrument `execve(2)` to associate process IDs and program names.

Although the detail and the frequency of data collection are drastically different, we have found that the basic set of information to collect is similar to that displayed by popular system utilities such as `ps(1)` and `top(1)`. These utilities display information about processes in the system. `Top(1)` periodically re-displays the process ID, process state (i.e., whether the process is running, runnable, or if blocked, for what reason), and a string so that the user can identify to which program a particular entry corresponds. The existence of utility programs such as Performance Monitor, System Monitor and Task Manager under Microsoft Windows platforms shows that the collection of the information we have determined necessary is possible under Microsoft Windows platforms although the lack of access to the operating system source code makes the task difficult.

In summary, the ways in which we gain access to these pieces of information are unique to our platform. However, these types of statistics and the concepts that they represent are applicable to other operating systems.

6.3 Existence of similar problems in other systems

In this section, we examine whether systems other than our target system experience performance problems under similar load conditions. We do so by porting the microbenchmarks we have devised in Section 5.3 to the Win32 programming environment [30] and running them on Microsoft Windows 95 and Windows NT 4.0. These benchmarks will either show that these systems experience similar performance problems demonstrating that these platforms can benefit from our methodology or they do not suffer

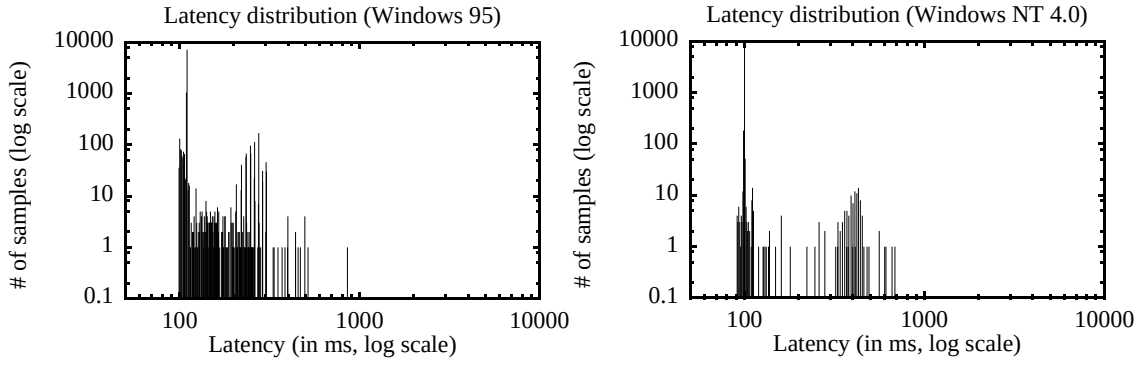


Figure 9: Latency Distribution under Windows 95 and Windows NT 4.0. We executed the microbenchmark described in Section 5.3.1.1 under Windows 95 and Windows NT version 4.0. The left graph shows the latency distribution under Windows 95; the right graph shows the latency distribution under Windows NT version 4.0. Although the longest latency observed is not as long as those observed under our target platform, the latencies are well above the human perceptual limit.

from similar problems. In the latter case, we describe possible reasons using available information.

6.3.1 Reliable response under heavy memory load

The first microbenchmark tests how consistently the system performs a short task introduced every 100 ms when memory-intensive tasks are present in the system. We ported the microbenchmark program used in Section 5.3.1.1 to take measurements under Windows 95 and Windows NT version 4.0. We used the machine Muenster for this measurement. The details of Muenster are shown in Table 8.

Figure 9 shows the measurement results using the same scales employed in Figure 6. Although the longest latency is not as long as those we observed under our target system, there still are cases in which the system does not handle the task in a timely manner. In both Windows 95 and Windows NT measurements, there are cases in which the system takes nearly one second completing the task. Such latency is well above the human perceptual limitation and is likely to negatively affect user-perceived performance.

6.3.2 Priority calculation problem during fork

In this section, we test the microbenchmarks used in Section 5.3.2.1. We devised the microbenchmark to reproduce a problem with BSD/OS process priority calculation algorithm. BSD/OS calculates a process' priority based on the process' CPU usage history, and as a result, the algorithm grants high priority to newly created processes, which have little or no CPU usage history information. These benchmarks measure how reliably the system schedules a process that imitates the behavior of an interactive process by sleeping frequently.

We ran the benchmark under Windows 95 and Windows NT 4.0 running on the machine Muenster. This benchmark consists of a measurement process that imitates the behavior of an interactive process and a process that creates a compute-bound process every two seconds¹⁰. The measurement process executes a loop that performs 10 ms worth of computation after sleeping for 50 ms. We measure how long each iteration of the loop takes by recording a timestamp after the computation phase. Ideally these timestamps are spaced $10 + 50 = 60$ ms.

Figure 10 shows the distribution of the intervals between two timestamps using the same scale used in Figure 8. Figure 10(a) shows the result measured under Windows 95; Figure 10(b) show the result measured under Windows NT 4.0. Although there are instances in which an iteration through the measurement loop took twice as long as the

10. The previously created compute-bound process is terminated prior to the creation of another such process so that there is only one compute-bound process running at any time.

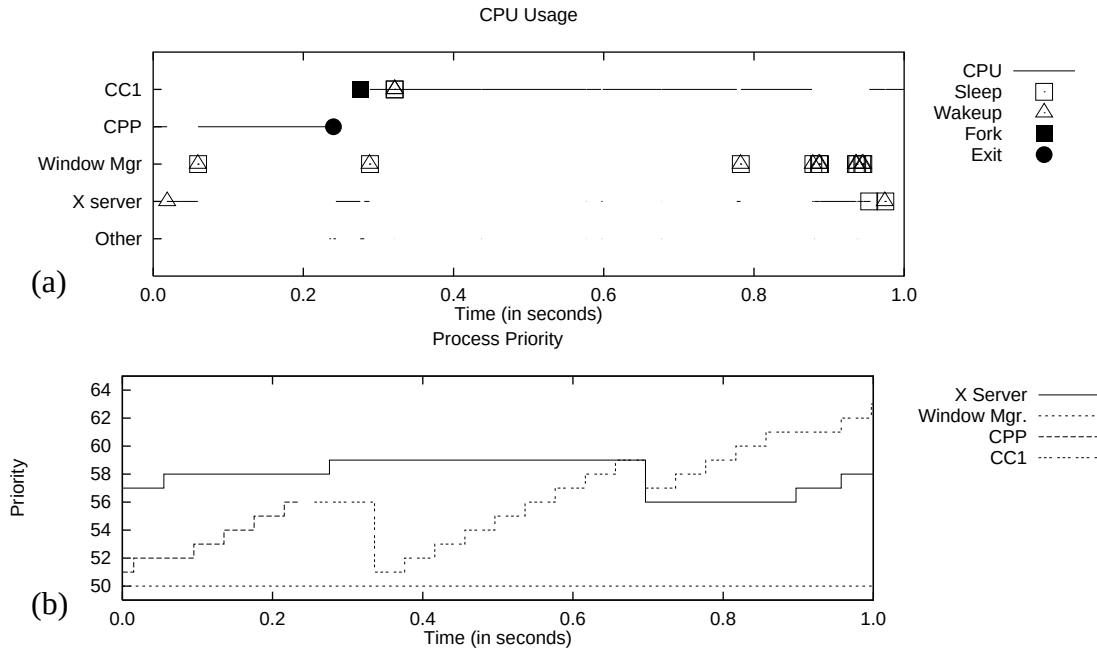


Figure 10: Windows 95 and NT Above figures show how reliably Windows 95 and Windows NT 4.0 executed a process that simulated an interactive process while a stream of new, compute-bound processes are created in the system. The measurement process simulates an interactive process by executing a loop that performs 10 ms of computation after sleeping for 50 ms. We measure how long each iteration of the loop took to complete. Figure 10(a) shows the latency distribution measured under Windows 95; Figure 10(b) shows the latency distribution measured under Windows NT 4.0.

ideal case, both systems performed more reliably than BSD/OS, whose result is shown in Figure 8.

Custer states that the Windows NT scheduler increases thread priority when a thread is unblocked [10], and according to King [17], 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 which have tendencies to block frequently. This is in stark contrast to BSD/OS's scheduling policy, in which the processes can receive an increase in priority only indirectly by not consuming the CPU¹¹.

11. 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.

We believe this policy difference is the reason that both Windows NT and Windows 95 performed better than our target system.

6.4 Conclusions

The current implementation of TIPME is platform-specific. However, the problems that it solves are present in other platforms, and the methodology has broad application. In this section, we identified the software and hardware requirements that the target system must meet in order to host TIPME. We have also classified the types of the information that TIPME must collect and suggested alternative means of obtaining necessary information when the new target system does not allow collection of such information in the same manner as the current implementation.

We then used the microbenchmarks devised in Section 5.3 to evaluate if platforms other than our test platform experience performance problems under similar load conditions. Results from running microbenchmarks showed that other platforms suffer performance problems under similar load conditions. These results demonstrate that platforms other than ours can benefit from a measurement infrastructure such as TIPME.

Chapter 7 Conclusions

In this thesis, we have devised a measurement methodology to improve user-perceived performance. We have identified determining factors of user-perceived performance, described and demonstrated the problems and weaknesses of current measurement techniques. We then determined the possible software causes for response-time variability and developed and demonstrated a measurement methodology that enables system experts to identify the cause. Unlike conventional measurement and tuning techniques that concentrate on improving the average performance, our methodology aims to analyze and eliminate performance problems that introduce response time variability, negatively affecting user-perceived performance.

Response time is a major determining factor of user-perceived performance, but subjective factors, such as user expectations, also affect user-perceived performance greatly. Users are frustrated when their expectation of system performance is not met. More specifically, what negatively impacts user-perceived performance is unexpectedly long latency.

We identified problems with the current interactive system measurement techniques both qualitatively and quantitatively in improving user perceived performance. In general, the conventional measurement techniques calculate system throughput by measuring how quickly a system handles a sequence of simulated user input. In other words, system throughput is derived from the sum of latencies. This technique treats every event latency equally even though some latencies are so short that they are imperceptible and have no impact on user-perceived performance. This technique also pays no attention to

latency variance even though HCI researchers have shown that the predictability of latency is at least as important as the duration.

Throughput benchmarks rely on the assumption that there is a direct relationship between the time spent by the system performing a task and the system performance. In order for this assumption to hold, a given benchmark program must perform a fixed amount of work regardless of the system speed. We have identified two cases in which such an assumption does not hold. We have also shown that, using throughput benchmarks alone, it is often impossible to detect such cases. We accomplished this task by monitoring how interactive tasks consume CPU time. We replaced the system's idle loop with our own and inferred the CPU busy time by measuring the time taken away from the idle loop. We have shown an instance in which a popular application changes the amount of work it performs depending on the arrival rate of the simulated user input, performing less work when input is presented at a high rate. While this behavior is desirable because it allows the system to respond quickly to user input, it makes it difficult to guarantee that a certain benchmark causes the system to perform a fixed amount of work. This problem is expected to worsen given the fact that the practice of running background tasks such as animation and interactive spell-checking that take advantage of idle time is becoming increasingly common.

We have also shown that for tasks such as animation, the system paces itself, spending a fixed amount of elapsed time regardless of the system speed. As a result, the time to complete such tasks has little or no relationship to the system performance. Worse yet, such behavior is often undetectable without detailed CPU usage profile.

We have constructed a measurement methodology to overcome problems of current techniques. Our methodology is centered around the observation that the software causes for response-time variability must be due to change(s) in the amount of one or more of the following: the computation required for the transaction, the time the process(es) spends waiting for I/O or resource availability, and the CPU scheduling decision. We have designed and developed The Interactive Performance Monitoring Environment, TIPME, to detect changes in any of these factors. TIPME continuously collects relevant operating system statistics that help us understand the system behavior during a performance problem. TIPME performs data collection with minimal overhead such that its overhead on user-perceived performance negligible. Using TIPME, we have found and determined the exact causes of performance problems that would otherwise go unexplained even though the problems do actually plague interactive users.

During these problems, the system became unresponsive from several hundred milliseconds to several seconds. Such delays are long enough to be both user-perceivable and annoying. These problems required delicate load conditions to be met in order for them to appear, making it extremely unlikely for them to be discovered and analyzed without using a methodology such as ours.

In both problem instances, the system failed to execute latency-critical processes, the X server in particular, in a timely manner. In once instance, the X server was prevented from executing by buffer contention. Removing this contention revealed that progress of the performance-critical thread was further hampered by the disk head scheduling algorithm designed to provide good throughput but paying little attention to latency. In another instance, an oversight in the process priority calculation algorithm enabled newly created

compute-bound processes to attain unfairly high priority levels. This caused the scheduler to fail to execute the X server for several hundred milliseconds, preventing the X server from handling mouse input in a timely fashion.

The discovery of these problems motivates performance improvement. The determination of the exact cause of the problem enables the creation of better-performing algorithms and enables the creation of microbenchmarks that can evaluate the effectiveness of new algorithms.

7.1 Lessons learned

The definition of performance is unique under interactive systems in that it is decided by users' perceptions, not by an easily quantifiable metric such as throughput. The key to improving user-perceived performance is the elimination of instances in which the user experienced unexpectedly long response time.

Conventional performance measurement and tuning methodologies are most effective in improving a system's average-case behavior but are ill-equipped to analyze cases in which the system fails to provide expected performance. To diagnose such performance problems, we must monitor the dynamic behavior of the system in much finer detail than we do using ordinary benchmarks. We must collect information that enables us to examine every performance-critical decision that the system makes.

For a long time, we have been relying heavily on throughput-based benchmarks and have tuned 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 GUI, has made user-perceived performance more impor-

tant than ever. We must recognize that uncommon cases that have 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.

7.2 Future work

One of the most obvious extensions to this work is introducing the infrastructure to enable measurement of operations that rely on the services of remote servers. Doing so involves running a TIPME-like monitoring system on the remote server and identifying and collecting information that enables us to associate the data collected on the server side with latency observed on the client side. Another is the automation and simplification of the measurement and analysis procedure such that a reasonably experienced user can use the infrastructure to collect performance data and a reasonably competent computer engineer can carry out the analysis. We envision using infrastructures such as this during a beta-test phase of operating system development. We would distribute an operating system along with this infrastructure to collect the performance problem data and use them to remove performance problems that conventional measurement techniques overlook.

7.3 Summary

This thesis proposed and demonstrated a measurement methodology designed to aid the improvement of user-perceived performance by analyzing and collecting instances in which the user found system performance unacceptable. Using this methodology, we determined the exact causes of two such problems. The causes of these performance prob-

lems were not due to the lack of raw processing power of the system but due to correctable software problems.

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?,” *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] Brian N. Bershad, Richard P. Draves, and Alessandro Forin, “Using Microbenchmarks to Evaluate System Performance,” *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE, April 1992, pages 148–253.
- [4] Aaron B. Brown and Margo I. Seltzer, “Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on Intel x86 Architecture,” *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1997, pages 214–224.
- [5] Business Applications Performance Corporation, “SYSmark for Windows NT,” Press Release by IDEAS International, Santa Clara, CA, March 1995.
- [6] 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.
- [7] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Mike Smith, and Margo Seltzer, “The Measured Performance of Personal Computer Operating Systems,” *ACM Transactions on Computer Systems*, Vol. 14, No. 1, February 1996, pages 3–40.

- [8] 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.
- [9] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *The Computer Journal*, Vol. 19, No. 1, 1976, pages 43–47.
- [10] Helen Custer, *Inside Windows NT*, Microsoft Press, 1993.
- [11] 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.
- [12] S. Graham, P. Kessler, and Marshall Kirk McKusick, "gprof: A Call Graph Execution Profiler", *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction*, *SIGPLAN Notices* 17, 6, June 1982.
- [13] Jan L. Guynes, "Impact of System Response Time on State Anxiety," *Communications of the ACM*, March 1988 v.31 n.3, pages 342–347.
- [14] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.
- [15] Hewlett Packard Co. and International Business Machines Inc, "Application Response Measurement," http://www.hp.com/openview/rpm/arm/index_f.html, 1998.
- [16] Intel Corporation, *Pentium Processor Family Developer's Manual. Volume 3: Architecture and Programming Manual*, Intel Corporation, 1995.
- [17] Adrian King, *Inside Windows 95*, Microsoft Press, 1994.

- [18] I. Scott MacKenzie and Colin Ware, "Lag as a Determinant of Human Performance in Interactive Systems," *Proceedings of ACM CHI'83 Conference on Human Factors in Computing Systems*, 1993, pages 488–493.
- [19] Marshall Kirk McKusick, "Using gprof to Tune the 4.2BSD Kernel," *Proceedings of the European UNIX Users Group Meeting*, April 1984.
- [20] 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.
- [21] Larry McVoy and Carl Staelin, "Lmbench: Portable tools for performance analysis," *Proceedings of the 1996 USENIX Technical Conference*, January 1996, pages 179-294.
- [22] Microsoft Corporation, *Microsoft Office 95*, Redmond, Washington, 1995.
- [23] Microsoft Corporation, *Microsoft Visual Test 4.0*, Redmond, Washington, 1995.
- [24] Microsoft Corporation, "Windows 98 Driver Development Kit (DDK)" in Microsoft Developer Network Professional Edition, Redmond, Washington, 1998.
- [25] Netscape Corporation, *Netscape Navigator*, Mountain View, California, 1998.
- [26] Adrian Nye, *Xlib Programming Manual for Version 11*, O'Reilly & Associates, Inc., 1995.
- [27] Paddy O'Donnell and Stephen W. Draper, "How Machine Delays Change User Strategies," *ACM SIG-CHI Bulletin*, Vol. 28, No. 2, 1996, pages 39-42.
- [28] Matt Pietrek, *Windows Internals*, Addison-Wesley, 1993.
- [29] Jeff Reilly, "SPEC Discusses the History and Reasoning behind SPEC95," *SPEC Newsletter*, Vol. 7, No. 3, September 1995.

- [30] Jeffrey Richter, *Advanced Windows: The Developer's Guide to the Win32 API for Windows NT 3.5 and Windows 95*, Microsoft Press, 1995.
- [31] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen, "Instrumentation and Optimization of Win32/Intel Executables Using Etch," *Proceedings of USENIX Windows NT Workshop*, August 1997, pages 1–7.
- [32] Avi Rushinek and Sara F. Rushinek, "What Makes User Happy?," *Communications of the ACM*, Vol. 29, No. 7, July 1986, pages 594–598.
- [33] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt, "The interactive performance of SLIM: a stateless, thin-client architecture," *Proceedings of the Seventeenth ACM Symposium on Operating System Principles*, December 1999, pages 32–47.
- [34] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham, "Experience with Grapevine: The Growth of a Distributed System," *ACM Transactions on Computer Systems*, February 1984, pages 3–23.
- [35] Ben Shneiderman, *Designing the User Interface*, Addison-Wesley, 1992.
- [36] Sleepycat Software, "The Berkeley DB Reference Manual," <http://www.sleepycat.com>, 1998.
- [37] Charles D. Spielberger, R. L. Gorusch, R. Lushene, P.R. Vagg, and G. A. Jacobs, *Manual for the State-Trait Anxiety Inventory*, Consulting Psychologist Press, 1983.
- [38] Amitabh Srivastava and Alan Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1994.
- [39] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall Inc., 1992.

[40] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall Inc., 1987.

[41] 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.

[42] Uresh Vahalia, *UNIX Internals: The New Frontiers*, Prentice-Hall Inc., 1996.

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

[44] Reinhold P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, October 1984.

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